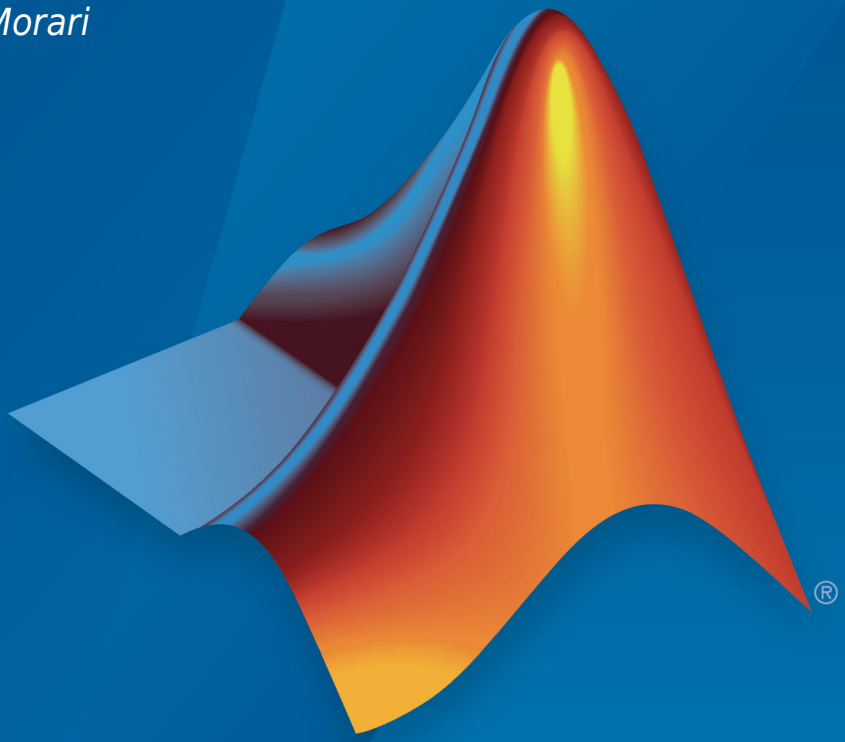
Model Predictive Control Toolbox™

Reference

Alberto Bemporad
N. Lawrence Ricker
Manfred Morari



MATLAB®

R2018**b**

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

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

Phone: 508-647-7000

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

Model Predictive Control Toolbox™ Reference

© COPYRIGHT 2005–2018 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

October 2004	First printing	New for Version 2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.2.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.2.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.2.4 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.1.1 (Release 2009b)
March 2010	Online only	Revised for Version 3.2 (Release 2010a)
September 2010	Online only	Revised for Version 3.2.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.1.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.1.2 (Release 2013a)
September 2013	Online only	Revised for Version 4.1.3 (Release R2013b)
March 2014	Online only	Revised for Version 4.2 (Release R2014a)
October 2014	Online only	Revised for Version 5.0 (Release R2014b)
March 2015	Online only	Revised for Version 5.0.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.1 (Release 2015b)
March 2016	Online only	Revised for Version 5.2 (Release 2016a)
September 2016	Online only	Revised for Version 5.2.1 (Release 2016b)
March 2017	Online only	Revised for Version 5.2.2 (Release 2017a)
September 2017	Online only	Revised for Version 6.0 (Release 2017b)
March 2018	Online only	Revised for Version 6.1 (Release 2018a)
September 2018	Online only	Revised for Version 6.2 (Release 2018b)

Contents

Apps — Alphabetical List

MPC Designer

Design and simulate model predictive controllers

Description

The **MPC Designer** app lets you design and simulate model predictive controllers in MATLAB® and Simulink®.

Using this app, you can:

- Interactively design model predictive controllers and validate their performance using simulation scenarios
- Obtain linear plant models by linearizing Simulink models (requires Simulink Control Design™)
- Review controller designs for potential run-time stability or numerical issues
- Compare response plots for multiple model predictive controllers
- Generate Simulink models with an MPC controller and plant model
- Generate MATLAB scripts to automate MPC controller design and simulation tasks

Limitations

The following advanced MPC features are not available in the **MPC Designer** app:

- Explicit MPC design
- Adaptive MPC design
- Nonlinear MPC design
- Mixed input/output constraints (`setconstraint`)
- Terminal weight specification (`setterminal`)
- Custom state estimation (`setEstimator`)
- Sensitivity analysis (`sensitivity`)
- Alternative cost functions with off-diagonal weights
- Specification of initial plant and controller states for simulation

- Specification of nominal state values using `mpcObj.Model.Nominal.X` and `mpcObj.Model.Nominal.DX`
- Updating weights, constraints, MV targets, and external MV online during simulations

If your application requires any of these features, design and simulate your controller at the command line. You can also run simulations in Simulink when using these features.

When using **MPC Designer** in MATLAB Online™, the following features are not available:

- Designing controllers in Simulink
- Generating Simulink models for your controller and plant

Open the MPC Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `mpcDesigner`.
- Simulink model editor: In the MPC Controller Block Parameters dialog box, click **Design**.

Examples

- “Design Controller Using MPC Designer”
- “Design MPC Controller in Simulink”
- “Compare Multiple Controller Responses Using MPC Designer”
- “Generate MATLAB Code from MPC Designer”
- “Generate Simulink Model from MPC Designer”

Programmatic Use

`mpcDesigner` opens the **MPC Designer** app. You can then import a plant or controller to start the design process, or open a saved design session.

`mpcDesigner(plant)` opens the app and creates a default MPC controller using `plant` as the internal prediction model. Specify `plant` as an `ss`, `tf`, or `zpk` LTI model.

If `plant` is a stable, continuous-time LTI system, **MPC Designer** sets the controller sample time to 0.1 T_r, where T_r is the average rise time of the plant. If `plant` is an unstable, continuous-time system, **MPC Designer** sets the controller sample time to `1`.

By default, plant input and output signals are treated as manipulated variables and measured outputs respectively. To specify a different input/output channel configuration, use `setmpcsignals` before opening **MPC Designer**.

You can also specify plant as a linear System Identification Toolbox™ model, such as an `idss` or `idtf` system. The app converts the identified model to a state-space system, discarding any noise channels. To convert noise channels to unmeasured disturbances, convert the identified model to a state-space model using the `'augmented'` option. For more information on identifying plant models, see “Identify Plant from Data”.

`mpcDesigner(MPCobj)` opens the app and imports the model predictive controller `MPCobj` from the MATLAB workspace. To create an MPC controller, use `mpc`.

`mpcDesigner(MPCobjs)` opens the app and imports multiple MPC controllers specified in the cell array `MPCobjs`. All of the controllers in `MPCobjs` must have the same input/output channel configuration.

`mpcDesigner(MPCobjs,names)` additionally specifies controller names when opening the app with multiple MPC controllers. Specify `names` as a cell array of character vectors or string array with the same length as `MPCobjs`. Specify a unique name for each controller.

`mpcDesigner(sessionFile)` opens the app and loads a previously saved session. Specify `sessionFile` as one of the following:

- The name of a session data file in the current working directory or on the MATLAB path, specified as a character vector or string. To save session data to disk, in the **MPC Designer** app, on the **MPC Designer** tab, click  **Save Session**. The saved session data includes all plants, controllers, and scenarios in the **Data Browser**, the current MPC structure, and the current plot configuration.
- A previously loaded `SessionData` object in the MATLAB workspace. To load a `SessionData` object from a session data file, at the command line, enter:

  ```
  load sessionFile
  ```

See Also

Functions
`mpc` | `sim`

Topics

“Design Controller Using MPC Designer”
“Design MPC Controller in Simulink”
“Compare Multiple Controller Responses Using MPC Designer”
“Generate MATLAB Code from MPC Designer”
“Generate Simulink Model from MPC Designer”

Introduced in R2015b

Functions — Alphabetical List

cloffset

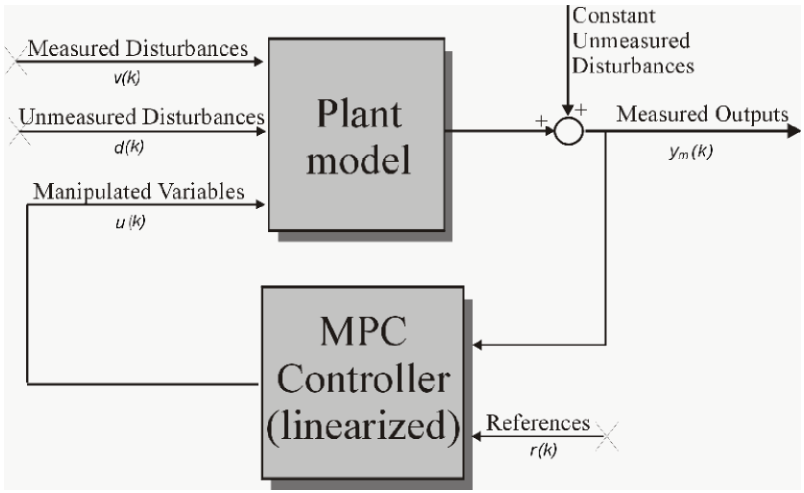
Compute MPC closed-loop DC gain from output disturbances to measured outputs assuming constraints are inactive at steady state

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`

Topics

"Compute Steady-State Gain"

Introduced before R2006a

compare

Compare two MPC objects

Syntax

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

Description

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

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

See Also

`mpc`

Introduced before R2006a

convertToMPC

Convert `nlmpc` object into one or more `mpc` objects

In practice, when producing comparable performance, linear MPC is preferred over nonlinear MPC due to its higher computational efficiency. Using the `convertToMPC` function, you can convert a nonlinear MPC controller into one or more linear MPC controllers at specific operating points. You can then implement gain-scheduled or adaptive MPC using the linear controllers, and compare their performance to the benchmark nonlinear MPC controller. For an example, see “Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant”.

To use `convertToMPC`, your nonlinear controller must not have custom cost or constraint functions, since these custom functions are not supported for linear MPC controllers.

Syntax

```
mpcobj = convertToMPC(nlmpcobj,states,inputs)
mpcobj = convertToMPC(nlmpcobj,states,inputs,MOIndex)
mpcobj = convertToMPC(nlmpcobj,states,inputs,MOIndex,parameters)
```

Description

`mpcobj = convertToMPC(nlmpcobj,states,inputs)` converts the nonlinear MPC controller object `nlmpcobj` into one or more linear MPC controller objects at the nominal conditions specified in `states` and `inputs`. The number of linear MPC controllers, *N*, is equal to the number of rows in `states` and `inputs`.

`mpcobj = convertToMPC(nlmpcobj,states,inputs,MOIndex)` specifies the indices of the measured outputs. Use this syntax when your controller has unmeasured output signals.

`mpcobj = convertToMPC(nlmpcobj,states,inputs,MOIndex,parameters)` specifies the values of prediction model parameters for each nominal condition. Use this syntax when your controller prediction model has optional parameters.

Examples

Create Linear MPC Controllers from Nonlinear MPC Controller

Create a nonlinear MPC controller with four states, one output variable, one manipulated variable, and one measured disturbance.

```
nlobj = nlmpc(4,1,'MV',1,'MD',2);
```

Specify the controller sample time and horizons.

```
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 3;
```

Specify the state function of the prediction model.

```
nlobj.Model.StateFcn = 'oxidationStateFcn';
```

Specify the prediction model output function and the output variable scale factor.

```
nlobj.Model.OutputFcn = @(x,u) x(3);
nlobj.OutputVariables.ScaleFactor = 0.03;
```

Specify the manipulated variable constraints and scale factor.

```
nlobj.ManipulatedVariables.Min = 0.0704;
nlobj.ManipulatedVariables.Max = 0.7042;
nlobj.ManipulatedVariables.ScaleFactor = 0.6;
```

Specify the measured disturbance scale factor.

```
nlobj.MeasuredDisturbances.ScaleFactor = 0.5;
```

Compute the state and input operating conditions for three linear MPC controllers using the `fsolve` function.

```
options = optimoptions('fsolve','Display','none');

uLow = [0.38 0.5];
xLow = fsolve(@(x) oxidationStateFcn(x,uLow),[1 0.3 0.03 1],options);

uMedium = [0.24 0.5];
xMedium = fsolve(@(x) oxidationStateFcn(x,uMedium),[1 0.3 0.03 1],options);
```

```
uHigh = [0.15 0.5];
xHigh = fsolve(@(x) oxidationStateFcn(x,uHigh),[1 0.3 0.03 1],options);
```

Create linear MPC controllers for each of these nominal conditions.

```
mpcobjLow = convertToMPC(nlobj,xLow,uLow);
mpcobjMedium = convertToMPC(nlobj,xMedium,uMedium);
mpcobjHigh = convertToMPC(nlobj,xHigh,uHigh);
```

You can also create multiple controllers using arrays of nominal conditions. The number of rows in the arrays specifies the number controllers to create. The linear controllers are returned as cell array of `mpc` objects.

```
u = [uLow; uMedium; uHigh];
x = [xLow; xMedium; xHigh];
mpcobjs = convertToMPC(nlobj,x,u);
```

View the properties of the `mpcobjLow` controller.

```
mpcobjLow
```

```
MPC object (created on 27-Aug-2018 17:34:57):
---------------------------------------------
Sampling time:      1 (seconds)
Prediction Horizon: 10
Control Horizon:    3

Plant Model:
                                    --------------
      1  manipulated variable(s)   -->|  4 states  |
                                     |             |-->  1 measured output(s)
      1  measured disturbance(s)   -->|  2 inputs  |
                                     |             |-->  0 unmeasured output(s)
      0  unmeasured disturbance(s) -->|  1 outputs |
                                    --------------
Indices:
  (input vector)    Manipulated variables: [1 ]
                    Measured disturbances: [2 ]
  (output vector)        Measured outputs: [1 ]

Disturbance and Noise Models:
        Output disturbance model: default (type "getoutdist(mpcobjLow)" for details)
         Measurement noise model: default (unity gain after scaling)
```

```
Weights:
        ManipulatedVariables: 0
    ManipulatedVariablesRate: 0.1000
             OutputVariables: 1
                         ECR: 100000

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

Constraints:
 0.0704 <= u1 <= 0.7042, u1/rate is unconstrained, y1 is unconstrained
```

Input Arguments

`nlmpcobj` — Nonlinear MPC controller

`nlmpc` object

Nonlinear MPC controller, specified as an `nlmpc` object.

Note Your `nlmpc` controller object must not have custom cost or constraint functions.

`states` — Nominal state values

array

Nominal state values, specified as an N-by-N_x array, where N_x is equal to `nlmpcobj.Dimensions.NumberOfStates`. Each row of `States` specifies a nominal set of states to be used in conversion.

The number of rows in `states` and `inputs` must match.

`inputs` — Nominal input values

array

Nominal input values, specified as an N-by-N_u array, where N_u is equal to `nlmpcobj.Dimensions.NumberOfInputs`. Each row of `Inputs` specifies a nominal set of inputs to be used in conversion.

The number of rows in `states` and `inputs` must match.

`MOIndex` — Measured output indices
`[]` (default) | vector

Measured output indices, specified as a vector of length N_y, where N_y is the number of outputs. If `MOIndex` is `[]`, every output is measured. Otherwise, any outputs not listed in `MOIndex` are unmeasured.

`convertToMPC` uses `MOIndex` to configure the default state estimators in `mpcobj`.

`parameters` — Prediction model parameter values
`{}` (default) | cell array

Prediction model parameter values, specified as an N-by-N_p cell array, where N_p is equal to `nlmpcobj.Model.NumberOfParameters`. Each row of `parameters` specifies the model parameter values for a given nominal condition. In each row, the order of the parameters must match the order specified in the model functions. Each parameter must be a numeric parameter with the correct dimensions; that is, the dimensions expected by the prediction model functions.

For each nominal condition, these parameters are passed to the state function (`nlmpcobj.Model.StateFcn`) and output function (`nlmpcobj.Model.OutputFcn`) of the nonlinear MPC controller.

The number of rows in `parameters` must match the number of rows in `states` and `inputs`.

If your controller prediction model has optional parameters, you must specify `parameters`.

Output Arguments

`mpcobj` — Linear MPC controllers
`mpc` object | cell array of `mpc` objects

Linear MPC controllers created for each nominal condition, returned as one of the following:

- Single `mpc` object when $N = 1$.
- Cell array of `mpc` objects of length N when $N > 1$. Each object corresponds to one nominal condition.

`convertToMPC` copies the following controller properties from `nlmpcobj` to the controllers in `mpcobj`:

- Sample time
- Prediction and control horizons
- Tuning weights
- Bounds on output variables, manipulated variables, and manipulated variable rates
- Scale factors, names, and units for variables and disturbances

If `nlmpcobj`:

- Has unmeasured disturbance channels, then the controllers in `mpcobj` have unity gains for their input and output disturbance models.
- Does not have unmeasured disturbance channels, then the controllers in `mpcobj` have default output disturbance models.

Any state bounds in `nlmpcobj` are dropped during conversion.

See Also

`nlmpc`

Topics

“Nonlinear MPC”
“Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant”

Introduced in R2018b

createParameterBus

Create Simulink bus object and configure Bus Creator block for passing model parameters to Nonlinear MPC Controller block

Syntax

```
createParameterBus(nlmpcobj,nlmpcblk,busName,parameters)
```

Description

`createParameterBus(nlmpcobj,nlmpcblk,busName,parameters)` creates a `Simulink.Bus` object, `busName`, in the MATLAB workspace for passing model parameters to a Nonlinear MPC Controller block, `nlmpcblk`. `createParameterBus` requires you to connect a Bus Creator block to the Nonlinear MPC Controller block in advance so that it can configure the Bus Creator block to use the bus object.

Examples

Create Parameter Bus for Nonlinear MPC Controller Block

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nlobj = nlmpc(4,2,1);
```

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The prediction model uses an optional parameter, `Ts`, to represent the sample time. Specify the number of parameters.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function of the model, passing the sample time parameter as an input argument.

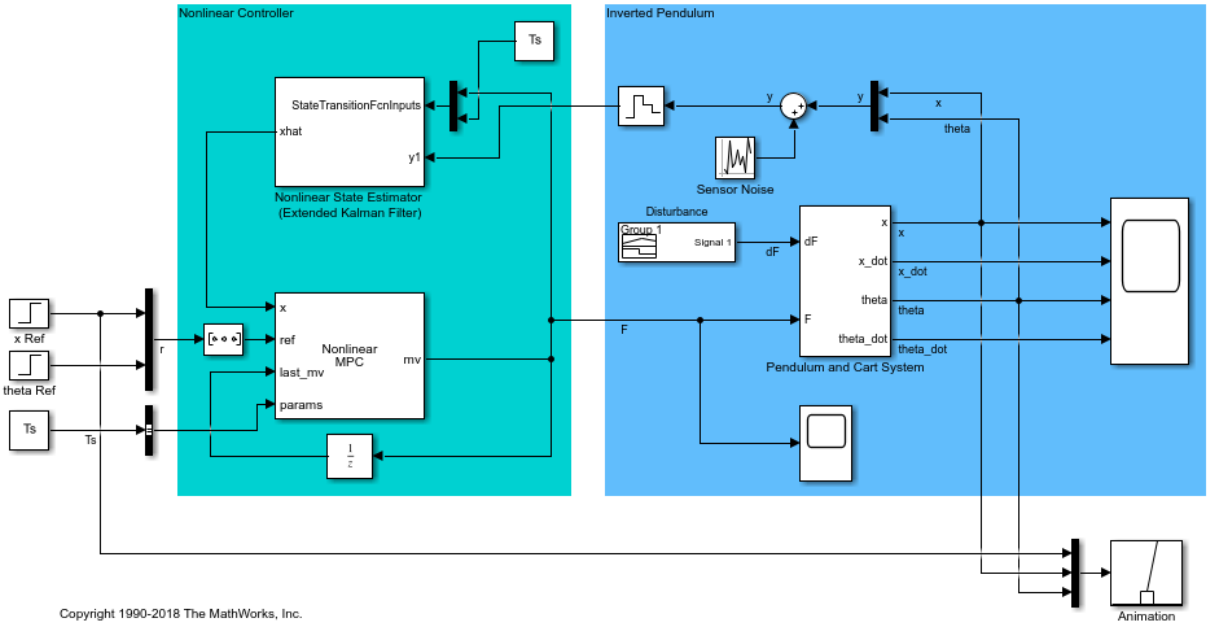
```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Define standard constraints for the controller.

```
nlobj.Weights.OutputVariables = [3 3];
nlobj.Weights.ManipulatedVariablesRate = 0.1;
nlobj.OV(1).Min = -10;
nlobj.OV(1).Max = 10;
nlobj.MV.Min = -100;
nlobj.MV.Max = 100;
```

Open Simulink model.

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



In this model, the Nonlinear MPC Controller block is configured to use the controller `nlobj`.

To use the optional parameter in the prediction model, the model has a Simulink Bus block connected to the `params` input port of the Nonlinear MPC Controller block. To configure this bus block to use the `Ts` parameter, create a Bus object in the MATLAB® workspace and configure the Bus Creator block to use this object. Name the Bus object `'myBusObject'`.

```
createParameterBus(nlobj,[mdl '/Nonlinear MPC Controller'],'myBusObject',{Ts});

bdclose(mdl)
```

Input Arguments

`nlmpcobj` — Nonlinear MPC controller

`nlmpc` object

Nonlinear MPC controller, specified as an `nlmpc` object.

`nlmpcblk` — Block path of Nonlinear MPC Controller block
string | string | character vector

Block path of Nonlinear MPC Controller block, specified as a string or character vector.

`busName` — Name of Simulink bus object
string | string | character vector

Name of Simulink bus object to be created in the MATLAB workspace and set in the Bus Creator block, specified as a string or character vector.

The corresponding Bus Creator block must already be connected to the `params` input port of the Nonlinear MPC Controller block specified by `nlmpcblk`. Also, the Bus Creator block must have the correct number of input ports, and these ports must already be properly connected.

`parameters` — Nominal prediction model parameter values
cell array

Nominal prediction model parameter values, specified as a cell array of length N_p, where N_p is equal to `nlmpcobj.Model.NumberOfParameters`. The order of the parameters must match the order specified in the model functions, and each parameter must be a numeric parameter with the correct dimensions.

See Also

Functions
`nlmpc` | `nlmpcmove` | `nlmpcmoveopt`

Blocks
Nonlinear MPC Controller

Topics
“Specify Prediction Model for Nonlinear MPC”

Introduced in R2018b

d2d

Change MPC controller sample

Syntax

```
MPCobj = d2d(MPCobj,Ts)
```

Description

The `d2d` function changes the sample time of the MPC controller `MPCobj` to `Ts`. All models are sampled or resampled as soon as the QP matrices must be computed, for example when `sim` or `mpcmove` are called.

See Also

`mpc` | `set`

Introduced before R2006a

generateExplicitMPC

Convert implicit MPC controller to explicit MPC controller

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

Syntax

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

Description

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

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

Examples

Generate Explicit MPC Controller

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

Define the double-integrator plant.

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

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

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

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming 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 no disturbance added to measured output channel #1.
-->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-parametric QP computat
Type 'EMPCobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

Input Arguments

`MPCobj` — Traditional MPC controller

MPC controller object

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

`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 4-2 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 ith element contains the details needed to compute the optimal manipulated variables when the solution lies within the ith region. See “Implementation”.

Property	Description
`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.

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.

See Also

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

Topics

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

Introduced in R2014b

generateExplicitOptions

Optimization options for explicit MPC generation

Syntax

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

Description

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

Examples

Generate Explicit MPC Controller

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

Define the double-integrator plant.

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

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

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

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming 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 no disturbance added to measured output channel #1.
-->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-parametric QP computat
Type 'EMPCobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

Input Arguments

`MPCobj` — Traditional MPC controller
MPC controller object

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

Output Arguments

`opt` — Options for generating explicit MPC controller
structure

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

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

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

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

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

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

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

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

Constraint normalization tolerance, specified as a positive scalar value.

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

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

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

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

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

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

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

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

See Also

`generateExplicitMPC`

Introduced in R2014b

generateExplicitRange

Bounds on explicit MPC control law parameters

Syntax

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

Description

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

Examples

Generate Explicit MPC Controller

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

Define the double-integrator plant.

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

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

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

```
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming 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 no disturbance added to measured output channel #1.
-->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-parametric QP computat
Type 'EMPCobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

Input Arguments

`MPCobj` — Traditional MPC controller

MPC controller object

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

Output Arguments

`Range` — Parameter bounds

structure

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

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

The fields of the range structure are as follows.

`State` — Bounds on controller state values

structure

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

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

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

`Reference` — Bounds on controller reference signal values
structure

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

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

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

`MeasuredDisturbance` — Bounds on measured disturbance values
structure

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

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

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

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

`ManipulatedVariable` — Bounds on manipulated variable values
structure

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

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

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

See Also

`generateExplicitMPC` | `generateExplicitOptions` | `mpc`

Introduced in R2014b

generatePlotParameters

Parameters for plotSection

Syntax

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

Description

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

Examples

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

Define a double integrator plant model and create a traditional implicit MPC controller for this plant. Constrain the manipulated variable to have an absolute value less than 1.

```
plant = tf(1,[1 0 0]);
MPCobj = mpc(plant,0.1,10,3);
```

```
-->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
```

```
MPCobj.MV = struct('Min',-1,'Max',1);
```

Define the parameter bounds for generating an explicit MPC controller.

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

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
```

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

```
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;
```

Create an explicit MPC controller.

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

```
Regions found / unexplored:       19/       0
```

Create a default plot parameter structure, which specifies that all of the controller parameters are fixed at their nominal values for plotting.

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

Allow the controller states to vary when creating a plot.

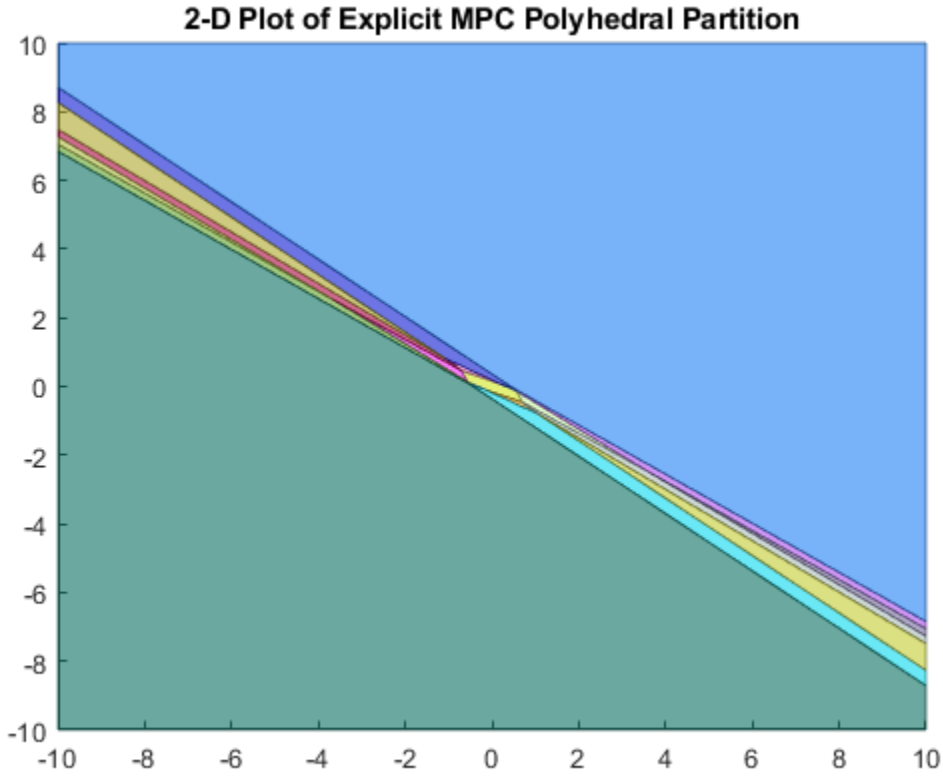
```
plotParams.State.Index = [];
plotParams.State.Value = [];
```

Fix the manipulated variable and reference signal to `0` for plotting.

```
plotParams.ManipulatedVariable.Index(1) = 1;
plotParams.ManipulatedVariable.Value(1) = 0;
plotParams.Reference.Index(1) = 1;
plotParams.Reference.Value(1) = 0;
```

Generate the 2-D section plot for the explicit MPC controller.

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




```
ans =
  Figure (1: PiecewiseAffineSectionPlot) with properties:

      Number: 1
        Name: 'PiecewiseAffineSectionPlot'
       Color: [0.9400 0.9400 0.9400]
    Position: [360 502 560 420]
       Units: 'pixels'

  Show all properties
```

Input Arguments

`EMPCobj` — Explicit MPC controller

explicit MPC controller object

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

Output Arguments

`plotParams` — Parameters for sectional plot

structure

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

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

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

`State` — Fixed controller states

structure

Fixed controller states, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.State.Index` is a vector that contains the indices of the controller states to fix for the plot, and `plotParams.State.Value` contains the corresponding constant state values.

Modify the default value of `plotParams.State` to generate the desired plot. See "Specify Fixed Parameters for 2-D Plot of Explicit Control Law" on page 2-31.

`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 2-31.

`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 2-31.

`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 2-31.

See Also

`generateExplicitMPC` | `plotSection`

Introduced in R2014b

get

MPC property values

Syntax

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

Description

`Value = get(MPCobj,PropertyName)` returns the current value of the property `PropertyName` of the MPC controller `MPCobj`. Specify `PropertyName` as a character vector or string that contains 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.

Tips

An alternative to the syntax

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

is the structure-like referencing

```
Value = MPCobj.PropertyName
```

For example,

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

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

See Also

`mpc` | `set`

Introduced before R2006a

getCodeGenerationData

Create data structures for `mpcmoveCodeGeneration`

Syntax

```
[configData,stateData,onlineData] = getCodeGenerationData(MPCobj)
[___] = getCodeGenerationData(___,Name,Value)
```

Description

`[configData,stateData,onlineData] = getCodeGenerationData(MPCobj)` creates data structures for use with `mpcmoveCodeGeneration`.

`[___] = getCodeGenerationData(___,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments.

Examples

Create MPC Code Generation Data Structures

Create a plant model, and define the MPC signal types.

```
plant = rss(3,2,2);
plant.D = 0;
plant = setmpcsignals(plant,'mv',1,'ud',2,'mo',1,'uo',2);
```

Create an MPC controller.

```
mpcObj = mpc(plant,0.1);
```

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

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

Configure your controller parameters. For example, define bounds for the manipulated variable.

```
mpcObj.ManipulatedVariables.Min = -1;
mpcObj.ManipulatedVariables.Max = 1;
```

Create code generation data structures.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcObj);
```

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

Specify Options for Creating MPC Code Generation Structures

Create a plant model, and define the MPC signal types.

```
plant = rss(3,2,2);
plant.D = 0;
```

Create an MPC controller.

```
mpcObj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->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 code generation data structures. Configure options to:

- Use single-precision floating-point values in the generated code
- Improve computational efficiency by not computing optimal sequence data.
- Use run your MPC controller in adaptive mode.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcObj,...
    'DataType','single','OnlyComputeCost',true,'IsAdaptive',true);
```

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

Input Arguments

`MPCobj` — Model predictive controller
implicit MPC controller object | explicit MPC controller object

Model predictive controller, specified as one of the following:

- Implicit MPC controller object — To create an implicit MPC controller, use `mpc`.
- Explicit MPC controller object — To create an explicit MPC controller, design an implicit controller and then use `generateExplicitMPC`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DataType','single'` specifies that the generated code uses single-precision floating point values.

`InitialState` — Initial controller state
`mpcstate` object

Initial controller state, specified as the comma-separated pair consisting of `'InitialState'` and an `mpcstate` object. This state is used in place of the default state information from `MPCobj`.

`DataType` — Data type used in generated code
`'double'` (default) | `'single'`

Data type used in generated code, specified as specified as the comma-separated pair consisting of `'DataType'` and one of the following:

- `'double'` — Use double-precision floating point values.
- `'single'` — Use single-precision floating point values.

`OnlyComputeCost` — Toggle for computing only optimal cost
`false` (default) | `true`

Toggle for computing only optimal cost during simulation, specified as specified as the comma-separated pair consisting of `'OnlyComputeCost'` and either `true` or `false`. To reduce computational load by not calculating optimal sequence data, set `OnlyComputeCost` to `true`.

`IsAdaptive` — Adaptive MPC indicator
`false` (default) | `true`

Adaptive MPC indicator, specified as specified as the comma-separated pair consisting of `'IsAdaptive'` and either `true` or `false`. Set `IsAdaptive` to `true` if your controller is running in adaptive mode.

For more information on adaptive MPC, see “Adaptive MPC”.

Note `IsAdaptive` and `IsLTV` cannot be `true` at the same time.

`IsLTV` — Time-varying MPC indicator
`false` (default) | `true`

Time-varying MPC indicator, specified as either `true` or `false`. Set `IsLTV` to `true` if your controller is running in time-varying mode.

For more information on time-varying MPC, see “Time-Varying MPC”.

Note `IsAdaptive` and `IsLTV` cannot be `true` at the same time.

Output Arguments

`configData` — MPC configuration parameters

structure

MPC configuration parameters that are constant at run time, returned as a structure. These parameters are derived from the controller settings in `MPCobj`. When simulating your controller, pass `configData` to `mpcmoveCodeGeneration` without changing any parameters.

For more information on how generated MPC code uses constant matrices in `configData` to solve the QP problem, see "QP Problem Construction for Generated C Code".

`stateData` — Initial controller states

structure

Initial controller states, returned as a structure. To initialize your simulation with the initial states defined in `MPCobj`, pass `stateData` to `mpcmoveCodeGeneration`. To use different initial conditions, modify `stateData`. You can specify non-default controller states using `InitialState`.

`stateData` has the following fields:

`Plant` — Plant model state estimates

`MPCobj` nominal plant states (default) | column vector of length n_{xp}

Plant model state estimates, returned as a column vector of length n_{xp}, where n_{xp} is the number of plant model states.

`Disturbance` — Unmeasured disturbance model state estimates

`[]` (default) | column vector of length n_{xd}

Unmeasured disturbance model state estimates, returned as a column vector of length n_{xd}, where n_{xd} is the number of unmeasured disturbance model states. `Disturbance` contains the input disturbance model states followed by the output disturbance model states.

To view the input and output disturbance models, use `getindist` and `getoutdist` respectively.

`Noise` — Output measurement noise model state estimates
`[]` (default) | column vector of length n_{xn}

Output measurement noise model state estimates, returned as a column vector of length n_{xn}, where n_{xn} is the number of noise model states.

`LastMove` — Manipulated variable control moves from previous control interval
`MPCobj` nominal MV values (default) | column vector of length n_{mv}

Manipulated variable control moves from previous control interval, returned as a column vector of length n_{mv}, where n_{mv} is the number of manipulated variables.

`Covariance` — Covariance matrix for controller state estimates
symmetrical *n*-by-*n* array

Covariance matrix for controller state estimates, returned as a symmetrical *n*-by-*n* array, where *n* is number of extended controller states; that is, the sum of n_{xp}, n_{xd}, and n_{xn}.

If the controller uses custom state estimation, `Covariance` is empty.

`iA` — Active inequality constraints
false (default) | logical vector of length *m*

Active inequality constraints, where the equal portion of the inequality is `true`, returned as a logical vector of length *m*. If `iA`(*i*) is `true`, then the *i*th inequality is active for the latest QP solver solution.

Note Do not change the value of `iA`. Always use the values returned by either `getCodeGenerationData` or `mpcmoveCodeGeneration`.

`onlineData` — Online controller data
structure

Online controller data that you must update at each control interval, returned as a structure with the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>signals</code></td><td>Input and output signals, returned as a structure with the following fields.

<table><tr><th>Field</th><th>Description</th></tr><tr><td><code>ym</code></td><td>Measured outputs</td></tr><tr><td><code>ref</code></td><td>Output references</td></tr><tr><td><code>md</code></td><td>Measured disturbances</td></tr><tr><td><code>mvTarget</code></td><td>Targets for manipulated variables</td></tr><tr><td><code>externalMV</code></td><td>Manipulated variables externally applied to the plant</td></tr></table></td></tr>
<tr><td><code>limits</code></td><td>Input and output constraints, returned as a structure with the following fields:

<table><tr><th>Field</th><th>Description</th></tr><tr><td><code>ymin</code></td><td>Lower bounds on output signals</td></tr><tr><td><code>ymax</code></td><td>Upper bounds on output signals</td></tr><tr><td><code>umin</code></td><td>Lower bounds on input signals</td></tr><tr><td><code>umax</code></td><td>Upper bounds on input signals</td></tr></table></td></tr>
<tr><td><code>weights</code></td><td>Updated QP optimization weights, returned as a structure with the following fields:

<table><tr><th>Field</th><th>Description</th></tr><tr><td><code>ywt</code></td><td>Output weights</td></tr><tr><td><code>uwt</code></td><td>Manipulated variable weights</td></tr><tr><td><code>duwt</code></td><td>Manipulated variable rate weights</td></tr><tr><td><code>ecr</code></td><td>Weight on slack variable used for constraint softening</td></tr></table></td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>model</td><td>Updated plant and nominal values for adaptive MPC and time-varying MPC, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>A, B, C, D</td><td>State-space matrices of discrete-time state-space model.</td></tr>
<tr><td>X</td><td>Nominal plant states</td></tr>
<tr><td>U</td><td>Nominal plant inputs</td></tr>
<tr><td>Y</td><td>Nominal plant outputs</td></tr>
<tr><td>DX</td><td>Nominal plant state derivatives</td></tr>
</table>
</td></tr>
</table>

`getCodeGenerationData` returns `onlineData` with empty matrices for all structure fields, except`signals.ref`, `signals.ym`, and `signals.md`. These fields contain the corresponding nominal signal values from `MPCobj`. If your controller does not have measured disturbances, `signals.md` is returned as an empty matrix.

For more information on configuring `onlineData` fields, see `mpcmoveCodeGeneration`

See Also

`mpcmoveCodeGeneration`

Topics

“Generate Code To Compute Optimal MPC Moves in MATLAB”
“Generate Code and Deploy Controller to Real-Time Targets”

Introduced in R2016a

getconstraint

Obtain mixed input/output constraints from model predictive controller

Syntax

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

Description

`[E,F,G,V,S] = getconstraint(MPCobj)` returns the mixed-input/output constraints previously defined for the MPC controller, `MPCobj`. The constraints are in the general form:

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

where $j = 0,\ldots,p$, and:

- p is the prediction horizon.
- k is the current time index.
- u is a column vector manipulated variables.
- y is a column vector of all plant output variables.
- v is a column vector of measured disturbance variables.
- ε is a scalar slack variable used for constraint softening (as in “Standard Cost Function”).
- E, F, G, V, and S are constant matrices.

Since the MPC controller does not optimize $u(k+p|k)$, `getconstraint` calculates the last constraint at time $k+p$ assuming that $u(k+p|k) = u(k+p-1|k)$.

Examples

Retrieve Custom Constraints from MPC Controller

Create a third-order plant model with two manipulated variables, one measured disturbance, and two measured outputs.

```
plant = rss(3,2,3);
plant.D = 0;
plant = setmpcsignals(plant,'mv',[1 2],'md',3);
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);
```

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

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

Retrieve the constraints from the controller.

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

```
E = 2×2

     1     1
     0     0


F = 2×2
```

```
     0     0
     0     1


G = 2×1

     5
    10


V = 2×1

     1
     1


S = 2×1

     0
     1
```

Input Arguments

`MPCobj` — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

`E` — Manipulated variable constraint constant

array

Manipulated variable constraint constant, returned as an N_c-by-N_{mv} array, where N_c is the number of constraints, and N_{mv} is the number of manipulated variables.

If `MPCobj` has no mixed input/output constraints, then `E` is `[]`.

F — Controlled output constraint constant
array

Controlled output constraint constant, returned as an N_c-by-N_y array, where N_y is the number of controlled outputs (measured and unmeasured).

If `MPCobj` has no mixed input/output constraints, then `F` is `[]`.

G — Mixed input/output constraint constant
column vector

Mixed input/output constraint constant, returned as a column vector of length N_c., where N_c is the number of constraints.

If `MPCobj` has no mixed input/output constraints, then `G` is `[]`.

V — Constraint softening constant
column vector

Constraint softening constant representing the equal concern for the relaxation (ECR), returned as a column vector with N_c elements, where N_c is the number of constraints. If `MPCobj` has no mixed input/output constraints, then `V` is `[]`.

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

To make the i^{th} constraint hard, specify $V(i) = 0$.

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

In general, as $V(i)$ decreases, the controller hardens the constraints by decreasing the constraint violation that is allowed.

S — Measured disturbance constraint constant
array

Measured disturbance constraint constant, returned as an N_c-by-N_v array, where N_v is the number of measured disturbances.

If there are no measured disturbances in the mixed input/output constraints, or `MPCobj` has no mixed input/output constraints, then `S` is `[]`.

See Also

`setconstraint`

Topics

“Constraints on Linear Combinations of Inputs and Outputs”

Introduced in R2011a

getEstimator

Obtain Kalman gains and model for estimator design

Syntax

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

Description

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

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

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

Examples

Extract Parameters for State Estimation

The plant is a stable, discrete LTI state-space 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.
-->Assuming output disturbance added to measured output channel #1 is integrated white
   Assuming no disturbance added to measured output channel #3.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

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

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

```
Poles = 6×1

   -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 = 6×2
10^-16 ×

         0   -0.4163
    0.1388         0
    0.1388   -0.1388
   -0.1388   -0.0694
   -0.2776         0
         0    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 2-55.

`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 2-55.

`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 2-55.

`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`.

Definitions

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.
$[j\|k]$	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.)

See Also

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

Topics

“Controller State Estimation”
“MPC Modeling”

Introduced in R2014b

getindist

Retrieve unmeasured input disturbance model

Syntax

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

Description

`indist = getindist(MPCobj)` returns the input disturbance model, `indist`, used by the model predictive controller, `MPCobj`.

`[indist,channels] = getindist(MPCobj)` also returns the input channels to which integrated white noise has been added by default. For more information on the default model, see “MPC Modeling”.

Examples

Retrieve Input Disturbance Model

Define a plant model with no direct feedthrough.

```
plant = rss(3,1,2);
plant.D = 0;
```

Set the first input signal as a manipulated variable and the second input as an unmeasured disturbance.

```
plant = setmpcsignals(plant,'MV',[1],'UD',[2]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

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

Extract the input disturbance model.

```
indist = getindist(MPCobj);
```

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

Retrieve Input Disturbance Model Channels with Default Integrated White Noise

Define a plant model with no direct feedthrough.

```
plant = rss(3,1,3);
plant.D = 0;
```

Set the first input signal as a manipulated variable and the other two inputs as unmeasured disturbances.

```
plant = setmpcsignals(plant,'MV',[1],'UD',[2 3]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->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
```

Extract the default output disturbance model.

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

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
```

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

```
Check which input disturbance channels have integrated white noise
added
```

```
by default.
```

```
channels
```

```
channels = 1
```

An integrator has been added only to the first unmeasured input disturbance. The other input disturbance uses a static unity gain to preserve state observability.

Input Arguments

`MPCobj` — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

`indist` — Input disturbance model

discrete-time, delay-free, state-space model

Input disturbance model used by the model predictive controller, `MPCobj`, returned as a discrete-time, delay-free, state-space model.

The input disturbance model has:

- Unit-variance white noise input signals. By default, the number of inputs depends upon the number of unmeasured input disturbances and the need to maintain controller state observability. For custom input disturbance models, the number of inputs is your choice.

- n_d outputs, where n_d is the number of unmeasured disturbance inputs defined in `MPCobj.Model.Plant`. Each disturbance model output is sent to the corresponding plant unmeasured disturbance input.

If `MPCobj` does not have any unmeasured disturbance, `indist` is returned as an empty state-space model.

This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

`channels` — Input channels with integrated white noise
vector of input indices

Input channels with integrated white noise added by default, returned as a vector of input indices. If you set `indist` to a custom input disturbance model using `setindist`, `channels` is empty.

Tips

- To specify a custom input disturbance model, use the `setindist` command.

See Also

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

Topics

“MPC Modeling”
“Controller State Estimation”

Introduced in R2006a

getname

Retrieve I/O signal names in MPC prediction model

Syntax

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

Description

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

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

See Also

`mpc` | `set` | `setname`

Introduced before R2006a

getoutdist

Retrieve unmeasured output disturbance model

Syntax

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

Description

`outdist = getoutdist(MPCobj)` returns the output disturbance model, `outdist`, used by the model predictive controller, `MPCobj`.

`[outdist,channels] = getoutdist(MPCobj)` also returns the output channels to which integrated white noise has been added by default. For more information on the default model, see “MPC Modeling”.

Examples

Retrieve Output Disturbance Model

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,2,2);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Extract the output disturbance model.

```
outdist = getoutdist(MPCobj);
```

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

Retrieve Output Disturbance Model Channels with Default Integrated White Noise

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.d = 0;
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->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.
```

Extract the default output disturbance model.

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

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->Assuming output disturbance added to measured output channel #3 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Check which channels have default integrated white noise disturbances.

```
channels
```

```
channels = 1×3

     1     2     3
```

Integrators have been added to all three output channels.

Input Arguments

`MPCobj` — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

`outdist` — Output disturbance model

discrete-time, delay-free, state-space model

Output disturbance model used by the model predictive controller, `MPCobj`, returned as a discrete-time, delay-free, state-space model.

The output disturbance model has:

- n_y outputs, where n_y is the number of plant outputs defined in `MPCobj.Model.Plant`. Each disturbance model output is added to the corresponding plant output. By default, disturbance models corresponding to unmeasured output channels are zero.
- Unit-variance white noise input signals. By default, the number of inputs is equal to the number of default integrators added.

This model, in combination with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

`channels` — Output channels with integrated white noise

vector of output indices

Output channels with integrated white noise added by default, returned as a vector of output indices. If you set `outdist` to a custom output disturbance model using `setoutdist`, `channels` is empty.

Tips

- To specify a custom output disturbance model, use the `setoutdist` command.

See Also

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

Topics

“MPC Modeling”
“Controller State Estimation”

Introduced before R2006a

gpc2mpc

Generate MPC controller using generalized predictive controller (GPC) settings

Syntax

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

Description

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

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

`mpcObj = 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.

`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);
```

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.

References

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

See Also

“MPC Controller Object” on page 4-2

Topics

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

Introduced in R2010a

mpc

Create MPC controller

Syntax

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

Description

`MPCobj = mpc(Plant)` creates a model predictive controller object based on a discrete-time prediction model. The prediction model `Plant` can be either an LTI model with a specified sample time or a linear System Identification Toolbox model. 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 linear System Identification Toolbox model. If the `Ts` input argument is unspecified, `Plant` must be either 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.

If you specify `Plant` as a linear System Identification Toolbox model, any noise channels are discarded by default. To convert noise channels to unmeasured disturbances, first convert the identified model to a state-space model using the `'augmented'` option. For more information on identifying plant models, see “Identify Plant from Data”.

`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, $1 \le m \le p$, or as a vector of blocking factors such that $\mathtt{sum(m)} \le p$ (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 2-77.

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 2-74.

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 2-76.

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 2-76.

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 2-79.

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.
`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 vector)	`InputName` of LTI plant model
`Units`	Read-only MV signal units (character vector)	`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 vector)	`OutputName` of LTI plant model
`Units`	Read-only OV signal units (character vector)	`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 vector)	`InputName` of LTI plant model
`Units`	Read-only DV signal units (character vector)	`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”).

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 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. 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".

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` = $R_{\Delta \mathrm{u}}$ are positive semidefinite matrices.

Note You cannot specify nondiagonal 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.)

<table>
<tr><th>Field Name</th><th>Content</th><th>Default</th></tr>
<tr><td>Nominal</td><td>Structure containing the state, input, and output values where Model.Plant 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>X</td><td>Plant state at operating point</td><td>[]</td></tr>
<tr><td>U</td><td>Plant input at operating point, including manipulated variables and measured and unmeasured disturbances</td><td>[]</td></tr>
<tr><td>Y</td><td>Plant output at operating point</td><td>[]</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>[]</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	`[]`
`U`	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances	`[]`
`Y`	Plant output at operating point	`[]`
`DX`	For continuous-time models, `DX` is the state derivative at operating point: `DX`=f(`X`,`U`). For discrete-time models, `DX`=$x(k+1)$-$x(k)$=f(`X`,`U`)-`X`.	`[]`

Ts

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

Optimizer

`Optimizer` is a structure with fields that contain parameters for the QP optimization.

Optimizer Properties

Field	Description	Default
`MaxIter`	Maximum number of iterations allowed in the QP solver, specified as one of the following: • `'Default'` — The MPC controller automatically computes the maximum number of QP solver iterations as: $\mathrm{MaxIter} = 4\left(n_c + n_v\right)$ where • n_{cy} is the total number of constraints across the prediction horizon. • n_v is the total number of optimization variables across the control horizon. The default `MaxIter` value has a lower bound of `120`. • Positive integer — The QP solver stops after `MaxIter` iterations. If the solver fails to converge in the final iteration, the controller: • Freezes the controller movement if `UseSuboptimalSolution` is `false`. • Applies the suboptimal solution reached after the final iteration if `UseSuboptimalSolution` is `true`. If `CustomSolver` or `CustomSolverCodeGen` is `true`, the controller does not require the custom solver to honor `MaxIter`.	`'Default'`

Field	Description	Default
`MinOutputECR`	Minimum value allowed for `OutputMinECR` and `OutputMaxECR`, specified as a nonnegative scalar. A value of 0 indicates that hard output constraints are allowed. If either of the `OutputVariables.MinECR` or `OutputVariables.MaxECR` properties of an MPC controller are less than `MinOutputECR`, a warning is displayed and the value is raised to `MinOutputECR` during computation.	`0`
`UseSuboptimalSolution`	Flag indicating whether to apply a suboptimal solution after the number of optimization iterations exceeds `MaxIter`, specified as a logical value.	`false`
`UseWarmStart`	Flag indicating whether to *warm start* each QP solver iteration by passing in a list of active inequalities from the previous iteration, specified as a logical value. Inequalities are active when their equal portion is true. If `CustomSolver` or `CustomSolverCodeGen` is `true`, the controller does not require the custom solver to honor `UseWarmStart`.	`true`

Field	Description	Default
`CustomSolver`	Flag indicating whether to use a custom QP solver for simulation, specified as a logical value. If `CustomSolver` is `true`, the user must provide an `mpcCustomSolver` function on the MATLAB path. This custom solver is not used for code generation. To generate code for a controller with a custom solver, use `CustomSolverCodeGen`. If `CustomSolver` is `true`, the controller does not require the custom solver to honor `MaxIter` and `UseWarmStart`. For more information on specifying custom solvers, see “Custom QP Solver”.	`false`
`CustomSolverCodeGen`	Flag indicating whether to use a custom QP solver for code generation, specified as a logical value. If `CustomSolverCodeGen` is `true`, the user must provide an `mpcCustomSolverCodeGen` function on the MATLAB path. This custom solver is not used for simulation. To simulate a controller with a custom solver, use `CustomSolver`. If `CustomSolverCodeGen` is `true`, the controller does not require the custom solver to honor `MaxIter` and `UseWarmStart`. For more information on specifying custom solvers, see “Custom QP Solver”.	`false`

Note The default `MaxIter` value can be very large for some controller configurations, such as those with large prediction and control horizons. When simulating such

controllers, if the QP solver cannot find a feasible solution, the simulation can appear to stop responding, since the solver continues searching for `MaxIter` iterations.

PredictionHorizon

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

ControlHorizon

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

History

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

Notes

`Notes` stores text or comments as a cell array of character vectors.

UserData

Any additional data stored within the MPC controller object.

Examples

Create MPC Controller with Specified Prediction and Control Horizons

Create a plant model with the transfer function $(s+1)/(s^2+2s)$.

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

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

Specify a sample time for the controller.

```
Ts = 0.1;
```

Define bounds on the manipulated variable, u, such that $-1 \le u \le 1$.

```
MV = struct('Min',-1,'Max',1);
```

`MV` contains only the upper and lower bounds on the manipulated variable. In general, you can specify additional MV properties. When you do not specify other properties, their default values apply.

Specify a 20-interval prediction horizon and a 3-interval control horizon.

```
p = 20;
m = 3;
```

Create an MPC controller using the specified values. The fifth input argument is empty, so default tuning weights apply.

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

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

Compatibility Considerations

Support for implementing economic MPC using a linear MPC controller has been removed

Errors starting in R2018b

Support for implementing economic MPC using a linear MPC controller has been removed. Implement economic MPC using the new nonlinear MPC controller instead. For more information on nonlinear MPC controllers, see “Nonlinear MPC”.

If you previously saved a linear MPC object configured with custom cost or constraint functions, the software generates a warning when the object is loaded and an error if it is simulated. To suppress the error and warning messages and continue using your linear MPC controller, `mpcobj`, without the custom costs and constraints, set the `IsEconomicMPC` flag to `false`.

```
mpcobj.IsEconomicMPC = false;
```

To implement your economic MPC controller using a nonlinear MPC object:

1. Create an `nlmpc` object.
2. Convert your custom cost function to the format required for nonlinear MPC. For more information on nonlinear MPC cost functions, see “Specify Cost Function for Nonlinear MPC”.
3. Convert your custom constraint function to the format required for nonlinear MPC. For more information on nonlinear MPC constraints, see “Specify Constraints for Nonlinear MPC”.
4. Implement your linear prediction model using state and output functions. For more information on nonlinear MPC prediction models, see “Specify Prediction Model for Nonlinear MPC”.

See Also

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

Topics

“MPC Modeling”
“Design MPC Controller at the Command Line”

Introduced before R2006a

mpcmove

Compute optimal control action

Syntax

```
mv = mpcmove(MPCobj,x,ym,r,v)
[mv,info] = mpcmove(MPCobj,x,ym,r,v)
[___] = mpcmove(___,options)
```

Description

`mv = 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.

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

`[___] = mpcmove(___,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 of constraints and weights during the closed-loop simulation.

Input Arguments

`MPCobj` — Model predictive controller
MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`x` — Current controller state
`mpcstate` object

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

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` — Current measured output values
column vector of length N_{ym}

Current measured output values at time k, specified as a column vector of length N_{ym}, where N_{ym} is the number of measured outputs.

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

`r` — Plant output reference values
p-by-N_y array

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

`r` must contain at least one row. If `r` contains fewer than p rows, `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
(p+1)-by-N_{md} array

Current and anticipated measured disturbances, specified as a (p+1)-by-N_{md} array, where p is the prediction horizon of `MPCobj` and N_{md} is the number of measured disturbances.

The first row of `v` specifies the current measured disturbance values. Row `v(i+1,:)` defines the anticipated 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 = []`.

If your model includes measured disturbances, `v` must contain at least one row. If `v` contains fewer than $p+1$ rows, `mpcmove` duplicates the last row to fill the $(p+1)$-by-N_{md} array. If you supply exactly one row, 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` — Run-time options
`mpcmoveopt` object

Run-time options, specified as an `mpcmoveopt` object. Use `options` to override selected properties of `MPCobj` during simulation. 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

`mv` — Optimal manipulated variable moves
column vector

Optimal manipulated variable moves, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution, `x.LastMove`.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

`info` — Solution details
structure

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

`Uopt` — Optimal manipulated variable sequence
(p+1)-by-N_{mv} array

Predicted optimal manipulated variable adjustments (moves), returned as a (p+1)-by-N_{mv} array, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

`Uopt(i,:)` contains the calculated optimal values at time `k+i-1`, for `i = 1,...,p`, where `k` is the current time. The first row of `Info.Uopt` contains the same manipulated variable values as output argument `mv`. Since the controller does not calculate optimal control moves at time `k+p`, `Uopt(p+1,:)` is equal to `Uopt(p,:)`.

`Yopt` — Optimal output variable sequence
(p+1)-by-N_y array

Optimal output variable sequence, returned as a (p+1)-by-N_y array, where p is the prediction horizon and N_y is the number of outputs.

The first row of `Info.Yopt` contains the calculated outputs at time `k` based on the estimated states and measured disturbances; it is not the measured output at time `k`. `Yopt(i,:)` contains the predicted output values at time `k+i-1`, for `i = 1,...,p+1`.

`Yopt(i,:)` contains the calculated output values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Yopt(1,:)` is computed based on the estimated states and measured disturbances.

`Xopt` — Optimal prediction model state sequence
(p+1)-by-N_x array

Optimal prediction model state sequence, returned as a (p+1)-by-N_x array, where p is the prediction horizon and N_x is the number of states in the plant and unmeasured disturbance models (states from noise models are not included).

`Xopt(i,:)` contains the calculated state values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Xopt(1,:)` is the same as the current states state values.

`Topt` — Time intervals
column vector of length $p+1$

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

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

`Slack` — Slack variable
nonnegative 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` — Number of solver iterations
positive integer | `0` | `-1` | `-2`

Number of solver iterations, returned as one of the following:

- Positive integer — Number of iterations needed to solve the optimization problem that determines the optimal sequences.
- `0` — Optimization problem could not be solved in the specified maximum number of iterations.
- `–1` — Optimization problem was infeasible. An optimization problem is infeasible if no solution can satisfy all the hard constraints.
- `–2` — Numerical error occurred when solving the optimization problem.

`QPCode` — Optimization solution status
`'feasible'` | `'infeasible'` | `'unrealiable'`

Optimization solution status, returned as one of the following:

- `'feasible'` — Optimal solution was obtained (`Iterations` > 0)
- `'infeasible'` — Solver detected a problem with no feasible solution (`Iterations` = –1) or a numerical error occurred (`Iterations` = –2)
- `'unreliable'` — Solver failed to converge (`Iterations` = 0). In this case, if `MPCobj.Optimizer.UseSuboptimalSolution` is `false`, `u` freezes at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.

`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. For more information, see "Optimization Problem".

The cost value is only meaningful when `QPCode = 'feasible'`, or when `QPCode = 'feasible'` and `MPCobj.Optimizer.UseSuboptimalSolution` is `true`.

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);
```

```
-->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
```

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

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

```
x = mpcstate(MPCobj);

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

Set the reference signal. There is no measured disturbance.

```
r = 1;
```

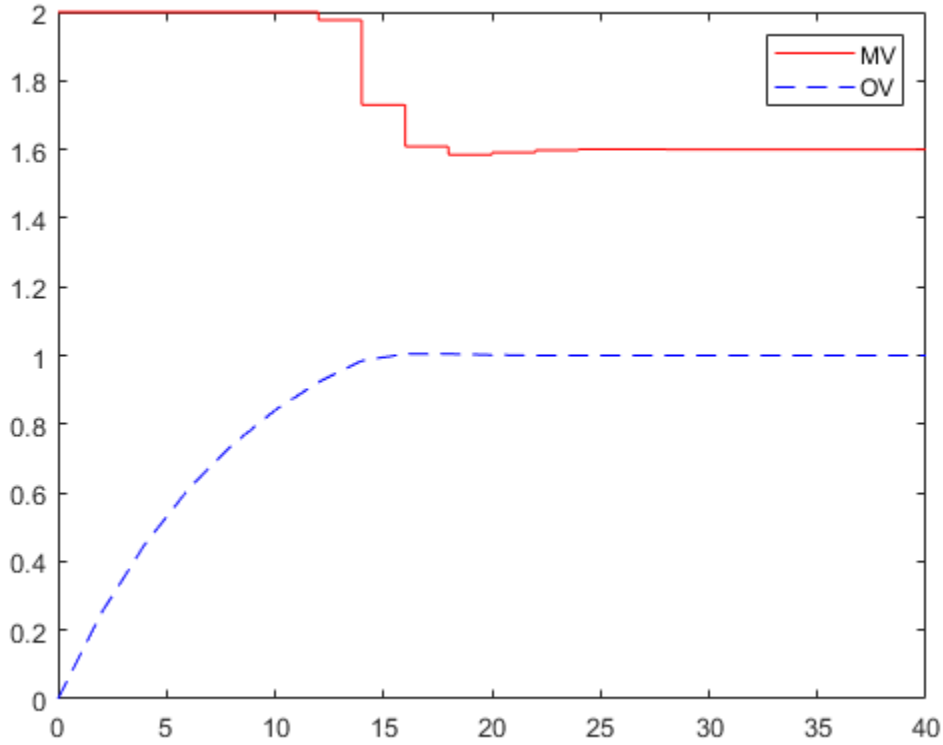
Simulate the closed-loop response by calling `mpcmove` iteratively.

```
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;
```

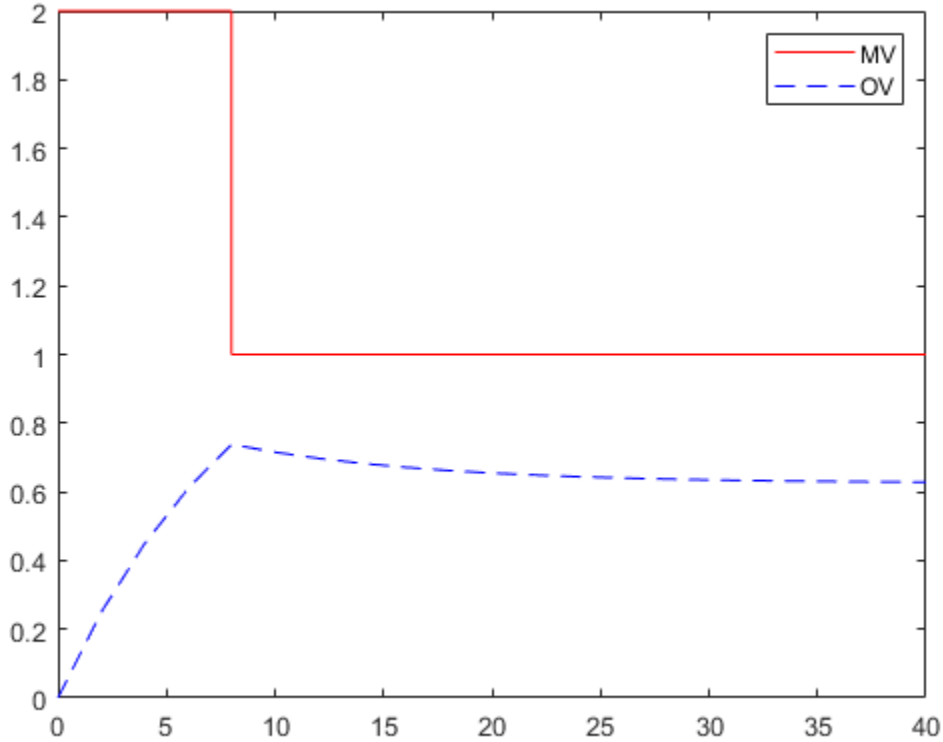
Simulate the closed-loop response and introduce the real-time upper limit change at eight seconds (the fifth iteration step).

```
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 constraints on both the manipulated variable and the rate of change of the manipulated variable. The prediction horizon is `10` intervals, and the control horizon is blocked.

```
MPCobj = mpc(Plant,ts,10,[2 3 5]);
```

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

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

Initialize an `mpcstate` object for simulation from a particular state.

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

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

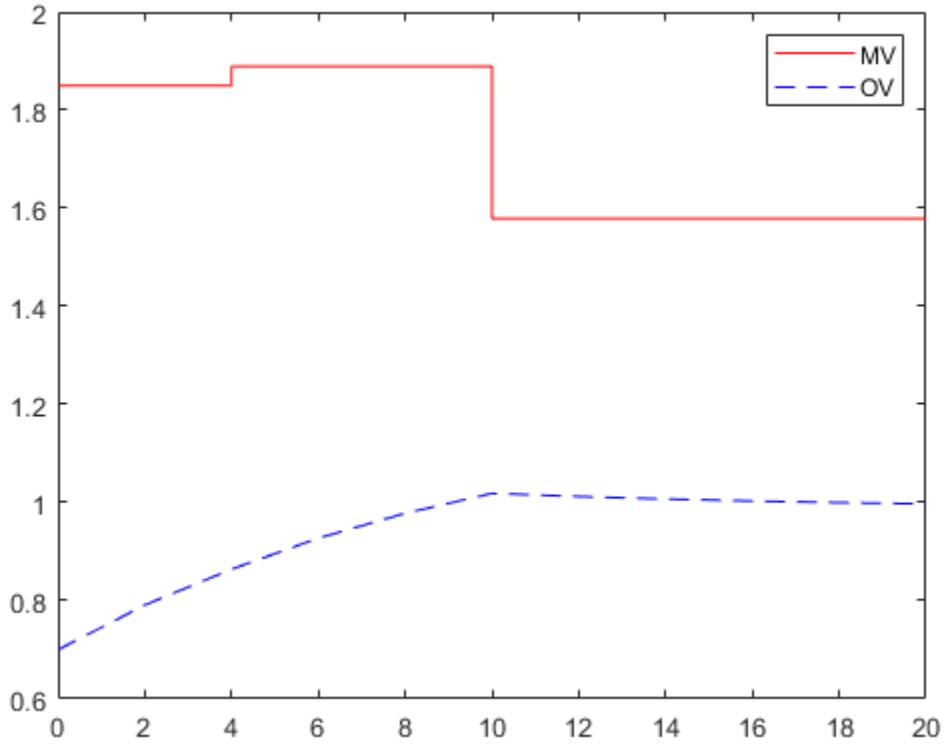
```
x.Plant = 2.8;
x.LastMove = 0.85;
```

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
```

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`.

Alternatives

- Use `sim` for plant mismatch and noise simulation when not using run-time constraints or weight changes.
- Use the **MPC Designer** app to interactively design and simulate model predictive controllers.
- Use the MPC Controller block in Simulink and for code generation.
- Use `mpcmoveCodeGeneration` for code generation.

See Also

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

Topics

“Improving Control Performance with Look-Ahead (Previewing)”
“Switching Controllers Based on Optimal Costs”
“Understanding Control Behavior by Examining Optimal Control Sequence”

Introduced before R2006a

mpcmoveAdaptive

Compute optimal control with prediction model updating

Syntax

```
mv = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v)
[mv,info] = mpcmoveAdaptive(MPCobj,x,Plant,Nominal,ym,r,v)
[___] = mpcmoveAdaptive(___,options)
```

Description

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

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

`[___] = mpcmoveAdaptive(___,options)` alters selected controller settings using options you specify with `mpcmoveopt`. These changes apply for the current time instant only, enabling a command-line simulation using `mpcmoveAdaptive` to mimic the Adaptive MPC Controller block in Simulink in a computationally efficient manner.

Input Arguments

`MPCobj` — MPC controller
MPC controller object

MPC controller, specified as an implicit MPC controller object. To create the MPC controller, use the `mpc` command.

`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.

For more information on state estimation for adaptive MPC and time-varying MPC, see "State Estimation".

`Plant` — Updated prediction model
discrete-time state-space model | model array

Updated prediction model, specified as one of the following:

- A delay-free, discrete-time state-space (`ss`) model. This plant is the update to `MPCobj.Model.Plant` and it must:
  - Have the same sample time as the controller; that is, `Plant.Ts` must match `MPCobj.Ts`
  - Have the same input and output signal configurations, such as type, order, and dimensions
  - Define the same states as the controller prediction model, `MPCobj.Model.Plant`
- An array of up to $p+1$ delay-free, discrete-time state-space models, where p is the prediction horizon of `MPCobj`. Use this option to vary the controller prediction model over the prediction horizon.

If `Plant` contains fewer than $p+1$ models, the last model repeats for the rest of the prediction horizon.

Tip If you use a plant other than a delay-free, discrete-time state-space model to define the prediction model in `MPCobj`, you can convert it to such a model to determine the prediction model structure.

If the original plant is	Then
Not a state-space model	Convert it to a state-space model using `ss`.
A continuous-time model	Convert it to a discrete-time model with the same sample time as the controller, `MPCobj.Ts`, using `c2d` with default forward Euler discretization.
A model with delays	Convert the delays to states using `absorbDelay`.

`Nominal` — Updated nominal conditions

structure | structure array | `[]`

Updated nominal conditions, specified as one of the following:

- A structure of with the following fields:

Field	Description	Default
`X`	Plant state at operating point	`[]`
`U`	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances	`[]`
`Y`	Plant output at operating point	`[]`

Field	Description	Default
DX	For continuous-time models, DX is the state derivative at operating point: DX=*f*(X,U). For discrete-time models, DX=*x*(*k*+1)-*x*(*k*)=*f*(X,U)-X.	[]

- An array of up to *p*+1 nominal condition structures, where *p* is the prediction horizon of `MPCobj`. Use this option to vary controller nominal conditions over the prediction horizon.

  If `Nominal` contains fewer than *p*+1 structures, the last structure repeats for the rest of the prediction horizon.

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

`ym` — Current measured outputs

row vector of length N_{ym}

Current measured outputs, specified as a row vector of length N_{ym} vector, where N_{ym} is the number of measured outputs.

If you are using custom state estimation, `ym` is ignored. If you set `ym = []`, then `mpcmoveAdaptive` uses the appropriate nominal value.

`r` — Plant output reference values

p-by-N_y array | [ ]

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

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

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

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

`v` — Current and anticipated measured disturbances

p-by-N_{md} array | [ ]

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

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

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

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

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

`options` — 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 `options` yields the same result as redefining or modifying `MPCobj` before each call to `mpcmoveAdaptive`, but involves considerably less overhead. Using `options` 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

`mv` — Optimal manipulated variable moves
column vector

Optimal manipulated variable moves, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution, `x.LastMove`.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

`info` — Solution details
structure

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

`Uopt` — Optimal manipulated variable sequence
(p+1)-by-N_{mv} array

Predicted optimal manipulated variable adjustments (moves), returned as a (p+1)-by-N_{mv} array, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

`Uopt(i,:)` contains the calculated optimal values at time `k+i-1`, for `i = 1,...,p`, where `k` is the current time. The first row of `Info.Uopt` contains the same manipulated variable values as output argument `mv`. Since the controller does not calculate optimal control moves at time `k+p`, `Uopt(p+1,:)` is equal to `Uopt(p,:)`.

`Yopt` — Optimal output variable sequence
(p+1)-by-N_y array

Optimal output variable sequence, returned as a (p+1)-by-N_y array, where p is the prediction horizon and N_y is the number of outputs.

The first row of `Info.Yopt` contains the calculated outputs at time `k` based on the estimated states and measured disturbances; it is not the measured output at time `k`. `Yopt(i,:)` contains the predicted output values at time `k+i-1`, for `i = 1,...,p+1`.

`Yopt(i,:)` contains the calculated output values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Yopt(1,:)` is computed based on the estimated states and measured disturbances.

`Xopt` — Optimal prediction model state sequence
(p+1)-by-N_x array

Optimal prediction model state sequence, returned as a (p+1)-by-N_x array, where p is the prediction horizon and N_x is the number of states in the plant and unmeasured disturbance models (states from noise models are not included).

`Xopt(i,:)` contains the calculated state values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Xopt(1,:)` is the same as the current states state values.

`Topt` — Time intervals
column vector of length p+1

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

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

`Slack` — Slack variable
nonnegative 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` — Number of solver iterations
positive integer | `0` | `-1` | `-2`

Number of solver iterations, returned as one of the following:

- Positive integer — Number of iterations needed to solve the optimization problem that determines the optimal sequences.
- `0` — Optimization problem could not be solved in the specified maximum number of iterations.
- `–1` — Optimization problem was infeasible. An optimization problem is infeasible if no solution can satisfy all the hard constraints.
- `–2` — Numerical error occurred when solving the optimization problem.

`QPCode` — Optimization solution status
`'feasible'` | `'infeasible'` | `'unrealiable'`

Optimization solution status, returned as one of the following:

- `'feasible'` — Optimal solution was obtained (`Iterations` > 0)
- `'infeasible'` — Solver detected a problem with no feasible solution (`Iterations` = –1) or a numerical error occurred (`Iterations` = –2)
- `'unreliable'` — Solver failed to converge (`Iterations` = 0). In this case, if `MPCobj.Optimizer.UseSuboptimalSolution` is `false`, `u` freezes at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.

`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. For more information, see "Optimization Problem".

The cost value is only meaningful when `QPCode = 'feasible'`, or when `QPCode = 'feasible'` and `MPCobj.Optimizer.UseSuboptimalSolution` is `true`.

Tips

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

See Also

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

Topics

"Adaptive MPC"
"Time-Varying MPC"
"Optimization Problem"

Introduced in R2014b

mpcmoveCodeGeneration

Compute optimal control moves with code generation support

Syntax

```
[mv,newStateData] = mpcmoveCodeGeneration(configData,stateData,
onlineData)
[___,info] = mpcmoveCodeGeneration(___)
```

Description

`[mv,newStateData] = mpcmoveCodeGeneration(configData,stateData, onlineData)` computes optimal MPC control moves and supports code generation for deployment to real-time targets. The input data structures, generated using `getCodeGenerationData`, define the MPC controller to simulate.

`mpcmoveCodeGeneration` does not check input arguments for correct dimensions and data types.

`[___,info] = mpcmoveCodeGeneration(___)` returns additional information about the optimization result, including the number of iterations and the objective function cost.

Examples

Compute Optimal Control Moves Using Code Generation Data Structures

Create a proper plant model.

```
plant = rss(3,1,1);
plant.D = 0;
```

Specify the controller sample time.

```
Ts = 0.1;
```

Create an MPC controller.

```
mpcObj = mpc(plant,Ts);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming 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 code generation data structures.

```
[configData,stateData,onlineData] = getCodeGenerationData(mpcObj);
```

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

Initialize the plant states to zero to match the default states used by the MPC controller.

Run a closed-loop simulation. At each control interval, update the online data structure and call `mpcmoveCodeGeneration` to compute the optimal control moves.

```
x = zeros(size(plant.B,1),1); % Initialize plant states to zero (|mpcObj| default).
Tsim = 20;
for i = 1:round(Tsim/Ts)+1
    % Update plant output.
    y = plant.C*x;
    % Update measured output in online data.
    onlineData.signals.ym = y;
    % Update reference signal in online data.
    onlineData.signals.ref = 1;
    % Compute control actions.
    [u,statedata] = mpcmoveCodeGeneration(configData,stateData,onlineData);
    % Update plant state.
    x = plant.A*x + plant.B*u;
end
```

Generate MEX function with MATLAB® Coder™, specifying `configData` as a constant.

```
func = 'mpcmoveCodeGeneration';
funcOutput = 'mpcmoveMEX';
```

```
Cfg = coder.config('mex');
Cfg.DynamicMemoryAllocation = 'off';
codegen('-config',Cfg,func,'-o',funcOutput,'-args',...
    {coder.Constant(configData),stateData,onlineData});
```

Input Arguments

`configData` — MPC configuration parameters

structure

MPC configuration parameters that are constant at run time, specified as a structure generated using `getCodeGenerationData`.

Note When using `codegen`, `configData` must be defined as `coder.Constant`.

`stateData` — Controller state

structure

Controller state at run time, specified as a structure. Generate the initial state structure using `getCodeGenerationData`. For subsequent control intervals, use the updated controller state from the previous interval. In general, use the `newStateData` structure directly.

If custom state estimation is enabled, you must manually update the state structure during each control interval. For more information, see “Using Custom State Estimation”.

`onlineData` — Online controller data

structure

Online controller data that you must update at run time, specified as a structure with the following fields:

`signals` — Updated input and output signals

structure

Updated input and output signals, specified as a structure with the following fields:

`ym` — Measured outputs

vector of length N_{ym}

Measured outputs, specified as a vector of length N_{ym}, where N_{ym} is the number of measured outputs.

By default,`getCodeGenerationData` sets `ym` to the nominal measured output values from the controller.

`ref` — Output references
row vector of length N_y | p-by-n_y array

Output references, specified as a row vector of length N_y, where N_y is the number of outputs.

If you are using reference signal previewing with implicit or adaptive MPC, specify a p-by-N_y array, where p is the prediction horizon.

By default,`getCodeGenerationData` sets `ref` to the nominal output values from the controller.

`md` — Measured disturbances
row vector of length N_{md} | p-by-N_{md} array

Measured disturbances, specified as:

- A row vector of length N_{md}, where N_{md} is the number of measured disturbances.
- Ap-by-N_{md} array, if you are using signal previewing with implicit or adaptive MPC.

By default, if your controller has measured disturbances,`getCodeGenerationData` sets `md` to the nominal measured disturbance values from the controller. Otherwise, this field is empty and ignored by `mpcmoveCodeGeneration`.

`mvTarget` — Targets for manipulated variables
`[]` (default) | vector of length N_{mv}

Targets for manipulated variables, specified as:

- A vector of length N_{mv}, where N_{mv} is the number of manipulated variables.
- `[]` to use the default targets defined in the original MPC controller.

This field is ignored when using an explicit MPC controller.

`externalMV` — Manipulated variables externally applied to the plant
`[]` (default) | vector of length N_{mv}

Manipulated variables externally applied to the plant, specified as:

- A vector of length N_{mv}.
- `[]` to apply the optimal control moves to the plant.

`limits` — Updated input and output constraints
structure

Updated input and output constraints, specified as a structure. If you do not expect constraints to change at run time, ignore `limits`. This structure contains the following fields:

`ymin` — Lower bounds on output signals
column vector of length N_y | `[]`

Lower bounds on output signals, specified as a column vector of length N_y.

If `ymin` is empty, `[]`, the default bounds defined in the original MPC controller are used.

`ymax` — Upper bounds on output signals
column vector of length N_y | `[]`

Upper bounds on output signals, specified as a column vector of length N_y.

If `ymax` is empty, `[]`, the default bounds defined in the original MPC controller are used.

`umin` — Lower bounds on manipulated variables
column vector of length N_{mv} | `[]`

Lower bounds on manipulated variables, specified as a column vector of length N_{mv}.

If `umin` is empty, `[]`, the default bounds defined in the original MPC controller are used.

`umax` — Upper bounds on manipulated variables
column vector of length N_{mv} | `[]`

Upper bounds on manipulated variables, specified as a column vector of length N_{mv}.

If `umax` is empty, `[]`, the default bounds defined in the original MPC controller are used.

`weights` — Updated QP optimization weights
structure

Updated QP optimization weights, specified as a structure. If you do not expect tuning weights to change at run time, ignore `weights`. This structure contains the following fields:

`ywt` — Output weights
column vector of length N_y | `[]`

Output weights, specified as a column vector of length N_y that contains nonnegative values.

If `ywt` is empty, `[]`, the default weights defined in the original MPC controller are used.

`uwt` — Manipulated variable weights
column vector of length N_{mv} | `[]`

Manipulated variable weights, specified as a column vector of length N_{mv} that contains nonnegative values.

If `uwt` is empty, `[]`, the default weights defined in the original MPC controller are used.

`duwt` — Manipulated variable rate weights
column vector of length N_{mv} | `[]`

Manipulated variable rate weights, specified as a column vector of length N_{mv} that contains nonnegative values.

If `duwt` is empty, `[]`, the default weights defined in the original MPC controller are used.

`ecr` — Weight on slack variable used for constraint softening
nonnegative scalar | `[]`

Weight on slack variable used for constraint softening, specified as a nonnegative scalar.

If `uwt` is empty, `[]`, the default weight defined in the original MPC controller are used.

`model` — Updated plant and nominal values
structure

Updated plant and nominal values for adaptive MPC and time-varying MPC, specified as a structure. `model` is only available if you specify `isAdaptive` or `isLTV` as `true` when creating code generation data structures. This structure contains the following fields:

`A` — State matrix of discrete-time state-space plant model
N_x-by-N_x array | N_x-by-N_x-by-(p+1) array

State matrix of discrete-time state-space plant model, specified as an:

- N_x-by-N_x array when using adaptive MPC,
- N_x-by-N_x-by-(p+1) array when using time-varying MPC,

where N_x is the number of plant states.

`B` — Input-to-state matrix of discrete-time state-space plant model
N_x-by-N_u array | N_x-by-N_u-by-(p+1) array

Input-to-state matrix of discrete-time state-space plant model, specified as an:

- N_x-by-N_u array when using adaptive MPC,
- N_x-by-N_u-by-(p+1) array when using time-varying MPC,

where N_u is the number of plant inputs.

`C` — State-to-output matrix of discrete-time state-space plant model
N_y-by-N_x array | N_y-by-N_x-by-(p+1) array

State-to-output matrix of discrete-time state-space plant model, specified as an:

- N_y-by-N_x array when using adaptive MPC.
- N_y-by-N_x-by-(p+1) array when using time-varying MPC.

`D` — Feedthrough matrix of discrete-time state-space plant model
N_y-by-N_u array | N_y-by-N_u-by-(p+1) array

Feedthrough matrix of discrete-time state-space plant model, specified as an:

- N_y-by-N_u array when using adaptive MPC.
- N_y-by-N_u-by-(p+1) array when using time-varying MPC.

Since MPC controllers do not support plants with direct feedthrough, specify `D` as an array of zeros.

`X` — Nominal plant states
column vector of length N_x | N_x-by-1-by-(p+1) array

Nominal plant states, specified as:

- A column vector of length N_x when using adaptive MPC.
- An N_x-by-1-by-(p+1) array when using time-varying MPC.

`U` — Nominal plant inputs
column vector of length N_u | N_u-by-1-by-(p+1) array

Nominal plant inputs, specified as:

- A column vector of length N_u when using adaptive MPC.
- An N_u-by-1-by-(p+1) array when using time-varying MPC.

`Y` — Nominal plant outputs
column vector of length N_y | N_y-by-1-by-(p+1) array

Nominal plant outputs, specified as:

- A column vector of length $N_y$when using adaptive MPC.
- An N_y-by-1-by-(p+1) array when using time-varying MPC.

`DX` — Nominal plant state derivatives
column vector of length N_x | N_x-by-1-by-(p+1) array

Nominal plant state derivatives, specified as:

- A column vector of length N_x when using adaptive MPC.
- An N_x-by-1-by-(p+1) array when using time-varying MPC.

Output Arguments

`mv` — Optimal manipulated variable moves
column vector

Optimal manipulated variable moves, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution, `x.LastMove`.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

`newStateData` — Updated controller state
structure

Updated controller state, returned as a structure. For subsequent control intervals, pass `newStateData` to `mpcmoveCodeGeneration` as `stateData`.

If custom state estimation is enabled, use `newStateData` to manually update the state structure before the next control interval. For more information, see “Using Custom State Estimation”.

`info` — Controller optimization information
structure

Controller optimization information, returned as a structure.

If you are using implicit or adaptive MPC, `info` contains the following fields:

Field	Description
`Iterations`	Number of QP solver iterations
`QPCode`	QP solver status code
`Cost`	Objective function cost
`Uopt`	Optimal manipulated variable adjustments
`Yopt`	Optimal predicted output variable sequence
`Xopt`	Optimal predicted state variable sequence
`Topt`	Time horizon intervals
`Slack`	Slack variable used in constraint softening

If `configData.OnlyComputeCost` is `true`, the optimal sequence information, `Uopt`, `Yopt`, `Xopt`, `Topt`, and `Slack`, is not available:

For more information, see `mpcmove` and `mpcmoveAdaptive`.

If you are using explicit MPC, `info` contains the following fields:

Field	Description
`Region`	Region in which the optimal solution was found
`ExitCode`	Solution status code

For more information, see `mpcmoveExplicit`.

See Also

`codegen` | `getCodeGenerationData` | `mpcmove` | `mpcmoveAdaptive` | `mpcmoveExplicit`

Topics

“Generate Code To Compute Optimal MPC Moves in MATLAB”
“Generate Code and Deploy Controller to Real-Time Targets”

Introduced in R2016a

mpcmoveExplicit

Compute optimal control using explicit MPC

Syntax

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

Description

`mv = 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.

`[mv,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`.

`[mv,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.

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 row vector of length N_{ym}, where 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 length N_y. `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 length N_{md}, where N_{md} is the number of measured disturbances. 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 length N_{mv}, where N_{mv} is the number of manipulated variables. 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

`mv` — Optimal manipulated variable moves

column vector

Optimal manipulated variable moves, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution, `x.LastMove`.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

`info` — Explicit MPC solution status

structure

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

`ExitCode` — Solution status code

1 | 0 | –1

Solution status code, returned as one of the following values:

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

`Region` — Region to which current controller input parameters belong
positive integer | 0

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

Tips

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

See Also

`generateExplicitMPC`

Topics

“Explicit MPC Control of a Single-Input-Single-Output Plant”
“Explicit MPC”
“Design Workflow for Explicit MPC”

Introduced in R2014b

mpcmoveMultiple

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

Syntax

```
mv = mpcmoveMultiple(MPCArray,states,index,ym,r,v)
[mv,info] = mpcmoveMultiple(MPCArray,states,index,ym,r,v)
[___] = mpcmoveMultiple(___,options)
```

Description

`mv = 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.

`[mv,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`.

`[___] = mpcmoveMultiple(___,options)` 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, `mpcmoveMultiple` 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
row vector

Current measured outputs, specified as a row vector of length N_{ym}, where 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.

`options` — 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 `options` yields the same result as redefining or modifying the selected controller before each call to `mpcmoveMultiple`, but involves considerably less overhead. Using `options` 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

`mv` — Optimal manipulated variable moves

column vector

Optimal manipulated variable moves, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution, `x.LastMove`.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

`info` — Solution details

structure

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

`Uopt` — Optimal manipulated variable sequence

(p+1)-by-N_{mv} array

Predicted optimal manipulated variable adjustments (moves), returned as a (p+1)-by-N_{mv} array, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

`Uopt(i,:)` contains the calculated optimal values at time `k+i-1`, for `i = 1,...,p`, where `k` is the current time. The first row of `Info.Uopt` contains the same manipulated variable values as output argument `mv`. Since the controller does not calculate optimal control moves at time `k+p`, `Uopt(p+1,:)` is equal to `Uopt(p,:)`.

`Yopt` — Optimal output variable sequence
(p+1)-by-N_y array

Optimal output variable sequence, returned as a (p+1)-by-N_y array, where p is the prediction horizon and N_y is the number of outputs.

The first row of `Info.Yopt` contains the calculated outputs at time `k` based on the estimated states and measured disturbances; it is not the measured output at time `k`. `Yopt(i,:)` contains the predicted output values at time `k+i-1`, for `i = 1,...,p+1`.

`Yopt(i,:)` contains the calculated output values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Yopt(1,:)` is computed based on the estimated states and measured disturbances.

`Xopt` — Optimal prediction model state sequence
(p+1)-by-N_x array

Optimal prediction model state sequence, returned as a (p+1)-by-N_x array, where p is the prediction horizon and N_x is the number of states in the plant and unmeasured disturbance models (states from noise models are not included).

`Xopt(i,:)` contains the calculated state values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Xopt(1,:)` is the same as the current states state values.

`Topt` — Time intervals
column vector of length p+1

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

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

`Slack` — Slack variable
nonnegative 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` — Number of solver iterations
positive integer | `0` | `-1` | `-2`

Number of solver iterations, returned as one of the following:

- Positive integer — Number of iterations needed to solve the optimization problem that determines the optimal sequences.
- `0` — Optimization problem could not be solved in the specified maximum number of iterations.
- `–1` — Optimization problem was infeasible. An optimization problem is infeasible if no solution can satisfy all the hard constraints.
- `–2` — Numerical error occurred when solving the optimization problem.

`QPCode` — Optimization solution status
`'feasible'` | `'infeasible'` | `'unrealiable'`

Optimization solution status, returned as one of the following:

- `'feasible'` — Optimal solution was obtained (`Iterations` > 0)
- `'infeasible'` — Solver detected a problem with no feasible solution (`Iterations` = –1) or a numerical error occurred (`Iterations` = –2)
- `'unreliable'` — Solver failed to converge (`Iterations` = 0). In this case, if `MPCobj.Optimizer.UseSuboptimalSolution` is `false`, `u` freezes at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.

`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. For more information, see “Optimization Problem”.

The cost value is only meaningful when `QPCode = 'feasible'`, or when `QPCode = 'feasible'` and `MPCobj.Optimizer.UseSuboptimalSolution` is `true`.

Tips

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

See Also

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

Introduced in R2014b

mpcmoveopt

Option set for `mpcmove` function

Description

To specify options for the `mpcmove`, `mpcmoveAdaptive`, and `mpcmoveMultiple` functions, use an `mpcmoveopt` object.

Using this object, you can specify run-time values for a subset of controller properties, such as tuning weights and constraints. If you do not specify a value for one of the `mpcmoveopt` properties, the value of the corresponding controller option is used instead.

Creation

Syntax

```
options = mpcmoveopt
```

Description

`options = mpcmoveopt` creates a default set of options for the `mpcmove` function. To modify the property values, use dot notation.

Properties

`OutputWeights` — Output variable tuning weights
`[ ]` (default) | vector | array

Output variable tuning weights that replace the `Weights.OutputVariables` property of the controller at run time, specified as a vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time k+1 to time k+p, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`MVWeights` — Manipulated variable tuning weights
[ ] (default) | vector | array

Manipulated variable tuning weights that replace the `Weights.ManipulatedVariables` property of the controller at run time, specified as a vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time k+p-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`MVRateWeights` — Manipulated variable rate tuning weights
[ ] (default) | vector | array

Manipulated variable rate tuning weights that replace the `Weights.ManipulatedVariablesRate` property of the controller at run time, specified as a vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time k+p-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`ECRWeight` — Slack variable tuning weight
[ ] (default) | positive scalar

Slack variable tuning weight that replaces the `Weights.ECR` property of the controller at run time, specified as a positive scalar.

`OutputMin` — Output variable lower bounds
[ ] (default) | row vector

Output variable lower bounds, specified as a row vector of length N_y, where N_y is the number of output variables. `OutputMin(i)` replaces the `OutputVariables(i).Min` property of the controller at run time.

If the `OutputVariables(i).Min` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), `OutputMin(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

`OutputMax` — Output variable upper bounds
[ ] (default) | row vector

Output variable upper bounds, specified as a row vector of length N_y, where N_y is the number of output variables. `OutputMax(i)` replaces the `OutputVariables(i).Max` property of the controller at run time.

If the `OutputVariables(i).Max` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), `OutputMax(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

`MVMin` — Manipulated variable lower bounds
[ ] (default) | row vector

Manipulated variable lower bounds, specified as a row vector of length N_{mv} , where N_{mv} is the number of manipulated variables. `MVMin(i)` replaces the `ManipulatedVariables(i).Min` property of the controller at run time.

If the `ManipulatedVariables(i).Min` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), `MVMin(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

`MVMax` — Manipulated variable upper bounds
[ ] (default) | row vector

Manipulated variable upper bounds, specified as a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables. `MVMax(i)` replaces the `ManipulatedVariables(i).Max` property of the controller at run time.

If the `ManipulatedVariables(i).Max` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), `MVMax(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

`CustomConstraint` — Custom mixed input/output constraints
`[]` (default) | structure

Custom mixed input/output constraints, specified as a structure with the following fields. These constraints replace the mixed input/output constraints previously set using `setconstraint`.

`E` — Manipulated variable constraint constant
matrix of zeros (default) | matrix

Manipulated variable constraint constant, specified as an N_c-by-N_{mv} array, where N_c is the number of constraints, and N_{mv} is the number of manipulated variables.

`F` — Controlled output constraint constant
matrix of zeros (default) | matrix

Controlled output constraint constant, specified as an N_c-by-N_y array, where N_y is the number of controlled outputs (measured and unmeasured).

`G` — Mixed input/output constraint constant
column vector of zeros (default) | column vector

Mixed input/output constraint constant, specified as a column vector of length N_c.

`S` — Measured disturbance constraint constant
matrix of zeros (default) | matrix

Measured disturbance constraint constant, specified as an N_c-by-N_v array, where N_v is the number of measured disturbances.

`OnlyComputeCost` — Flag indicating whether to calculate the optimal control sequence
`0` (default) | `1`

Flag indicating whether to calculate the optimal control sequence, specified as one of the following:

- `0` — Controller returns the predicted optimal control moves in addition to the objective function cost value.
- `1` — Controller returns the objective function cost only, which saves computational effort.

`MVused` — Manipulated variable values used in the plant during the previous control interval
`[]` (default) | row vector

Manipulated variable values used in the plant during the previous control interval, specified as a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables. If you do not specify `MVused`, the `mpvmove` uses the `LastMove` property of its current controller state input argument, `x`.

`MVTarget` — Manipulated variable targets
`[]` (default) | row vector

Manipulated variable targets, specified as a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables. `MVTarget(i)` replaces the `ManipulatedVariables(i).Target` property of the controller at run time.

Object Functions

mpcmove	Compute optimal control action
mpcmoveAdaptive	Compute optimal control with prediction model updating
mpcmoveMultiple	Compute gain-scheduling MPC control action at a single time instant

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);

-->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.

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

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

```
options = mpcmoveopt;
```

Pre-allocate storage and initialize the controller state.

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

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

Use `mpcmove` to simulate the following:

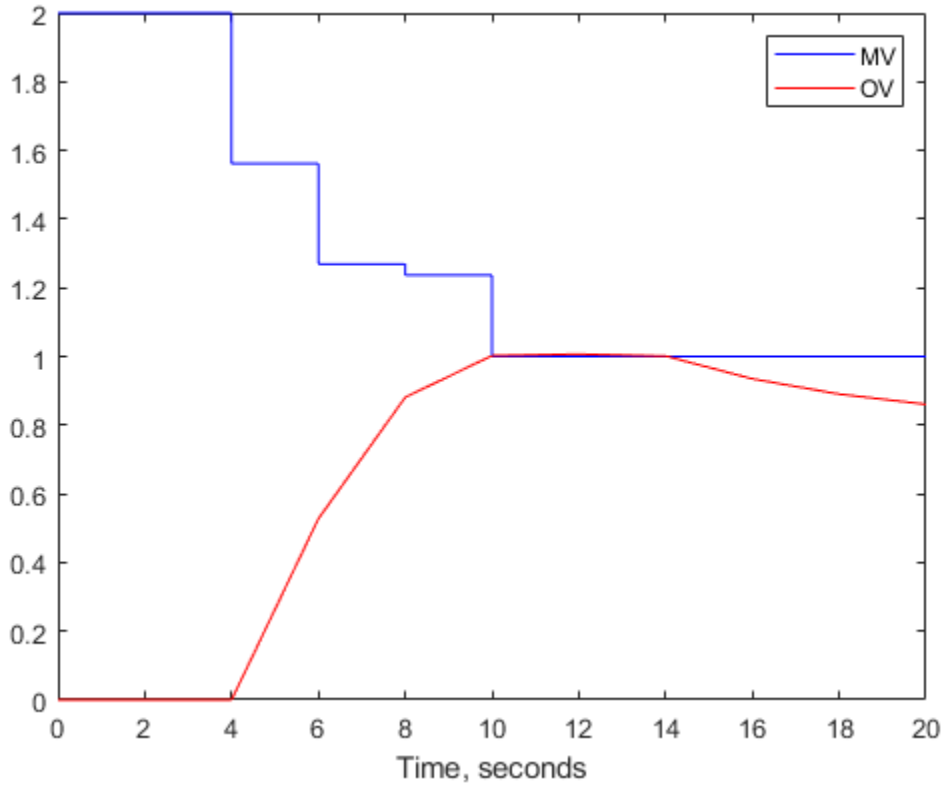
- Reference (setpoint) step change from initial condition $r = 0$ to $r = 1$ (servo response).
- MV upper bound step decrease from 2 to 1, occurring at $t = 10$.

```
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 active 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);

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

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
```

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, that is, 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];
```

However, using the `mpcmoveopt` object involves much less computational overhead.

Tips

- If a variable is unconstrained in the initial controller design, you cannot constrain it using `mpcmoveopt`. The controller ignores any such specifications.
- You cannot remove a constraint from a variable that is constrained in the initial controller design. However, you can change it to a large (or small) value such that it is unlikely to become active.

See Also

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

Introduced in R2018b

mpcprops

Provide help on MPC controller properties

Syntax

```
mpcprops
```

Description

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

See Also

`get` | `set`

Introduced before R2006a

mpcqpsolver

Solve a quadratic programming problem using the KWIK algorithm

Syntax

```
[x,status] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,options)
[x,status,iA,lambda] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,options)
```

Description

`[x,status] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,options)` finds an optimal solution, `x`, to a quadratic programming problem by minimizing the objective function:

$$J = \frac{1}{2} x^{\infty} H x + f^{\infty} x$$

subject to inequality constraints $Ax \geq b$, and equality constraints $A_{eq} x = b_{eq}$. `status` indicates the validity of `x`.

`[x,status,iA,lambda] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,options)` also returns the active inequalities, `iA`, at the solution, and the Lagrange multipliers, `lambda`, for the solution.

Examples

Solve Quadratic Programming Problem

Find the values of *x* that minimize

$$f(x) = 0.5x_1^2 + x_2^2 - x_1 x_2 - 2x_1 - 6x_2,$$

subject to the constraints

$$x_1 \geq 0$$
$$x_2 \geq 0$$
$$x_1 + x_2 \leq 2$$
$$-x_1 + 2x_2 \leq 2$$
$$2x_1 + x_2 \leq 3.$$

Specify the Hessian and linear multiplier vector for the objective function.

```
H = [1 -1; -1 2];
f = [-2; -6];
```

Specify the inequality constraint parameters.

```
A = [1 0; 0 1; -1 -1; 1 -2; -2 -1];
b = [0; 0; -2; -2; -3];
```

Define `Aeq` and `beq` to indicate that there are no equality constraints.

```
Aeq = [];
beq = zeros(0,1);
```

Find the lower-triangular Cholesky decomposition of `H`.

```
[L,p] = chol(H,'lower');
Linv = inv(L);
```

It is good practice to verify that `H` is positive definite by checking if `p = 0`.

```
p
```

```
p = 0
```

Create a default option set for `mpcqpsolver`.

```
opt = mpcqpsolverOptions;
```

To cold start the solver, define all inequality constraints as inactive.

```
iA0 = false(size(b));
```

Solve the QP problem.

```
[x,status] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,opt);
```

Examine the solution, x.

```
x
```

```
x = 2×1

    0.6667
    1.3333
```

Check Active Inequality Constraints for QP Solution

Find the values of x that minimize

$$f(x) = 3x_1^2 + 0.5x_2^2 - 2x_1x_2 - 3x_1 + 4x_2,$$

subject to the constraints

$$x_1 \geq 0$$
$$x_1 + x_2 \leq 5$$
$$x_1 + 2x_2 \leq 7.$$

Specify the Hessian and linear multiplier vector for the objective function.

```
H = [6 -2; -2 1];
f = [-3; 4];
```

Specify the inequality constraint parameters.

```
A = [1 0; -1 -1; -1 -2];
b = [0; -5; -7];
```

Define Aeq and beq to indicate that there are no equality constraints.

```
Aeq = [];
beq = zeros(0,1);
```

Find the lower-triangular Cholesky decomposition of H.

```
[L,p] = chol(H,'lower');
Linv = inv(L);
```

Verify that `H` is positive definite by checking if `p = 0`.

```
p
```

```
p = 0
```

Create a default option set for `mpcqpsolver`.

```
opt = mpcqpsolverOptions;
```

To cold start the solver, define all inequality constraints as inactive.

```
iA0 = false(size(b));
```

Solve the QP problem.

```
[x,status,iA,lambda] = mpcqpsolver(Linv,f,A,b,Aeq,beq,iA0,opt);
```

Check the active inequality constraints. An active inequality constraint is at equality for the optimal solution.

```
iA
```

```
iA = 3x1 logical array

   1
   0
   0
```

There is a single active inequality constraint.

View the Lagrange multiplier for this constraint.

```
lambda.ineqlin(1)
```

```
ans = 5.0000
```

Input Arguments

`Linv` — Inverse of lower-triangular Cholesky decomposition of Hessian matrix

n-by-*n* matrix

Inverse of lower-triangular Cholesky decomposition of Hessian matrix, specified as an *n*-by-*n* matrix, where $n > 0$ is the number of optimization variables. For a given Hessian matrix, *H*, `Linv` can be computed as follows:

```
[L,p] = chol(H,'lower');
Linv = inv(L);
```

H is an *n*-by-*n* matrix, which must be symmetric and positive definite. If $p>0$, then *H* is positive definite.

Note The KWIK algorithm requires the computation of `Linv` instead of using *H* directly, as in the `quadprog` command.

`f` — Multiplier of objective function linear term

column vector

Multiplier of objective function linear term, specified as a column vector of length *n*.

`A` — Linear inequality constraint coefficients

m-by-*n* matrix | `[]`

Linear inequality constraint coefficients, specified as an *m*-by-*n* matrix, where *m* is the number of inequality constraints.

If your problem has no inequality constraints, use `[]`.

`b` — Right-hand side of inequality constraints

column vector of length *m*

Right-hand side of inequality constraints, specified as a column vector of length *m*.

If your problem has no inequality constraints, use `zeros(0,1)`.

`Aeq` — Linear equality constraint coefficients

q-by-*n* matrix | `[]`

Linear equality constraint coefficients, specified as a *q*-by-*n* matrix, where *q* is the number of equality constraints, and `q <= n`. Equality constraints must be linearly independent with `rank(Aeq) = q`.

If your problem has no equality constraints, use `[]`.

`beq` — Right-hand side of equality constraints
column vector of length *q*

Right-hand side of equality constraints, specified as a column vector of length *q*.

If your problem has no equality constraints, use `zeros(0,1)`.

`iA0` — Initial active inequalities
logical vector of length *m*

Initial active inequalities, where the equal portion of the inequality is true, specified as a logical vector of length *m* according to the following:

- If your problem has no inequality constraints, use `false(0,1)`.
- For a *cold start*, `false(m,1)`.
- For a *warm start*, set `iA0(i) == true` to start the algorithm with the *i*th inequality constraint active. Use the optional output argument `iA` from a previous solution to specify `iA0` in this way. If both `iA0(i)` and `iA0(j)` are `true`, then rows *i* and *j* of `A` should be linearly independent. Otherwise, the solution can fail with `status = -2`.

`options` — Option set for `mpcqpsolver`
structure

Option set for `mpcqpsolver`, specified as a structure created using `mpcqpsolverOptions`.

Output Arguments

`x` — Optimal solution to the QP problem
column vector

Optimal solution to the QP problem, returned as a column vector of length *n*. `mpcqpsolver` always returns a value for `x`. To determine whether the solution is optimal or feasible, check the solution `status`.

`status` — Solution validity indicator
positive integer | `0` | `-1` | `-2`

Solution validity indicator, returned as an integer according to the following:

Value	Description
`> 0`	`x` is optimal. `status` represents the number of iterations performed during optimization.
`0`	The maximum number of iterations was reached. The solution, `x`, may be suboptimal or infeasible.
`-1`	The problem appears to be infeasible, that is, the constraint $Ax \geq b$ cannot be satisfied.
`-2`	An unrecoverable numerical error occurred.

`iA` — Active inequalities
logical vector of length *m*

Active inequalities, where the equal portion of the inequality is true, returned as a logical vector of length *m*. If `iA(i) == true`, then the *i*th inequality is active for the solution `x`.

Use `iA` to *warm start* a subsequent `mpcqpsolver` solution.

`lambda` — Lagrange multipliers
structure

Lagrange multipliers, returned as a structure with the following fields:

Field	Description
`ineqlin`	Multipliers of the inequality constraints, returned as a vector of length *n*. When the solution is optimal, the elements of `ineqlin` are nonnegative.
`eqlin`	Multipliers of the equality constraints, returned as a vector of length *q*. There are no sign restrictions in the optimal solution.

Tips

- The KWIK algorithm requires that the Hessian matrix, *H*, be positive definite. When calculating `Linv`, use:

```
[L, p] = chol(H,'lower');
```

If $p = 0$, then H is positive definite. Otherwise, p is a positive integer.

- `mpcqpsolver` provides access to the QP solver used by Model Predictive Control Toolbox software. Use this command to solve QP problems in your own custom MPC applications. For an example of a custom MPC application using `mpcqpsolver`, see "Solve Custom MPC Quadratic Programming Problem and Generate Code".

Algorithms

`mpcqpsolver` solves the QP problem using an active-set method, the KWIK algorithm, based on [1]. For more information, see "QP Solver".

The KWIK algorithm defines inequality constraints as $Ax \geq b$ rather than $Ax \leq b$, as in the `quadprog` command.

References

[1] Schmid, C., and L. T. Biegler. "Quadratic programming methods for reduced Hessian SQP." *Computers & Chemical Engineering*. Vol. 18, No. 9, 1994, pp. 817–832.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use `mpcqpsolver` as a general-purpose QP solver that supports code generation. Create the function `myCode` that uses `mpcqpsolver`.

```
function [out1,out2] = myCode(in1,in2)
%#codegen
...
[x,status] = mpcqpsolver(Linv,f,A,b,Aeq,Beq,iA0,options);
...
```

Generate C code with MATLAB Coder™.

```
func = 'myCode';
cfg = coder.config('mex'); % or 'lib', 'dll'
codegen('-config',cfg,func,'-o',func);
```

- For code generation, use the same precision for all real inputs, including options. Configure the precision as `'double'` or `'single'` using `mpcqpsolverOptions`.

See Also

`mpcqpsolverOptions` | `quadprog`

Topics

“QP Solver”
“Solve Custom MPC Quadratic Programming Problem and Generate Code”

Introduced in R2015b

mpcqpsolverOptions

Create default option set for `mpcqpsolver`

Syntax

```
options = mpcqpsolverOptions
options = mpcqpsolverOptions(type)
```

Description

`options = mpcqpsolverOptions` creates a structure of default options for `mpcqpsolver`, which solves a quadratic programming (QP) problem using the KWIK algorithm.

`options = mpcqpsolverOptions(type)` creates a default option set using the specified input data type. All real options are specified using this data type.

Examples

Create Default Option Set for MPC QP Solver

```
opt = mpcqpsolverOptions;
```

Create and Modify Default MPC QP Solver Option Set

Create default option set.

```
opt = mpcqpsolverOptions;
```

Specify the maximum number of iterations allowed during computation.

```
opt.MaxIter = 100;
```

Specify a feasibility tolerance for verifying that the optimal solution satisfies the inequality constraints.

```
opt.FeasibilityTol = 1.0e-3;
```

Create Option Set Specifying Input Argument Type

```
opt = mpcqpsolverOptions('single');
```

Input Arguments

`type` — MPC QP solver input argument data type

`'double'` (default) | `'single'`

MPC QP solver input argument data type, specified as either `'double'` or `'single'`. This data type is used for both simulation and code generation. All real options in the option set are specified using this data type, and all real input arguments to `mpcqpsolver` must match this type.

Output Arguments

`options` — Option set for `mpcqpsolver`

structure

Option set for `mpcqpsolver`, returned as a structure with the following fields:

Field	Description	Default
`DataType`	Input argument data type, specified as either `'double'` or `'single'`. This data type is used for both simulation and code generation, and all real input arguments to `mpcqpsolver` must match this type.	`'double'`
`MaxIter`	Maximum number of iterations allowed when computing the QP solution, specified as a positive integer.	`200`

Field	Description	Default
`FeasibilityTol`	Tolerance used to verify that inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger `FeasibilityTol` value allows for larger constraint violations.	`1.0e-6`
`IntegrityChecks`	Indicator of whether integrity checks are performed on the `mpcqpsolver` input data, specified as a logical value. If `IntegrityChecks` is `true`, then integrity checks are performed and diagnostic messages are displayed. Use `false` for code generation only.	`true`

See Also

`mpcqpsolver`

Introduced in R2015b

mpcsimopt

MPC simulation options

Syntax

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

Description

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

Output Arguments

`options`

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

MPC Simulation Options Properties

Property	Description
`PlantInitialState`	Initial state vector of the plant model generating the data.
`ControllerInitialState`	Initial condition of the MPC controller. This must be a valid `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'`
`MDLookAhead`	Preview on measured disturbance signal (`'on'` or `'off'`).
`Constraints`	Use MPC constraints (`'on'` or `'off'`). Default: `'on'`

Property	Description
`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'`

Examples

Simulate MPC Control with Plant Model Mismatch

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

Define the predicted plant model.

```
p1 = tf(1,[1 2 1])*[1 1;0 1];
plantPredict = ss([p1 p1]);
plantPredict.InputName = {'mv1','mv2','umd3','umd4'};
```

Specify the MPC signal types.

```
plantPredict = setmpcsignals(plantPredict,'MV',[1 2],'UD',[3 4]);
```

Create the MPC controller.

```
mpcobj = mpc(plantPredict,1,40,2);
```

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

Define the unmeasured input disturbance model used by the controller.

```
distModel = eye(2,2)*ss(-0.5,1,1,0);
mpcobj.Model.Disturbance = distModel;
```

Define an actual plant model which differs from the predicted model and has unforeseen unmeasured disturbance inputs.

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

Configure the unmeasured disturbance and output reference trajectories.

```
dist = ones(1,3);
refs = [1 2];
```

Create and configure a simulation option set.

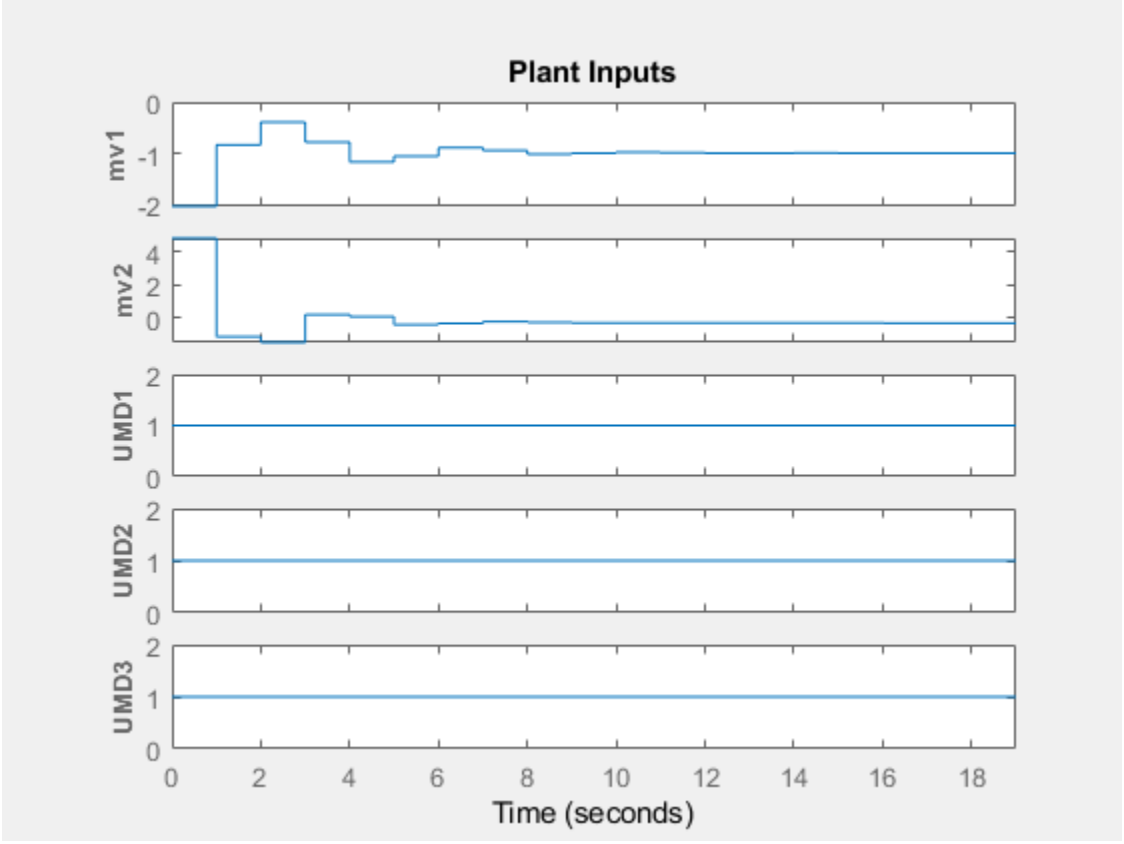
```
options = mpcsimopt(mpcobj);
options.UnmeasuredDisturbance = dist;
options.Model = plantActual;
```

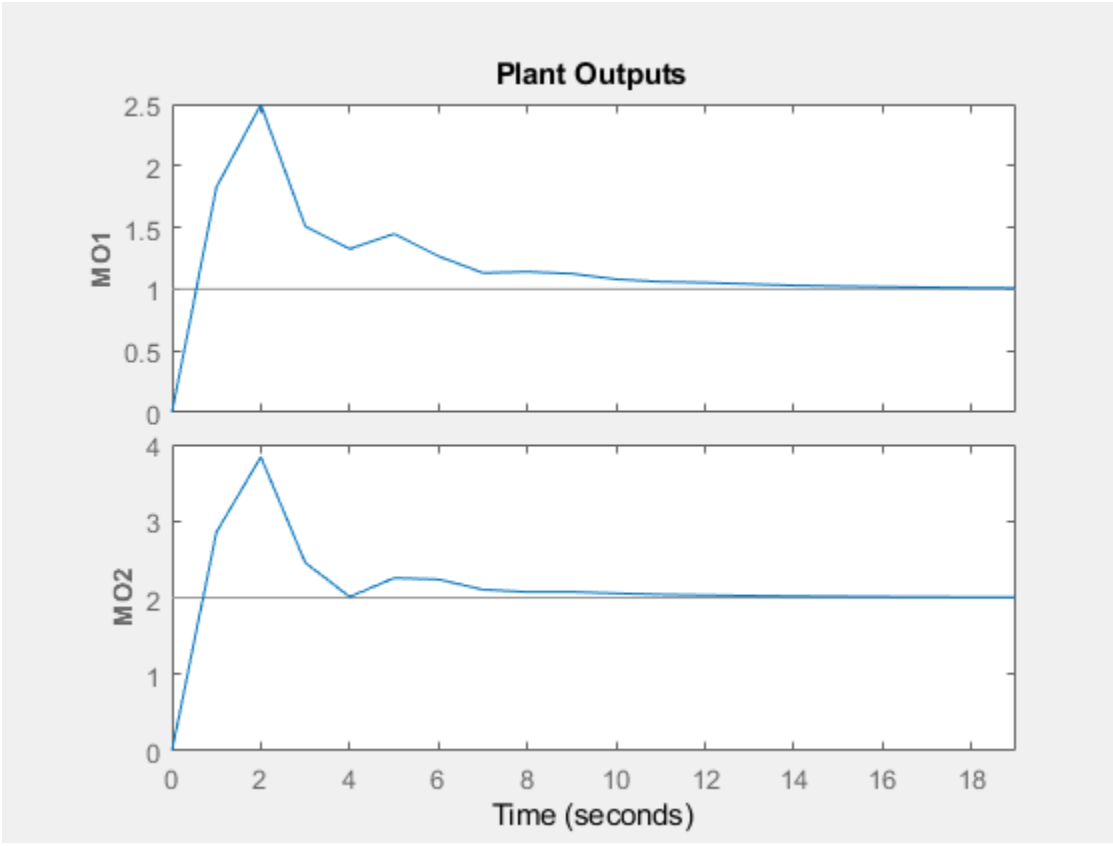
Simulate the system.

```
sim(mpcobj,20,refs,options)
```

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

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





See Also

`sim`

Topics

“Simulate Controller with Nonlinear Plant”

Introduced before R2006a

mpcstate

Define MPC controller state

Syntax

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

Description

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

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

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

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

Input Arguments

`MPCobj`

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

xp

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

xd

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

xn

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

u

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

p

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

Output Arguments

xmpc

MPC state object, containing the following properties.

Property	Description
`Plant`	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 `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.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
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 = 4×4

    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

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

Introduced before R2006a

mpcverbosity

Change toolbox verbosity level

Syntax

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

Description

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

`mpcverbosity off` turns messages off.

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

`mpcverbosity` just shows the verbosity status.

By default, messages are turned on.

See also “Construction and Initialization” on page 4-15 .

See Also

`mpc`

Introduced before R2006a

nlmpc

Nonlinear model predictive controller

Description

A nonlinear model predictive controller computes optimal control moves across the prediction horizon using a nonlinear prediction model, a nonlinear cost function, and nonlinear constraints. For more information on nonlinear MPC, see “Nonlinear MPC”.

Creation

Syntax

```
nlobj = nlmpc(nx,ny,nu)

nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex)
nlobj = nlmpc(nx,ny,'MV',mvIndex,'UD',udIndex)
nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex,'UD',udIndex)
```

Description

`nlobj = nlmpc(nx,ny,nu)` creates an `nlmpc` object whose prediction model has `nx` states, `ny` outputs, and `nu` inputs, where all inputs are manipulated variables. Use this syntax if your model has no measured or unmeasured disturbance inputs.

`nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex)` creates an `nlmpc` object whose prediction model has measured disturbance inputs. Specify the input indices for the manipulated variables, `mvIndex`, and measured disturbances, `mdIndex`.

`nlobj = nlmpc(nx,ny,'MV',mvIndex,'UD',udIndex)` creates an `nlmpc` object whose prediction model has unmeasured disturbance inputs. Specify the input indices for the manipulated variables and unmeasured disturbances, `udIndex`.

`nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex,'UD',udIndex)` creates an `nlmpc` object whose prediction model has both measured and unmeasured disturbance

inputs. Specify the input indices for the manipulated variables, measured disturbances, and unmeasured disturbances.

Input Arguments

`nx` — Number of prediction model states

positive integer

Number of prediction model states, specified as a positive integer. You cannot change the number of states after creating the controller object.

`ny` — Number of prediction model outputs

positive integer

Number of prediction model outputs, specified as a positive integer. You cannot change the number of outputs after creating the controller object.

`nu` — Number of prediction model inputs

positive integer

Number of prediction model inputs, which are all manipulated variables, specified as a positive integer. You cannot change the number of manipulated variables after creating the controller object.

`mvIndex` — Manipulated variable indices

vector of positive integers

Manipulated variable indices, specified as a vector of positive integers. You cannot change these indices after creating the controller object. This value is stored in the `Dimensions.MVIndex` controller property.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from `1` through N_u, where N_u is the number of prediction model inputs.

`mdIndex` — Measured disturbance indices

vector of positive integers

Measured disturbance indices, specified as a vector of positive integers. You cannot change these indices after creating the controller object. This value is stored in the `Dimensions.MDIndex` controller property.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from `1` through N_u, where N_u is the number of prediction model inputs.

`udIndex` — Unmeasured disturbance indices
vector of positive integers

Unmeasured disturbance indices, specified as a vector of positive integers. You cannot change these indices after creating the controller object. This value is stored in the `Dimensions.UDIndex` controller property.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from `1` through N_u, where N_u is the number of prediction model inputs.

Properties

`Ts` — Prediction model sample time
`1` (default) | positive finite scalar

Prediction model sample time, specified as a positive finite scalar. The controller uses a discrete-time model with sample time `Ts` for prediction. If you specify a continuous-time prediction model (`Model.IsContinuousTime` is `true`), then the controller discretizes the model using the built-in implicit trapezoidal rule with a sample time of `Ts`.

`PredictionHorizon` — Prediction horizon
`10` (default) | positive integer

Prediction horizon steps, specified as a positive integer. The product of `PredictionHorizon` and `Ts` is the prediction time; that is, how far the controller looks into the future.

`ControlHorizon` — Control horizon
`2` (default) | positive integer | vector of positive integers

Control horizon, specified as one of the following:

- Positive integer, m, between `1` and p, inclusive, where p is equal to `PredictionHorizon`. In this case, the controller computes m free control moves occurring at times k through $k+m-1$, and holds the controller output constant for the remaining prediction horizon steps from $k+m$ through $k+p-1$. Here, k is the current control interval. For optimal trajectory planning set m equal to p.

- Vector of positive integers, [m_1, m_2, ...], where the sum of the integers equals the prediction horizon, p. In this case, the controller computes M blocks of free moves, where M is the length of the `ControlHorizon` vector. The first free move applies to times k through $k+m_1-1$, the second free move applies from time $k+m_1$ through $k+m_1+m_2-1$

  , and so on. Using block moves can improve the robustness of your controller compared to the default case.

`Dimensions` — Prediction model dimensional information
structure

This property is read-only.

Prediction model dimensional information, specified when the controller is created and stored as a structure with the following fields.

`NumberOfStates` — Number of states
positive integer

Number of states in the prediction model, specified as a positive integer. This value corresponds to `nx`.

`NumberOfOutputs` — Number of outputs
positive integer

Number of outputs in the prediction model, specified as a positive integer. This value corresponds to `ny`.

`NumberOfInputs` — Number of states
positive integer

Number of inputs in the prediction model, specified as a positive integer. This value corresponds to either `nmv` or the sum of the lengths of `mvIndex`, `mdIndex`, and `udIndex`.

`MVIndex` — Manipulated variable indices
vector of positive integers

Manipulated variable indices for the prediction model, specified as a vector of positive integers. This value corresponds to `mvIndex`.

`MDIndex` — Measured disturbance indices
vector of positive integers

Measured disturbance indices for the prediction model, specified as a vector of positive integers. This value corresponds to `mdIndex`.

`UDIndex` — Unmeasured disturbance indices
vector of positive integers

Unmeasured disturbance indices for the prediction model, specified as a vector of positive integers. This value corresponds to `udIndex`.

`Model` — Prediction model
structure

Prediction model, specified as a structure with the following fields.

`StateFcn` — State function
string | character vector | function handle

State function, specified as a string, character vector, or function handle. For a continuous-time prediction model, `StateFcn` is the state derivative function. For a discrete-time prediction model, `StateFcn` is the state update function.

If your state function is continuous-time, the controller automatically discretizes the model using the implicit trapezoidal rule. This method can handle moderately stiff models, and its prediction accuracy depends on the controller sample time `Ts`; that is, a large sample time leads to inaccurate prediction.

If the default discretization method does not provide satisfactory prediction for your application, you can specify your own discrete-time prediction model that uses a different method, such as the multistep forward Euler rule.

You can specify your state function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myStateFunction";
  ```
- Handle to a function in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myStateFunction;
  ```
- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myStateFunction(x,u,params)
  ```

For more information, see “Specify Prediction Model for Nonlinear MPC”.

`OuputFcn` — Output function
`[ ]` (default) | string | character vector | function handle

Output function, specified as a string, character vector, or function handle. If the number of states and outputs of the prediction model are the same, you can omit `OutputFcn`, which implies that all states are measurable; that is, each output corresponds to one state.

Note You output function cannot have direct feedthrough from any manipulated variable to any output at any time.

You can specify your output function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.OutputFcn = "myOutputFunction";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Model.OutputFcn = @myOutputFunction;
  ```

- Anonymous function

  ```
  Model.OutputFcn = @(x,u,params) myOutputFunction(x,u,params)
  ```

For more information, see “Specify Prediction Model for Nonlinear MPC”.

`IsContinuousTime` — Flag indicating prediction model time domain
`true` (default) | `false`

Flag indicating prediction model time domain, specified as one of the following:

- `true` — Continuous-time prediction model. In this case, the controller automatically discretizes the model during prediction using `Ts`.
- `false` — Discrete-time prediction model. In this case, `Ts` is the sample time of the model.

Note `IsContinuousTime` must be consistent with the functions specified in `Model.StateFcn` and `Model.OutputFcn`.

`NumberOfParameters` — Number of optional model parameters

`0` (default) | nonnegative integer

Number of optional model parameters used by the prediction model, custom cost function, and custom constraint functions, specified as a nonnegative integer. The number of parameters includes all the parameters used by these functions. For example, if the state function uses only parameter `p1`, the constraint functions use only parameter `p2`, and the cost function uses only parameter `p3`, then `NumberOfParameters` is `3`.

`States` — State information, bounds, and scale factors

structure array

State information, bounds, and scale factors, specified as a structure array with N_x elements, where N_x is the number of states. Each structure element has the following fields.

`Min` — State lower bound

`-Inf` (default) | scalar | vector

State lower bound, specified as a scalar or vector. By default, the lower bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

State bounds are always hard constraints.

`Max` — State upper bound

`Inf` (default) | scalar | vector

State upper bound, specified as a scalar or vector. By default, the upper bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

State bounds are always hard constraints.

`Name` — State name
string | character vector

State name, specified as a string or character vector. The default state name is `"x#"`, where `#` is its state index.

`Units` — State units
`""` (default) | string | character vector

State units, specified as a string or character vector.

`ScaleFactor` — State scale factor
`1` (default) | positive finite scalar

State scale factor, specified as a positive finite scalar. In general, use the operating range of the state. Specifying the proper scale factor can improve numerical conditioning for optimization.

`OutputVariables` — Output variable information, bounds, and scale factors
structure array

Output variable (OV) information, bounds, and scale factors, specified as a structure array with N_y elements, where N_y is the number of output variables. To access this property, you can use the alias `OV` instead of `OutputVariables`.

Each structure element has the following fields.

`Min` — OV lower bound
`-Inf` (default) | scalar | vector

OV lower bound, specified as a scalar or vector. By default, this lower bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

`Max` — OV upper bound
`Inf` (default) | scalar | vector

OV upper bound, specified as a scalar or vector. By default, this upper bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

`MinECR` — OV lower bound softness
1 (default) | nonnegative finite scalar | vector

OV lower bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, OV upper bounds are soft constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

`MaxECR` — OV upper bound softness
1 (default) | nonnegative finite scalar | vector

OV upper bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, OV lower bounds are soft constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

`Name` — OV name
string | character vector

OV name, specified as a string or character vector. The default OV name is `"y#"`, where `#` is its output index.

`Units` — OV units
`""` (default) | string | character vector

OV units, specified as a string or character vector.

`ScaleFactor` — OV scale factor
`1` (default) | positive finite scalar

OV scale factor, specified as a positive finite scalar. In general, use the operating range of the output variable. Specifying the proper scale factor can improve numerical conditioning for optimization.

`ManipulatedVariables` — Manipulated variable information, bounds, and scale factors
structure array

Manipulated Variable (MV) information, bounds, and scale factors, specified as a structure array with N_{mv} elements, where N_{mv} is the number of manipulated variables. To access this property, you can use the alias `MV` instead of `ManipulatedVariables`.

Each structure element has the following fields.

`Min` — MV lower bound
`-Inf` (default) | scalar | vector

MV lower bound, specified as a scalar or vector. By default, this lower bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

`Max` — MV upper bound
`Inf` (default) | scalar | vector

MV upper bound, specified as a scalar or vector. By default, this upper bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

`MinECR` — MV lower bound softness
`0` (default) | nonnegative scalar | vector

MV lower bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative scalar or vector. By default, MV lower bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

`MaxECR` — MV upper bound
`0` (default) | nonnegative scalar | vector

MV upper bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative scalar or vector. By default, MV upper bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

`RateMin` — MV rate of change lower bound
`-Inf` (default) | nonpositive scalar | vector

MV rate of change lower bound, specified as a nonpositive scalar or vector. The MV rate of change is defined as $MV(k) - MV(k-1)$, where k is the current time. By default, this lower bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

`RateMax` — MV rate of change upper bound
`Inf` (default) | nonnegative scalar | vector

MV rate of change upper bound, specified as a nonnegative scalar or vector. The MV rate of change is defined as $MV(k) - MV(k-1)$, where k is the current time. By default, this lower bound is unconstrained.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

`RateMinECR` — MV rate of change lower bound softness
`0` (default) | nonnegative finite scalar | vector

MV rate of change lower bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, MV rate of change lower bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR values over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR values are used for the remaining steps of the prediction horizon.

`RateMaxECR` — MV rate of change upper bound softness
`0` (default) | nonnegative finite scalar | vector

MV rate of change upper bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, MV rate of change upper bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR values over the prediction horizon from time k to time $k+p-1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR values are used for the remaining steps of the prediction horizon.

`Name` — MV name
string | character vector

MV name, specified as a string or character vector. The default MV name is `"u#"`, where `#` is its input index.

`Units` — MV units
`""` (default) | string | character vector

MV units, specified as a string or character vector.

`ScaleFactor` — MV scale factor
1 (default) | positive finite scalar

MV scale factor, specified as a positive finite scalar. In general, use the operating range of the manipulated variable. Specifying the proper scale factor can improve numerical conditioning for optimization.

`MeasuredDisturbances` — Measured disturbance information and scale factors
structure array

Measured disturbance (MD) information and scale factors, specified as a structure array with N_{md} elements, where N_{md} is the number of measured disturbances. If your model does not have measured disturbances, then `MeasuredDisturbances` is `[]`.To access this property, you can use the alias `MD` instead of `MeasuredDisturbances`.

Each structure element has the following fields.

`Name` — MD name
string | character vector

MD name, specified as a string or character vector. The default MD name is `"u#"`, where `#` is its input index.

`Units` — MD units
`""` (default) | string | character vector

MD units, specified as a string or character vector.

`ScaleFactor` — MD scale factor

1 (default) | positive finite scalar

MD scale factor, specified as a positive finite scalar. In general, use the operating range of the disturbance. Specifying the proper scale factor can improve numerical conditioning for optimization.

`Weights` — Standard cost function tuning weights

structure

Standard cost function tuning weights, specified as a structure. The controller applies these weights to the scaled variables. Therefore, the tuning weights are dimensionless values.

Note If you define a custom cost function using `Optimization.CustomCostFcn` and set `Optimization.ReplaceStandardCost` to `true`, then the controller ignores the standard cost function tuning weights in `Weights`.

`Weights` has the following fields.

`ManipulatedVariables` — Manipulated variable tuning weights

row vector | array

Manipulated variable tuning weights, which penalize deviations from MV targets, specified as a row vector or array of nonnegative values. The default weight for all manipulated variables is `0`.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

To specify MV targets at run time, create an `nlmpcmoveopt` object, and set its `MVTarget` property.

`ManipulatedVariablesRate` — Manipulated variable rate tuning weights

row vector | array

Manipulated variable rate tuning weights, which penalize large changes in control moves, specified as a row vector or array of nonnegative values. The default weight for all manipulated variable rates is `0.1`.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`OutputVariables` — Output variable tuning weights
vector | array

Output variable tuning weights, which penalize deviation from output references, specified as a row vector or array of nonnegative values. The default weight for all output variables is `1`.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`ECR` — Slack variable tuning weight
`1e5` (default) | positive scalar

Slack variable tuning weight, specified as a positive scalar.

`Optimization` — Custom optimization functions and solver
structure

Custom optimization functions and solver, specified as a structure with the following fields.

`CustomCostFcn` — Custom cost function
`[]` | string | character vector | function handle

Custom cost function, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomCostFcn = "myCostFunction";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Optimization.CustomCostFcn = @myCostFunction;
  ```

- Anonymous function

  ```
  Optimization.CustomCostFcn = @(X,U,e,data,params) myCostFunction(X,U,e,data,params);
  ```

Your cost function must have the signature:

```
function J = myCostFunction(X,U,e,data,params)
```

For more information, see “Specify Cost Function for Nonlinear MPC”.

`ReplaceStandardCost` — Flag indicating whether to replace the standard cost function
`true` (default) | `false`

Flag indicating whether to replace the standard cost function with the custom cost function, specified as one of the following:

- `true` — The controller uses the custom cost alone as the objective function during optimization. In this case, the `Weights` property of the controller is ignored.
- `false` — The controller uses the sum of the standard cost and custom cost as the objective function during optimization.

If you do not specify a custom cost function using `CustomCostFcn`, then the controller ignores `RepalceStandardCost`.

For more information, see “Specify Cost Function for Nonlinear MPC”.

`CustomEqConFcn` — Custom equality constraint function
`[ ]` (default) | string | character vector | function handle

Custom equality constraint function, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

```
Optimization.CustomEqConFcn = "myEqConFunction";
```

- Handle to a function in the current working folder or on the MATLAB path

```
Optimization.CustomEqConFcn = @myEqConFunction;
```

- Anonymous function

```
Optimization.CustomEqConFcn = @(X,U,data,params) myEqConFunction(X,U,data,params);
```

Your equality constraint function must have the signature:

```
function ceq = myEqConFunction(X,U,data,p1,p2,...)
```

For more information, see “Specify Constraints for Nonlinear MPC”.

`CustomIneqConFcn` — Custom inequality constraint function
[ ] (default) | string | character vector | function handle

Custom inequality constraint function, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

```
Optimization.CustomIneqConFcn = "myIneqConFunction";
```

- Handle to a function in the current working folder or on the MATLAB path

```
Optimization.CustomIneqConFcn = @myIneqConFunction;
```

- Anonymous function

```
Optimization.CustomIneqConFcn = @(X,U,data,params) myIneqConFunction(X,U,data,params);
```

Your equality constraint function must have the signature:

```
function cineq = myIneqConFunction(X,U,data,params)
```

For more information, see “Specify Constraints for Nonlinear MPC”.

`CustomSolverFcn` — Custom nonlinear programming solver
[ ] (default) | string | character vector | function handle

Custom nonlinear programming solver function, specified as a string, character vector, or function handle. If you do not have Optimization Toolbox™ software, you must specify your own custom nonlinear programming solver. You can specify your custom solver function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomSolverFcn = "myNLPSolver";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Optimization.CustomSolverFcn = @myNLPSolver;
  ```

For more information, see “Configure Optimization Solver for Nonlinear MPC”.

`SolverOptions` — Solver options
options object for `fmincon` | `[]`

Solver options, specified as an options object for `fmincon` or `[]`.

If you have Optimization Toolbox software, `SolverOptions` contains an options object for the `fmincon` solver.

If you do not have Optimization Toolbox, `SolverOptions` is `[]`.

For more information, see “Configure Optimization Solver for Nonlinear MPC”.

`UseSuboptimalSolution` — Flag indicating whether a suboptimal solution is acceptable
`false` (default) | `true`

Flag indicating whether a suboptimal solution is acceptable, specified as a logical value. When the nonlinear programming solver reaches the maximum number of iterations without finding a solution (the exit flag is `0`), the controller:

- Freezes the MV values if `UseSuboptimalSolution` is `false`
- Applies the suboptimal solution found by the solver after the final iteration if `UseSuboptimalSolution` is `true`

To specify the maximum number of iterations, use `Optimization.SolverOptions.MaxIter`.

`Jacobian` — Jacobians of model functions, and custom cost and constraint functions
structure

Jacobians of model functions, and custom cost and constraint functions, specified as a structure. It is best practice to use Jacobians whenever they are available, since they

improve optimization efficiency. If you do not specify a Jacobian for a given function, the nonlinear programming solver must numerically compute the Jacobian.

The `Jacobian` structure contains the following fields.

`StateFcn` — Jacobian of state function
[ ] (default) | string | character vector | function handle

Jacobian of state function `z` from `Model.StateFcn`, specified as one of the following

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myStateJacobian";
  ```
- Handle to a function in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myStateJacobian;
  ```
- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myStateJacobian(x,u,params)
  ```

For more information, see “Specify Prediction Model for Nonlinear MPC”.

`OuputFcn` — Jacobian of output function
[ ] (default) | string | character vector | function handle

Jacobian of output function `y` from `Model.OutputFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myOutputJacobian";
  ```
- Handle to a function in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myOutputJacobian;
  ```
- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myOutputJacobian(x,u,params)
  ```

For more information, see “Specify Prediction Model for Nonlinear MPC”.

`CustomCostFcn` — Jacobian of custom cost function
[ ] | string | character vector | function handle

Jacobian of custom cost function `J` from `Optimization.CustomCostFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.CustomCostFcn = "myCostJacobian";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.CustomCostFcn = @myCostJacobian;
  ```

- Anonymous function

  ```
  Jacobian.CustomCostFcn = @(X,U,e,data,params) myCostJacobian(X,U,e,data,params)
  ```

Your cost Jacobian function must have the signature:

```
function [G,Gmv,Ge] = myCostJacobian(X,U,e,data,params)
```

For more information, see “Specify Cost Function for Nonlinear MPC”.

`CustomEqConFcn` — Jacobian of custom equality constraints
`[]` (default) | string | character vector | function handle

Jacobian of custom equality constraints `ceq` from `Optimization.CustomEqConFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.CustomEqConFcn = "myEqConJacobian";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.CustomEqConFcn = @myEqConJacobian;
  ```

- Anonymous function

  ```
  Jacobian.CustomEqConFcn = @(X,U,data,params) myEqConJacobian(X,U,data,params);
  ```

Your equality constraint Jacobian function must have the signature:

```
function [G,Gmv] = myEqConJacobian(X,U,data,params)
```

For more information, see “Specify Constraints for Nonlinear MPC”.

`CustomIneqConFcn` — Jacobian of custom inequality constraints

[ ] (default) | string | character vector | function handle

Jacobian of custom inequality constraints `c` from `Optimization.CustomIneqConFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.CustomEqConFcn = "myIneqConJacobian";
  ```

- Handle to a function in the current working folder or on the MATLAB path

  ```
  Jacobian.CustomEqConFcn = @myIneqConJacobian;
  ```

- Anonymous function

  ```
  Jacobian.CustomEqConFcn = @(X,U,data,params) myIneqConJacobian(X,U,data,params);
  ```

Your inequality constraint Jacobian function must have the signature:

```
function [G,Gmv,Ge] = myIneqConJacobian(X,U,data,params)
```

For more information, see “Specify Constraints for Nonlinear MPC”.

Object Functions

nlmpcmove	Compute optimal control action for nonlinear MPC controller
validateFcns	Examine prediction model and custom functions of nlmpc object for potential problems
convertToMPC	Convert nlmpc object into one or more mpc objects
createParameterBus	Create Simulink bus object and configure Bus Creator block for passing model parameters to Nonlinear MPC Controller block

Examples

Create Nonlinear MPC Controller with Discrete-Time Prediction Model

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nx = 4;
ny = 2;
```

```
nu = 1;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The discrete-time state function uses an optional parameter, the sample time `Ts`, to integrate the continuous-time model. Therefore, you must specify the number of optional parameters as `1`.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function for the controller. In this case, define the first and third states as outputs. Even though this output function does not use the optional sample time parameter, you must specify the parameter as an input argument (`Ts`).

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Validate the prediction model functions for nominal states `x0` and nominal inputs `u0`. Since the prediction model uses a custom parameter, you must pass this parameter to `validateFcns`.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Nonlinear MPC Controller with Measured and Unmeasured Disturbances

Create a nonlinear MPC controller with three states, one output, and four inputs. The first two inputs are measured disturbances, the third input is the manipulated variable, and the fourth input is an unmeasured disturbance.

```
nlobj = nlmpc(3,1,'MV',3,'MD',[1 2],'UD',4);
```

To view the controller state, output, and input dimensions and indices, use the `Dimensions` property of the controller.

```
nlobj.Dimensions
```

```
ans = struct with fields:
     NumberOfStates: 3
    NumberOfOutputs: 1
     NumberOfInputs: 4
            MVIndex: 3
            MDIndex: [1 2]
            UDIndex: 4
```

Specify the controller sample time and horizons.

```
nlobj.Ts = 0.5;
nlobj.PredictionHorizon = 6;
nlobj.ControlHorizon = 3;
```

Specify the prediction model state function, which is in the file `exocstrStateFcnCT.m`.

```
nlobj.Model.StateFcn = 'exocstrStateFcnCT';
```

Specify the prediction model output function, which is in the file `exocstrOutputFcn.m`.

```
nlobj.Model.OutputFcn = 'exocstrOutputFcn';
```

Validate the prediction model functions using the initial operating point as the nominal condition for testing and setting the unmeasured disturbance state, `x0(3)`, to `0`. Since the model has measured disturbances, you must pass them to `validateFcns`.

```
x0 = [311.2639; 8.5698; 0];
u0 = [10; 298.15; 298.15];
validateFcns(nlobj,x0,u0(3),u0(1:2)');
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Validate Nonlinear MPC Prediction Model and Custom Functions

Create nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Validate the prediction model and custom functions at the initial states (`x0`) and initial inputs (`u0`) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);

Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Linear MPC Controllers from Nonlinear MPC Controller

Create a nonlinear MPC controller with four states, one output variable, one manipulated variable, and one measured disturbance.

```
nlobj = nlmpc(4,1,'MV',1,'MD',2);
```

Specify the controller sample time and horizons.

```
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 3;
```

Specify the state function of the prediction model.

```
nlobj.Model.StateFcn = 'oxidationStateFcn';
```

Specify the prediction model output function and the output variable scale factor.

```
nlobj.Model.OutputFcn = @(x,u) x(3);
nlobj.OutputVariables.ScaleFactor = 0.03;
```

Specify the manipulated variable constraints and scale factor.

```
nlobj.ManipulatedVariables.Min = 0.0704;
nlobj.ManipulatedVariables.Max = 0.7042;
nlobj.ManipulatedVariables.ScaleFactor = 0.6;
```

Specify the measured disturbance scale factor.

```
nlobj.MeasuredDisturbances.ScaleFactor = 0.5;
```

Compute the state and input operating conditions for three linear MPC controllers using the `fsolve` function.

```
options = optimoptions('fsolve','Display','none');

uLow = [0.38 0.5];
xLow = fsolve(@(x) oxidationStateFcn(x,uLow),[1 0.3 0.03 1],options);

uMedium = [0.24 0.5];
xMedium = fsolve(@(x) oxidationStateFcn(x,uMedium),[1 0.3 0.03 1],options);

uHigh = [0.15 0.5];
xHigh = fsolve(@(x) oxidationStateFcn(x,uHigh),[1 0.3 0.03 1],options);
```

Create linear MPC controllers for each of these nominal conditions.

```
mpcobjLow = convertToMPC(nlobj,xLow,uLow);
mpcobjMedium = convertToMPC(nlobj,xMedium,uMedium);
mpcobjHigh = convertToMPC(nlobj,xHigh,uHigh);
```

You can also create multiple controllers using arrays of nominal conditions. The number of rows in the arrays specifies the number controllers to create. The linear controllers are returned as cell array of `mpc` objects.

```
u = [uLow; uMedium; uHigh];
x = [xLow; xMedium; xHigh];
mpcobjs = convertToMPC(nlobj,x,u);
```

View the properties of the `mpcobjLow` controller.

```
mpcobjLow
```

```
MPC object (created on 27-Aug-2018 17:34:57):
---------------------------------------------
Sampling time:      1 (seconds)
Prediction Horizon: 10
Control Horizon:    3

Plant Model:
                                    --------------
      1  manipulated variable(s)   -->|  4 states  |
                                     |             |-->  1 measured output(s)
      1  measured disturbance(s)   -->|  2 inputs  |
                                     |             |-->  0 unmeasured output(s)
```

```
      0  unmeasured disturbance(s) -->|  1 outputs |
                                      --------------
Indices:
  (input vector)    Manipulated variables: [1 ]
                    Measured disturbances: [2 ]
  (output vector)        Measured outputs: [1 ]

Disturbance and Noise Models:
        Output disturbance model: default (type "getoutdist(mpcobjLow)" for details)
         Measurement noise model: default (unity gain after scaling)

Weights:
        ManipulatedVariables: 0
    ManipulatedVariablesRate: 0.1000
             OutputVariables: 1
                         ECR: 100000

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

Constraints:
 0.0704 <= u1 <= 0.7042, u1/rate is unconstrained, y1 is unconstrained
```

Plan Optimal Trajectory Using Nonlinear MPC

Create nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Specify linear constraints on the manipulated variables.

```
for ct = 1:nu
    nlobj.MV(ct).Min = 0;
    nlobj.MV(ct).Max = 1;
end
```

Validate the prediction model and custom functions at the intial states (`x0`) and initial inputs (`u0`) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Compute the optimal state and manipulated variable trajectories, which are returned in the `info`.

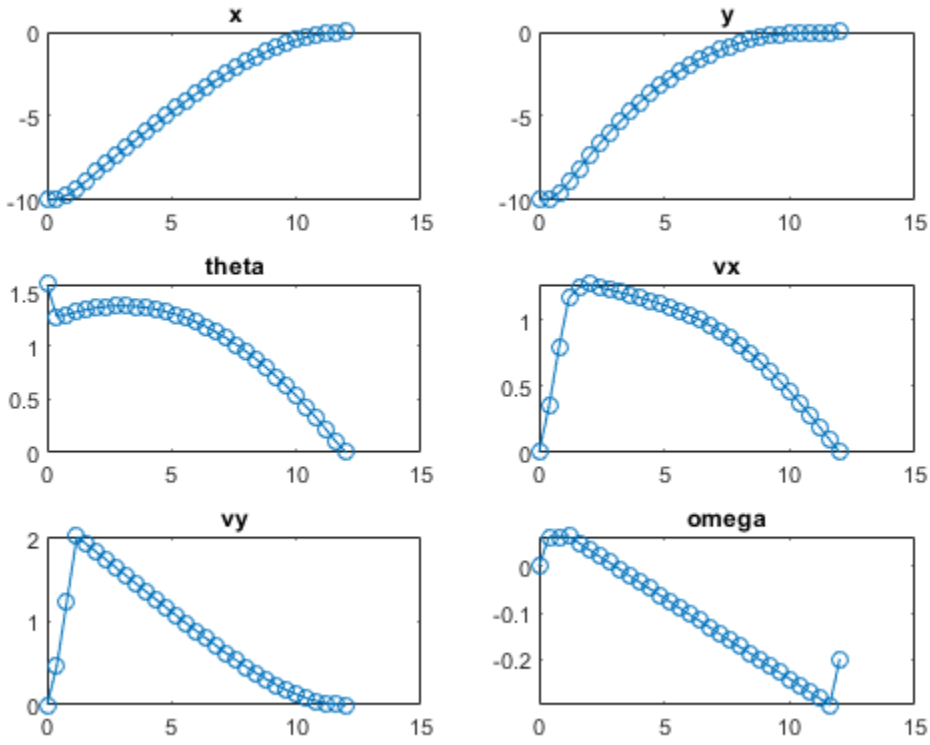
```
[~,~,info] = nlmpcmove(nlobj,x0,u0);
```

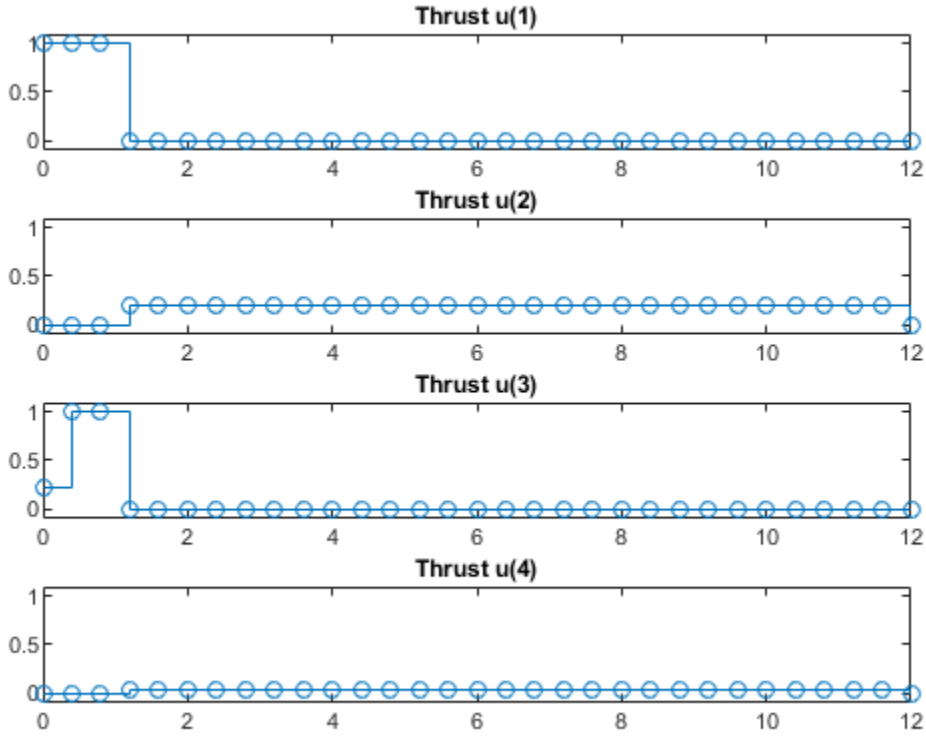
```
Slack variable unused or zero-weighted in your custom cost function. All constraints wi
```

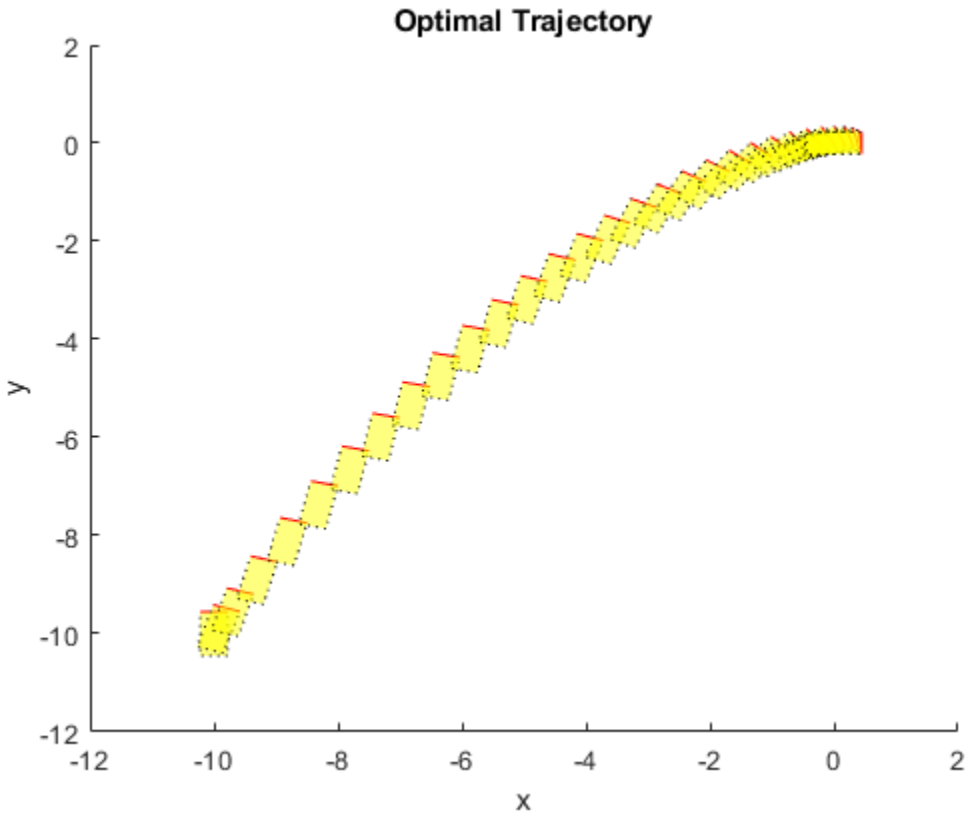
Plot the optimal trajectories.

```
FlyingRobotPlotPlanning(info)
```

```
Optimal fuel consumption =   4.712383
```




Thrust u(1)
Thrust u(2)
Thrust u(3)
Thrust u(4)
0
0.5
1
0
2
4
6
8
10
12



Simulate Closed-Loop Control using Nonlinear MPC Controller

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nlobj = nlmpc(4,2,1);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
```

```
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The prediction model uses an optional parameter, `Ts`, to represent the sample time. Specify the number of parameters.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function of the model, passing the sample time parameter as an input argument.

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Define standard constraints for the controller.

```
nlobj.Weights.OutputVariables = [3 3];
nlobj.Weights.ManipulatedVariablesRate = 0.1;
nlobj.OV(1).Min = -10;
nlobj.OV(1).Max = 10;
nlobj.MV.Min = -100;
nlobj.MV.Max = 100;
```

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Klman filter for estimingthe four plant states. Its state transition function is defined in`pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero intiial maipulated variable value..

```
x = [0;0;-pi;0];
y = [x(1);x(3)];
EKF.State = x;
mv = 0;
```

Specify the output reference value.

```
yref = [0 0];
```

Create an `nlmpcmoveopt` object, and specify the sample time parameter.

```
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};
```

Run the simulation for `10` seconds. During each cotnrol interval:

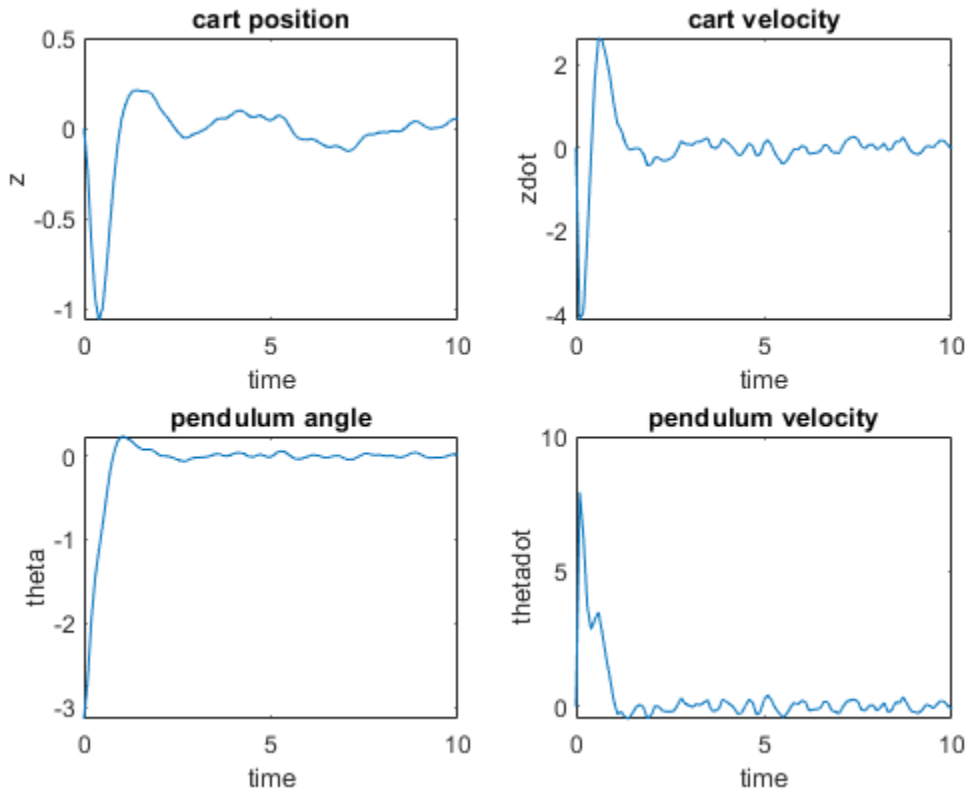
1. Correct the previous prediction using the current measurement.
2. Compute optimal control moves using `nlmpcmoveopt`. This function returns the computed optimal sequences in `nloptions`. Passing the updated options object to `nlmpcmoveopt` in the next control interval provides initial guesses for the optimal sequences.
3. Predict the model states.
4. Apply the first computed optimal control move to the plant, updating the plant states.
5. Generate sensor data with white noise.
6. Save the plant states.

```
Duration = 10;
xHistory = x;
for ct = 1:(Duration/Ts)
    % Correct previous prediction
    xk = correct(EKF,y);
    % Compute optimal control moves
    [mv,nloptions] = nlmpcmove(nlobj,xk,mv,yref,[],nloptions);
    % Predict prediction model states for the next iteration
    predict(EKF,[mv; Ts]);
    % Implement first optimal control move
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data
    y = x([1 3]) + randn(2,1)*0.01;
```

```
    % Save plant states
    xHistory = [xHistory x];
end
```

Plot the resulting state trajectories.

```
figure
subplot(2,2,1)
plot(0:Ts:Duration,xHistory(1,:))
xlabel('time')
ylabel('z')
title('cart position')
subplot(2,2,2)
plot(0:Ts:Duration,xHistory(2,:))
xlabel('time')
ylabel('zdot')
title('cart velocity')
subplot(2,2,3)
plot(0:Ts:Duration,xHistory(3,:))
xlabel('time')
ylabel('theta')
title('pendulum angle')
subplot(2,2,4)
plot(0:Ts:Duration,xHistory(4,:))
xlabel('time')
ylabel('thetadot')
title('pendulum velocity')
```



See Also

Blocks
Nonlinear MPC Controller

Topics

“Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC”
“Nonlinear Model Predictive Control of an Exothermic Chemical Reactor”
“Swing-up Control of a Pendulum Using Nonlinear Model Predictive Control”
“Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant”

“Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver”
“Nonlinear MPC”

Introduced in R2018b

nlmpcmove

Compute optimal control action for nonlinear MPC controller

Syntax

```
mv = nlmpcmove(nlmpcobj,x,lastmv)
mv = nlmpcmove(nlmpcobj,x,lastmv,ref)
mv = nlmpcmove(nlmpcobj,x,lastmv,ref,md)
mv = nlmpcmove(nlmpcobj,x,lastmv,ref,md,options)
[mv,opt] = nlmpcmove( ___ )
[mv,opt,info] = nlmpcmove( ___ )
```

Description

`mv = nlmpcmove(nlmpcobj,x,lastmv)` computes the optimal manipulated variable control action for the current time. To simulate closed-loop nonlinear MPC control, call `nlmpcmove` repeatedly.

`mv = nlmpcmove(nlmpcobj,x,lastmv,ref)` specifies reference values for the plant outputs. If you do not specify reference values, `nlmpcmove` uses zeros by default.

`mv = nlmpcmove(nlmpcobj,x,lastmv,ref,md)` specifies run-time measured disturbance values. If your controller has measured disturbances, you must specify `md`.

`mv = nlmpcmove(nlmpcobj,x,lastmv,ref,md,options)` specifies additional run-time options for computing optimal control moves. Using `options`, you can specify initial guesses for state and manipulated variable trajectories, update tuning weights at constraints, or modify prediction model parameters.

`[mv,opt] = nlmpcmove( ___ )` returns an `nlmpcmoveopt` object that contains initial guesses for the state and manipulated trajectories to be used in the next control interval.

`[mv,opt,info] = nlmpcmove( ___ )` returns additional solution details, including the final optimization cost function value and the optimal manipulated variable, state, and output trajectories.

Examples

Plan Optimal Trajectory Using Nonlinear MPC

Create nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Specify linear constraints on the manipulated variables.

```
for ct = 1:nu
    nlobj.MV(ct).Min = 0;
    nlobj.MV(ct).Max = 1;
end
```

Validate the prediction model and custom functions at the intial states (`x0`) and initial inputs (`u0`) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Compute the optimal state and manipulated variable trajectories, which are returned in the `info`.

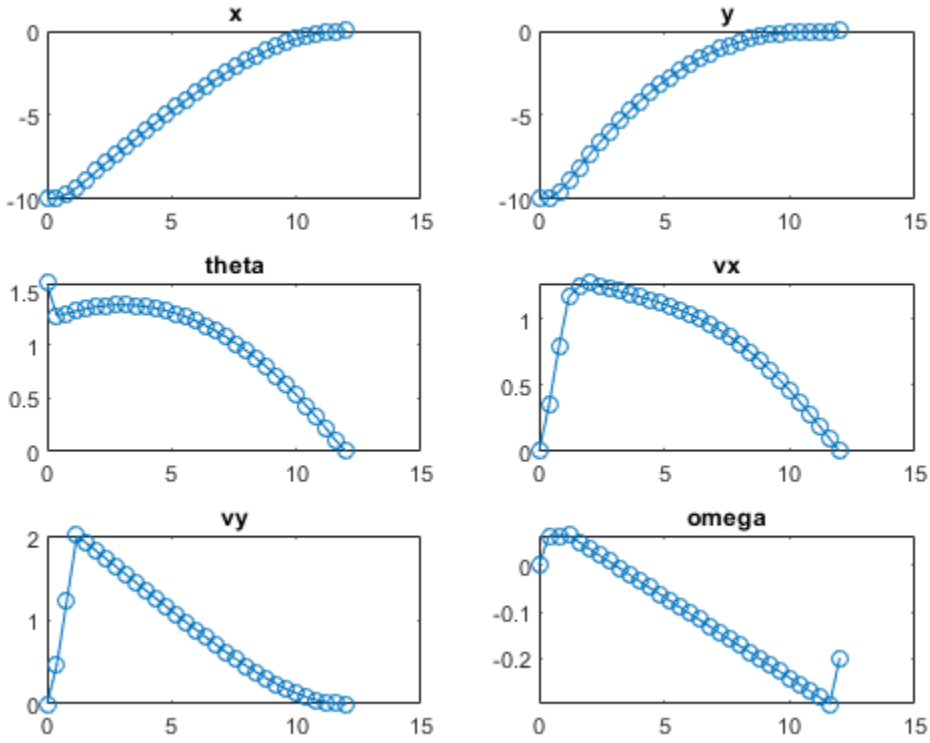
```
[~,~,info] = nlmpcmove(nlobj,x0,u0);
```

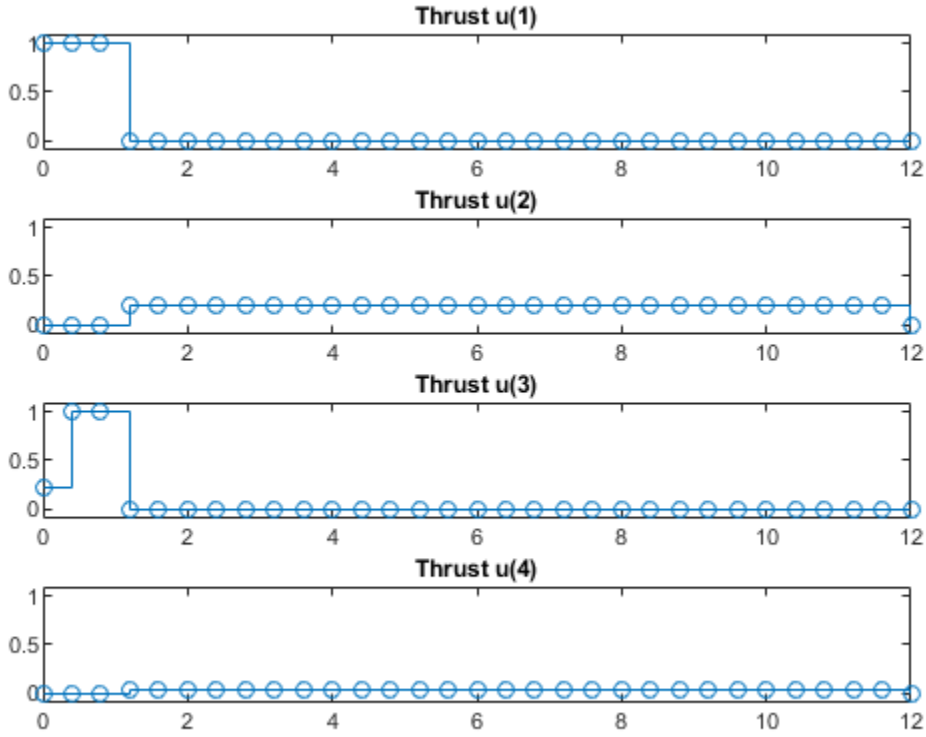
```
Slack variable unused or zero-weighted in your custom cost function. All constraints wi
```

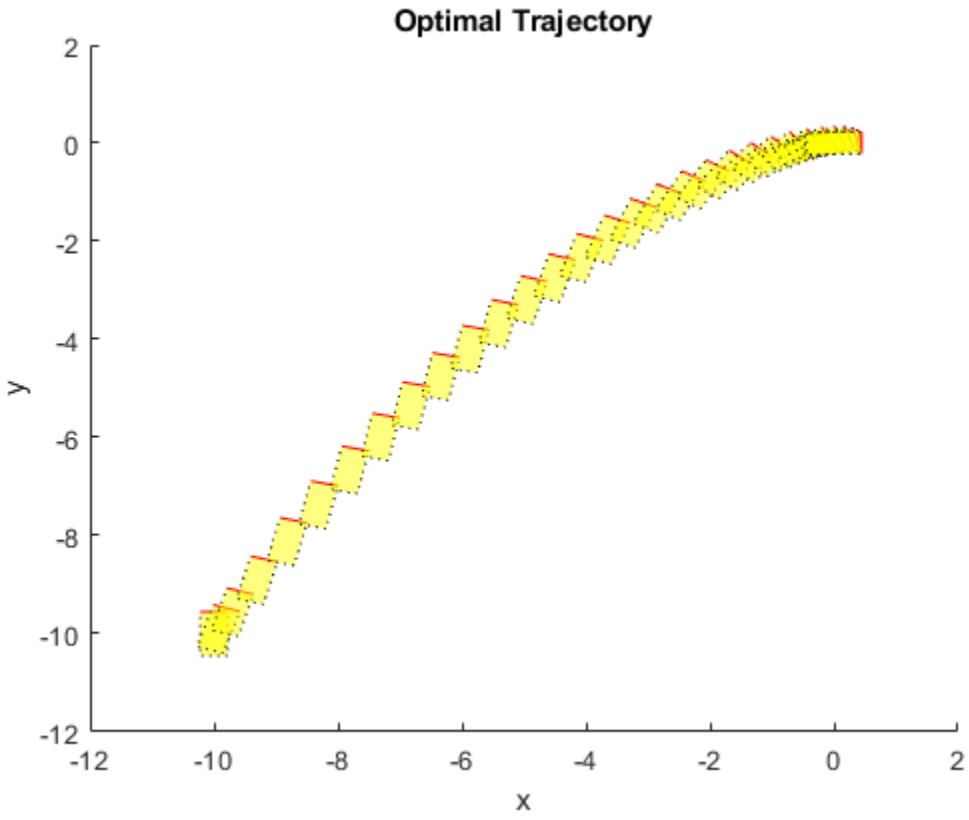
Plot the optimal trajectories.

```
FlyingRobotPlotPlanning(info)
```

```
Optimal fuel consumption =   4.712383
```


x
y
theta
vx
vy
omega


Thrust u(1)
Thrust u(2)
Thrust u(3)
Thrust u(4)
0
0.5
1
0
2
4
6
8
10
12



Simulate Closed-Loop Control using Nonlinear MPC Controller

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nlobj = nlmpc(4,2,1);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
```

```
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The prediction model uses an optional parameter, `Ts`, to represent the sample time. Specify the number of parameters.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function of the model, passing the sample time parameter as an input argument.

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Define standard constraints for the controller.

```
nlobj.Weights.OutputVariables = [3 3];
nlobj.Weights.ManipulatedVariablesRate = 0.1;
nlobj.OV(1).Min = -10;
nlobj.OV(1).Max = 10;
nlobj.MV.Min = -100;
nlobj.MV.Max = 100;
```

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Klman filter for estimingthe four plant states. Its state transition function is defined in`pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero intiial maipulated variable value..

```
x = [0;0;-pi;0];
y = [x(1);x(3)];
EKF.State = x;
mv = 0;
```

Specify the output reference value.

```
yref = [0 0];
```

Create an `nlmpcmoveopt` object, and specify the sample time parameter.

```
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};
```

Run the simulation for `10` seconds. During each cotnrol interval:

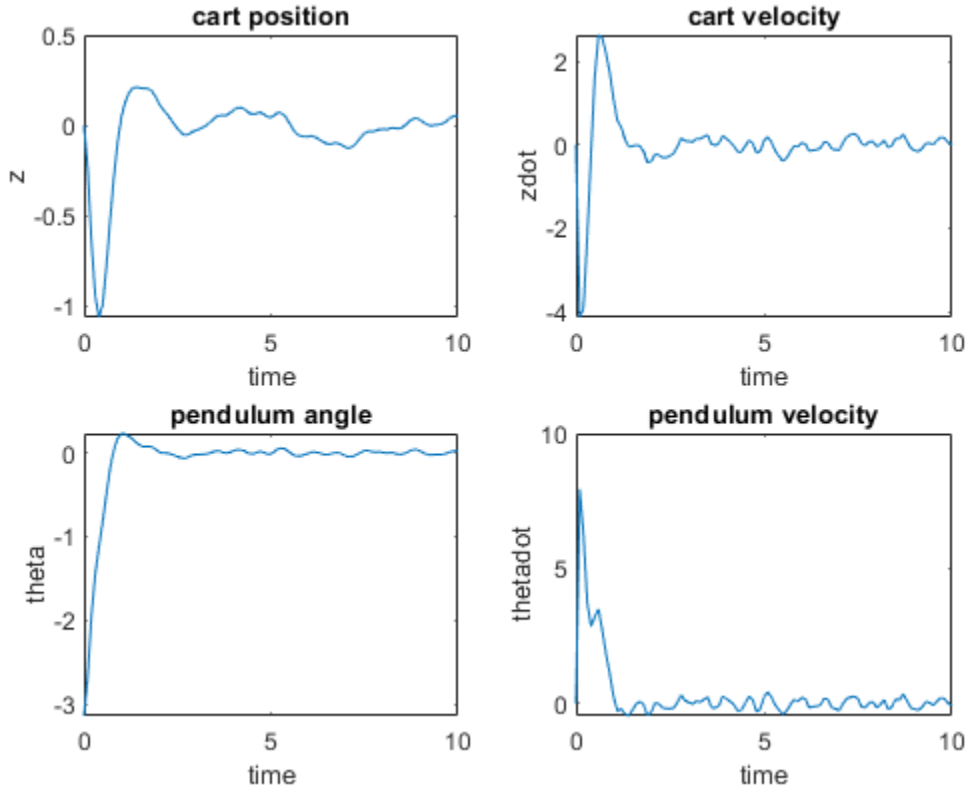
1. Correct the previous prediction using the current measurement.
2. Compute optimal control moves using `nlmpcmoveopt`. This function returns the computed optimal sequences in `nloptions`. Passing the updated options object to `nlmpcmoveopt` in the next control interval provides initial guesses for the optimal sequences.
3. Predict the model states.
4. Apply the first computed optimal control move to the plant, updating the plant states.
5. Generate sensor data with white noise.
6. Save the plant states.

```
Duration = 10;
xHistory = x;
for ct = 1:(Duration/Ts)
    % Correct previous prediction
    xk = correct(EKF,y);
    % Compute optimal control moves
    [mv,nloptions] = nlmpcmove(nlobj,xk,mv,yref,[],nloptions);
    % Predict prediction model states for the next iteration
    predict(EKF,[mv; Ts]);
    % Implement first optimal control move
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data
    y = x([1 3]) + randn(2,1)*0.01;
```

```
    % Save plant states
    xHistory = [xHistory x];
end
```

Plot the resulting state trajectories.

```
figure
subplot(2,2,1)
plot(0:Ts:Duration,xHistory(1,:))
xlabel('time')
ylabel('z')
title('cart position')
subplot(2,2,2)
plot(0:Ts:Duration,xHistory(2,:))
xlabel('time')
ylabel('zdot')
title('cart velocity')
subplot(2,2,3)
plot(0:Ts:Duration,xHistory(3,:))
xlabel('time')
ylabel('theta')
title('pendulum angle')
subplot(2,2,4)
plot(0:Ts:Duration,xHistory(4,:))
xlabel('time')
ylabel('thetadot')
title('pendulum velocity')
```



Input Arguments

`nlmpcobj` — Nonlinear MPC controller
`nlmpc` object

Nonlinear MPC controller, specified as an `nlmpc` object.

x — Current prediction model states
vector

Current prediction model states, specified as a vector of lengthN_x, where N_x is the number of prediction model states. Since the nonlinear MPC controller does not perform

state estimation, you must either measure or estimate the current prediction model states at each control interval.

`lastmv` — Control signals used in plant at previous control interval

vector

Control signals used in plant at previous control interval, specified as a vector of lengthN_{mv}, where N_{mv} is the number of manipulated variables.

Note Specify `last_mv` as the MV signals applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode; that is, the controller output is not driving the plant, then feeding the actual control signal to `last_mv` can help achieve bumpless transfer when the controller is switched back online.

`ref` — Plant output reference values

`[]` (default) | row vector | array

Plant output reference values, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. If you do not specify `ref`, the default reference values are zero.

To use the same reference values across the prediction horizon, specify a row vector.

To vary the reference values over the prediction horizon from time $k+1$ to time $k+p$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the reference values for one prediction horizon step. If you specify fewer than p rows, the values in the final row are used for the remaining steps of the prediction horizon.

`md` — Measured disturbance values

`[]` (default) | row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your controller has measured disturbances, you must specify `md`. If your controller has no measured disturbances, specify `md` as `[]`.

To use the same disturbance values across the prediction horizon, specify a row vector.

To vary the disturbance values over the prediction horizon from time k to time $k+p$, specify an array with up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbance values for one prediction horizon step. If you specify fewer than p rows, the values in the final row are used for the remaining steps of the prediction horizon.

`options` — Run-time options
`nlmpcmoveopt` object

Run-time options, specified as an `nlmpcmoveopt` object. Using these options, you can:

- Tune controller weights
- Update linear constraints
- Set manipulated variable targets
- Specify prediction model parameters
- Provide initial guesses for state and manipulated variable trajectories

These options apply to only the current `nlmpcmove` time instant.

To improve solver efficiency, it is best practice to specify initial guesses for the state and manipulated variable trajectories.

Output Arguments

`mv` — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the solver converges to a local optimum solution (`info.ExitFlag` is positive), then `mv` contains the optimal solution.

If the solver reaches the maximum number of iterations without finding an optimal solution (`info.ExitFlag = 0`) and:

- `nlmpcobj.Optimization.UseSuboptimalSolution` is `true`, then `mv` contains the suboptimal solution

- `nlmpcobj.Optimization.UseSuboptimalSolution` is `false`, then `mv` contains `lastmv`.

If the solver fails (`info.ExitFlag` is negative), then `mv` contains `lastmv`.

`opt` — Run-time options with initial guesses
`nlmpcmoveopt` object

Run-time options with initial guesses for the state and manipulated variable trajectories to be used in the next control interval, returned as an `nlmpcmoveopt` object. Any run-time options that you specified using `options`, such as weights, constraints, or parameters, are copied to `opt`.

The initial guesses for the states (`opt.X0`) and manipulated variables (`opt.MV0`) are the optimal trajectories computed by `nlmpcmove` and correspond to the last p-1 rows of `info.Xopt` and `info.MVopt`, respectively.

To use these initial guesses in the next control interval, specify `opt` as the `options` input argument to `nlmpcmove`.

`info` — Solution details
structure

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

`MVopt` — Optimal manipulated variable sequence
array

Optimal manipulated variable sequence, returned as a (p+1)-by-N_{mv} array, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

`MVopt(i,:)` contains the calculated optimal manipulated variable values at time `k+i-1`, for `i = 1,...,p`, where `k` is the current time. `MVopt(1,:)` contains the same manipulated variable values as output argument `mv`. Since the controller does not calculate optimal control moves at time `k+p`, `MVopt(p+1,:)` is equal to `MVopt(p,:)`.

`Xopt` — Optimal prediction model state sequence
array

Optimal prediction model state sequence, returned as a (p+1)-by-N_x array, where p is the prediction horizon and N_x is the number of states in the prediction model.

`Xopt(i,:)` contains the calculated state values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Xopt(1,:)` is the same as the current states in `x`.

`Yopt` — Optimal output variable sequence
array

Optimal output variable sequence, returned as a (p+1)-by-N_y array, where p is the prediction horizon and N_y is the number of outputs.

`Yopt(i,:)` contains the calculated output values at time `k+i-1`, for `i = 2,...,p+1`, where `k` is the current time. `Yopt(1,:)` is computed based on the current states in `x` and the current measured disturbances in `md`, if any.

`Topt` — Prediction horizon time sequence
column vector

Prediction horizon time sequence, returned as a column vector of length p+1, p is the prediction horizon. `Topt` contains the time sequence from time k to time k+p, where k is the current time.

`Topt(1)` = 0 represents the current time. Subsequent time steps `Topt(i)` are `Ts*(i-1)`, where `Ts = nlmpcobj.Ts` is the controller sample time.

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

`Slack` — Slack variable
nonnegative scalar

Slack variable, ε, used in constraint softening, returned as a nonnegative scalar value.

- ε = 0 — All soft constraints are satisfied over 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).

`ExitFlag` — Optimization exit code
integer

Optimization exit code, returned as one of the following:

- Positive Integer — Solver converged to an optimal solution

- `0` — Maximum number of iterations reached without converging to an optimal solution
- Negative integer — Solver failed

`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.

The cost value is only meaningful when `ExitFlag` is nonnegative.

Tips

During closed-loop simulations, it is best practice to *warm start* the nonlinear solver by using the predicted state and manipulated variable trajectories from the previous control interval as the initial guesses for the current control interval. To use these trajectories as initial guesses:

1. Return the `opt` output argument when calling `nlmpcmove`. This `nlmpcmoveopt` object contains any run-time options you specified in the previous call to `nlmpcmove` along with the initial guesses for the state (`opt.X0`) and manipulated variable (`opt.MV0`) trajectories.
2. Pass this object in as the `options` input argument to `nlmpcmove` for the next control interval.

These steps are the recommended best practice, even if you do not specify any other run-time options.

See Also

`nlmpc` | `nlmpcmoveopt`

Topics

“Nonlinear MPC”
“Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC”

Introduced in R2018b

nlmpcmoveopt

Option set for `nlmpcmove` function

Description

To specify options for the `nlmpcmove` function, use an `nlmpcmoveopt` option set.

Using this option set, you can specify run-time values for a subset of controller properties, such as tuning weights and constraints. If you do not specify a value for one of the `nlmpcmoveopt` properties, the corresponding value defined in the `nlmpc` controller object is used instead.

Creation

Syntax

```
options = nlmpcmoveopt
```

Description

`options = nlmpcmoveopt` creates a default set of options for the `nlmpcmove` function. To modify the property values, use dot notation.

Properties

`OutputWeights` — Output variable tuning weights
[ ] (default) | row vector | array

Output variable tuning weights that replace the `Weights.OutputVariables` property of the controller at run time, specified as a row vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`MVWeights` — Manipulated variable tuning weights
[ ] (default) | row vector | array

Manipulated variable tuning weights that replace the `Weights.ManipulatedVariables` property of the controller at run time, specified as a row vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`MVRateWeights` — Manipulated variable rate tuning weights
[ ] (default) | row vector | array

Manipulated variable rate tuning weights that replace the `Weights.ManipulatedVariablesRate` property of the controller at run time, specified as a row vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`ECR Weight` — Slack variable tuning weight
[ ] (default) | positive scalar

Slack variable tuning weight that replaces the `Weights.ECR` property of the controller at run time, specified as a positive scalar.

`OutputMin` — Output variable lower bounds
[ ] (default) | row vector | array

Output variable lower bounds, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. `OutputMin(:,i)` replaces the `OutputVariables(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k+1 to time k+p, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`OutputMax` — Output variable upper bounds
[ ] (default) | row vector | array

Output variable upper bounds, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. `OutputMax(:,i)` replaces the `OutputVariables(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k+1 to time k+p, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`MVMin` — Manipulated variable lower bounds
[ ] (default) | row vector | array

Manipulated variable lower bounds, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVMin(:,i)` replaces the `ManipulatedVariables(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time k+p-1, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each

row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`MVMax` — Manipulated variable upper bounds
`[ ]` (default) | row vector | array

Manipulated variable upper bounds, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVMax(:,i)` replaces the `ManipulatedVariables(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`MVRateMin` — Manipulated variable rate lower bounds
`[ ]` (default) | row vector | array

Manipulated variable rate lower bounds, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMin(:,i)` replaces the `ManipulatedVariables(i).RateMin` property of the controller at run time. `MVRateMin` bounds must be nonpositive.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`MVRateMax` — Manipulated variable rate upper bounds
`[ ]` (default) | row vector | array

Manipulated variable rate upper bounds, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMax(:,i)` replaces the `ManipulatedVariables(i).RateMax` property of the controller at run time. `MVRateMax` bounds must be nonnegative.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`StateMin` — State lower bounds

[ ] (default) | row vector | array

State lower bounds, specified as a row vector of length N_x or an array with N_x columns, where N_x is the number of states. `StateMin(:,i)` replaces the `States(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`StateMax` — State upper bounds

[ ] (default) | row vector | array

State upper bounds, specified as a row vector of length N_x or an array with N_x columns, where N_x is the number of states. `StateMax(:,i)` replaces the `States(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

`MVTarget` — Manipulated variable targets

[ ] (default) | row vector | array

Manipulated variable targets, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables.

To use the same manipulated variable targets across the prediction horizon, specify a row vector.

To vary the targets over the prediction horizon (previewing) from time k to time $k+p-1$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the targets for one prediction horizon step. If you specify fewer than p rows, the final targets are used for the remaining steps of the prediction horizon.

`Parameters` — Parameter values
`{}` (default) | cell vector

Parameter values used by the prediction model, custom cost function, and custom constraints, specified as a cell vector with length equal to the `Model.NumberOfParameters` property of the controller. If the controller has no parameters, then `Parameters` must be `{}`.

The controller, `nlmpcobj`, passes these parameters to the:

- Model functions in `nlmpcobj.Model` (`StateFcn` and `OutputFcn`)
- Cost function `nlmpcobj.Optimization.CustomCostFcn`
- Constraint functions in `nlmpcobj.Optimization` (`CustomEqConFcn` and `CustomIneqConFcn`)
- Jacobian functions in `nlmpcobj.Jacobian`

The order of the parameters must match the order defined for these functions.

`X0` — Initial guesses for the optimal state solutions
`[]` (default) | vector | array

Initial guesses for the optimal state solutions, specified as a row vector of length N_x or an array with N_x columns, where N_x is the number of states.

To use the same initial guesses across the prediction horizon, specify a row vector.

To vary the initial guesses over the prediction horizon from time $k+1$ to time $k+p$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the final guesses are used for the remaining steps of the prediction horizon.

If `X0` is `[]`, the default initial guesses are the current states of the prediction model (`x` input argument to `nlmpcmove`).

In general, during closed-loop simulation, you do not specify `X0` yourself. Instead, when calling `nlmpcmove`, return the `opt` output argument, which is an `nlmpcmoveopt` object.

`opt.X0` contains the calculated optimal state trajectories as initial guesses. You can then pass `opt` in as the `options` input argument to `nlmpcmove` for the next control interval. These steps are the recommended best practice, even if you do not specify any other run-time options.

`MV0` — Initial guesses for the optimal manipulated variable solutions

`[]` (default) | vector | array

Initial guesses for the optimal manipulated variable solutions, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables.

To use the same initial guesses across the prediction horizon, specify a row vector.

To vary the initial guesses over the prediction horizon from time k to time $k+p-1$, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the final guesses are used for the remaining steps of the prediction horizon.

If `MV0` is `[]`, the default initial guesses are the control signals used in the plant at the previous control interval (`lastmv` input argument to `nlmpcmove`).

In general, during closed-loop simulation, you do not specify `MV0` yourself. Instead, when calling `nlmpcmove`, return the `opt` output argument, which is an `nlmpcmoveopt` object. `opt.MV0` contains the calculated optimal manipulated variable trajectories as initial guesses. You can then pass `opt` in as the `options` input argument to `nlmpcmove` for the next control interval. These steps are the recommended best practice, even if you do not specify any other run-time options.

`Slack0` — Initial guess for the slack variable at the solution

`[]` (default) | nonnegative scalar

Initial guess for the slack variable at the solution, specified as a nonnegative scalar. If `Slack0` is `[]`, the default initial guess is `0`.

In general, during closed-loop simulation, you do not specify `Slack0` yourself. Instead, when calling `nlmpcmove`, return the `opt` output argument, which is an `nlmpcmoveopt` object. `opt.Slack` contains the calculated slack variable as an initial guess. You can then pass `opt` in as the `options` input argument to `nlmpcmove` for the next control interval. These steps are the recommended best practice, even if you do not specify any other run-time options.

Object Functions

nlmpcmove Compute optimal control action for nonlinear MPC controller

Examples

Specify Run-Time Parameters for Nonlinear MPC

Create a default `nlmpcmoveopt` option set.

```
options = nlmpcmoveopt;
```

Specify the run-time values for the controller prediction model parameters. For this example, assume that the controller has the following optional parameters, which are input arguments to all of the prediction model functions and custom functions of the controller.

- Sample time of the model, specified as a single numeric value. Specify a value of `0.25`.
- Gain factors, specified as a two-element row vector. Specify a value of `[0.7 0.35]`.

The order in which you specify the parameters must match the order specified in the custom function argument lists. Also, the dimensions of the parameters must match the dimensions expected by the custom functions.

```
options.Parameters = {0.25,[0.7 0.35]};
```

To use these parameters when computing optional control actions for a nonlinear MPC controller, pass `options` to the `nlmpcmove` function.

See Also

`nlmpc`

Topics

“Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC”

Introduced in R2018b

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

`mpc` | `sim`

Introduced before R2006a

plotSection

Visualize explicit MPC control law as 2-D sectional plot

Syntax

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

Description

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

Examples

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

Define a double integrator plant model and create a traditional implicit MPC controller for this plant. Constrain the manipulated variable to have an absolute value less than 1.

```
plant = tf(1,[1 0 0]);
MPCobj = mpc(plant,0.1,10,3);
```

```
-->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
```

```
MPCobj.MV = struct('Min',-1,'Max',1);
```

Define the parameter bounds for generating an explicit MPC controller.

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

```
-->Converting the "Model.Plant" property of "mpc" object to state-space.
-->Converting model to discrete time.
```

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

```
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;
```

Create an explicit MPC controller.

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

```
Regions found / unexplored:       19/       0
```

Create a default plot parameter structure, which specifies that all of the controller parameters are fixed at their nominal values for plotting.

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

Allow the controller states to vary when creating a plot.

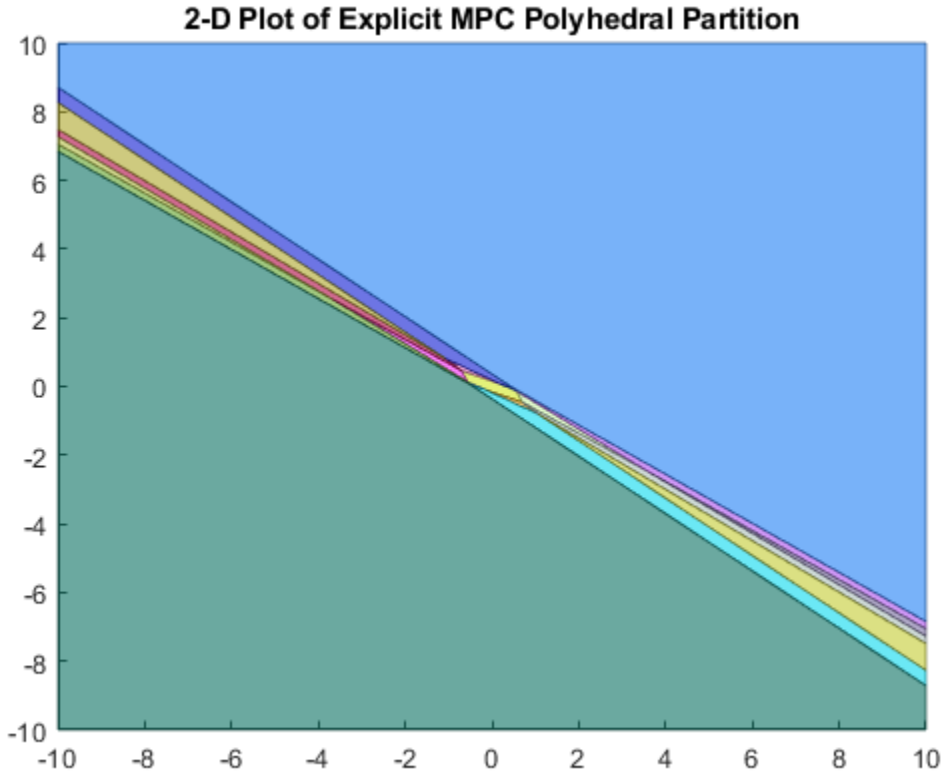
```
plotParams.State.Index = [];
plotParams.State.Value = [];
```

Fix the manipulated variable and reference signal to `0` for plotting.

```
plotParams.ManipulatedVariable.Index(1) = 1;
plotParams.ManipulatedVariable.Value(1) = 0;
plotParams.Reference.Index(1) = 1;
plotParams.Reference.Value(1) = 0;
```

Generate the 2-D section plot for the explicit MPC controller.

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




```
ans =
  Figure (1: PiecewiseAffineSectionPlot) with properties:

      Number: 1
        Name: 'PiecewiseAffineSectionPlot'
       Color: [0.9400 0.9400 0.9400]
    Position: [360 502 560 420]
       Units: 'pixels'

  Show all properties
```

Input Arguments

`EMPCobj` — Explicit MPC controller
explicit MPC controller object

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

`plotParams` — Parameters for sectional plot
structure

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

See Also

`generateExplicitMPC` | `generatePlotParameters`

Introduced in R2014b

review

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

Syntax

```
review(mpcobj)

results = review(mpcobj)
```

Description

`review(mpcobj)` checks for potential design issues in the model predictive controller, `mpcobj`, and generates a testing report. The testing report provides information about each test, highlights test warnings and failures, and suggests possible solutions. For more information on the tests performed by the `review` function, see “Algorithms” on page 2-236.

`results = review(mpcobj)` returns the test results and suppresses the testing report.

Examples

Examine MPC Controller for Design Errors or Stability Problems

Define a plant model, and create an MPC controller.

```
plant = tf(1, [10 1]);
Ts = 2;
MPCobj = mpc(plant,Ts);
```

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

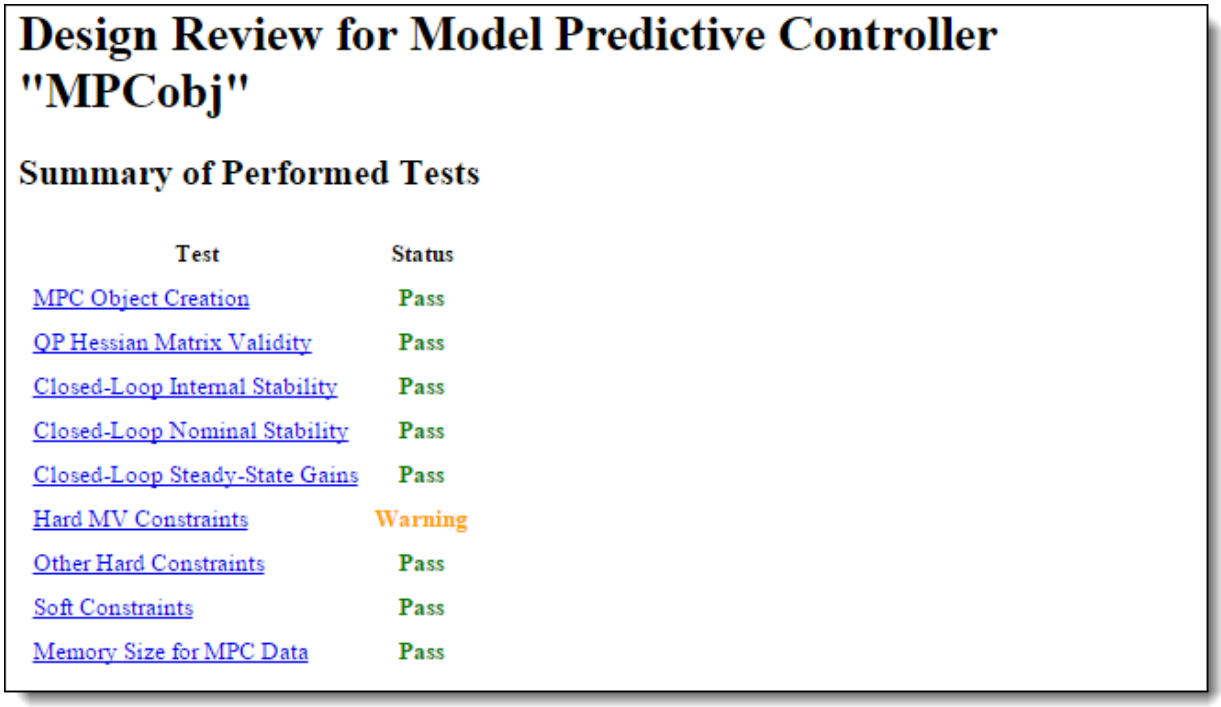
Set hard upper and lower bounds on the manipulated variable and its rate of change.

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

Review the controller design. The review function generates and opens a report in the Web Browser window.

```
review(MPCobj)
```

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

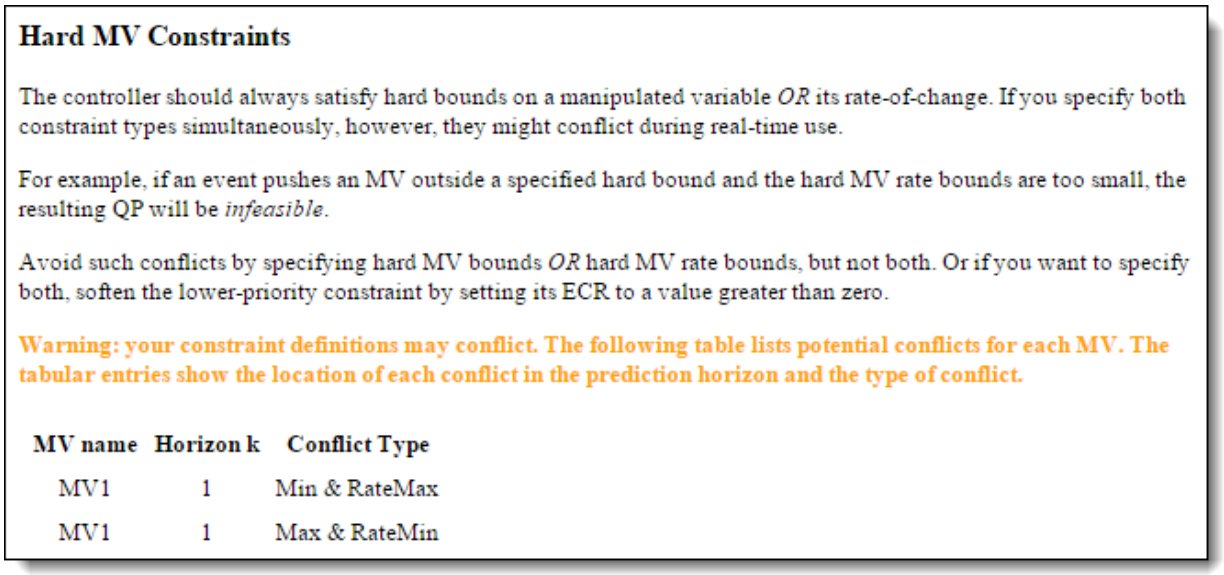


Design Review for Model Predictive Controller "MPCobj"

Summary of Performed Tests

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

`review` flags a potential constraint conflict that could result if this controller was used to control a real process. To view details about the warning, click **Hard MV Constraints**.



Hard MV Constraints

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

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

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

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

MV name	Horizon k	Conflict Type
MV1	1	Min & RateMax
MV1	1	Max & RateMin

Obtain Test Results and Suppress Testing Report

Define a plant model and create an MPC controller.

```
plant = rss(3,1,1);
plant.D = 0;
Ts = 0.1;
MPCobj = mpc(plant,Ts);
```

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

Specify contraints for the controller.

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

Review the controller design and suppress the testing report.

```
results = review(MPCobj)
```

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

results = struct with fields:
          ObjectCreation: 1
           HessianMatrix: 1
       InternalStability: 1
        NominalStability: 1
             SteadyState: 1
       HardMVConstraints: 0
    HardOtherConstraints: 1
         SoftConstraints: 1
```

All of the tests passed, except for the the hard MV constraints test, which generated a warning.

Obtain Test Results for Multiple Controllers

Create and review designs for gain-scheduled model predictive controllers for two plant operating conditions.

Define the model parameters.

```
M1 = 1;
M2 = 5;
k1 = 1;
k2 = 0.1;
b1 = 0.3;
b2 = 0.8;
```

```
yeq1 = 10;
yeq2 = -10;
```

Create plant models for each of the two operating conditions.

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

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

Design an MPC controller for each operating condition.

```
Ts = 0.2;
p = 6;
m = 2;
MPC1 = mpc(sys1,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.
```

```
MPC2 = mpc(sys2,Ts,p,m);
```

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

```
controllers = {MPC1,MPC2};
```

Review the controller designs, and store the test result structures.

```
for i = 1:2
    results(i) = review(controllers{i});
end
```

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

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

Input Arguments

`mpcobj` — MPC controller

`mpc` object

MPC controller object, specified as an `mpc` object.

Output Arguments

`results` — Test results

structure

Test results, returned as a structure with the following fields:

- `ObjectCreation` — MPC object creation test
- `HessianMatrix` — QP Hessian matrix validity test
- `InternalStability` — Internal stability test
- `NominalStability` — Nominal stability test
- `SteadyState` — Closed-loop steady-state gains test
- `HardMVConstraints` — Hard MV constraints test
- `HardOtherConstraints` — Other hard constraints test
- `SoftConstraints` — Soft constraints test

For more information on the tests performed by the `review` function, see “Algorithms” on page 2-236.

The `results` structure does not contain a field for the **Memory Size for MPC Data** test.

For each test, the result is returned as one of the following:

- 1 — Pass

- `0` — Warning
- `-1` — Fail

If a given test generates a warning or fails, generate a testing report by calling `review` without an output argument. The testing report provides details about the warnings and failures, and suggests possible solutions.

Tips

- You can also review your controller design in the **MPC Designer** app. On the **Tuning** tab, in the **Analysis** section, click **Review Design**.
- Test your controller design using techniques such as simulations, since `review` cannot detect all possible performance factors.

Algorithms

The review command performs the following tests.

Test	Description
MPC Object Creation	Test whether the controller specifications generate a valid MPC controller. If the controller is invalid, additional tests are not performed.
QP Hessian Matrix Validity	Test whether the MPC quadratic programming (QP) problem for the controller has a unique solution. You must choose cost function parameters (penalty weights) and horizons such that the QP Hessian matrix is positive-definite.
Closed-Loop Internal Stability	Extract the `A` matrix from the state-space realization of the unconstrained controller, and then calculate its eigenvalues. If the absolute value of each eigenvalue is less than or equal to `1` and the plant is stable, then your feedback system is internally stable.

Test	Description
Closed-Loop Nominal Stability	Extract the `A` matrix from the discrete-time state-space realization of the closed-loop system; that is, the plant and controller connected in a feedback configuration. Then calculate the eigenvalues of `A`. If the absolute value of each eigenvalue is less than or equal to `1`, then the nominal (unconstrained) system is stable.
Closed-Loop Steady-State Gains	Test whether the controller forces all controlled output variables to their targets at steady state in the absence of constraints.
Hard MV Constraints	Test whether the controller has hard constraints on both a manipulated variable and its rate of change, and if so, whether these constraints may conflict at run time.
Other Hard Constraints	Test whether the controller has hard output constraints or hard mixed input/output constraints, and if so, whether these constraints may become impossible to satisfy at run time.
Soft Constraints	Test whether the controller has the proper balance of hard and soft constraints by evaluating the constraint ECR parameters.
Memory Size for MPC Data	Estimate the memory size required by the controller at run time.

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. Convert the `mpc` object to a state-space model using `ss`, and call `isstable`.

See Also

`cloffset` | `mpc` | `ss`

Topics

“Simulation and Code Generation Using Simulink Coder”
“Review Model Predictive Controller for Stability and Robustness Issues”

Introduced in R2011b

sensitivity

Compute effect of controller tuning weights on performance

Syntax

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

Description

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

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

`PerfFunc` must be one of the following:

`'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

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

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

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

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` — n_y element row vector that contains the w_j^y values
- `ManipulatedVariables` — n_u element row vector that contains the w_j^u values
- `ManipulatedVariablesRate` — n_u element row vector that contains the $w_j^{\Delta u}$ values

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

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

`Tstop` is the integer number of controller sampling intervals to be simulated. The final time for the simulations will be *Tstop* × Δ*t*, where Δ*t* is the controller sampling interval specified in `MPCobj`.

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

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

- `OutputVariables` — n_y element row vector of sensitivities with respect to `MPCobj.Weights.OutputVariables`
- `ManipulatedVariables` — n_u element row vector of sensitivities with respect to `MPCobj.Weights.ManipulatedVariables`
- `ManipulatedVariablesRate` — n_u element row vector of sensitivities with respect to `MPCobj.Weights.ManipulatedVariablesRate`

See “Weights” on page 2-77 for details on the tuning weights contained in `MPCobj`.

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

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

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

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

Examples

Compute Controller Performance and Sensitivity

Define a third-order plant model with three manipulated variables and two controlled outputs.

```
plant = rss(3,2,3);
plant.D = 0;
```

Create an MPC controller for the plant.

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

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

Specify an integral absolute error performance function and set the performance weights.

```
PerfFunc = 'IAE';
PerfWts.OutputVariables = [1 0.5];
PerfWts.ManipulatedVariables = zeros(1,3);
PerfWts.ManipulatedVariablesRate = zeros(1,3);
```

Define a `20` second simulation scenario with a unit step in the output 1 setpoint and a setpoint of zero for output 2.

```
Tstop = 20;
r = [1 0];
```

Define the nominal values of the manipulated variables to be zeros.

```
utarget = zeros(1,3);
```

Calculate the performance metric, `J`, and sensitivities, `sens`, for the specified controller and simulation scenario.

```
[J,sens] = sensitivity(MPCobj,PerfFunc,PerfWts,Tstop,r,[],[],utarget);

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

See Also

`mpc` | `sim`

Introduced in R2009a

set

Set or modify MPC object properties

Syntax

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

Description

The `set` function is used to set or modify the properties of an MPC controller (see “MPC Controller Object” on page 4-2 for background on MPC properties). `set` uses property Name,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 character vector or string `Property`. `Property` 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,Value pair updates one particular property.

`set(MPCobj,Property)` displays admissible values for the property specified by the character vector `Property`. See “MPC Controller Object” on page 4-2 for an overview of legitimate MPC property values.

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

See Also

`get` | `mpc` | `mpcprops`

Introduced before R2006a

setconstraint

Set mixed input/output constraints for model predictive controller

Syntax

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

setconstraint(MPCobj)
```

Description

`setconstraint(MPCobj,E,F,G)` specifies mixed input/output constraints of the following form for the MPC controller, `MPCobj`:

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

where $j = 0, \ldots, p$, and:

- p is the prediction horizon.
- k is the current time index.
- E, F, and G are constant matrices. Each row of E, F, and G represents a linear constraint to be imposed at each prediction horizon step.
- u is a column vector of manipulated variables.
- y is a column vector of all plant output variables.
- ε is a slack variable used for constraint softening (as in “Standard Cost Function”).

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

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

where `V` is a constant vector representing the equal concern for the relaxation (ECR).

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

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

where:

- v is a column vector of measured disturbance variables.
- S is a constant matrix.

`setconstraint(MPCobj)` removes all mixed input/output constraints from the MPC controller, `MPCobj`.

Examples

Specify Custom Constraints on Linear Combination of Inputs and Outputs

Specify a constraint of the form $0 \le u_2 - 2u_3 + y_2 \le 15$ on an MPC controller.

Create a third-order plant model with three manipulated variables and two measured outputs.

```
plant = rss(3,2,3);
plant.D = 0;
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);
```

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

Formulate the constraint in the required form:

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

Specify the constraint matrices.

```
E = [0 -1 2;0 1 -2];
F = [0 -1;0 1];
G = [0;15];
```

Set the constraints in the MPC controller.

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

Specify Custom Hard Constraints for MPC Controller

Create a third-order plant model with two manipulated variables and two measured outputs.

```
plant = rss(3,2,2);
plant.D = 0;
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);
```

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

Assume that you have two hard constraints.

$$u_1 + u_2 \le 5$$

$$y_1 + y_2 \le 10$$

Specify the constraint matrices.

```
E = [1 1; 0 0];
F = [0 0; 1 1];
G = [5;10];
```

Specify the constraints as hard by setting `V` to zero for both constraints.

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

Set the constraints in the MPC controller.

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

Specify Custom Constraints for MPC Controller with Measured Disturbances

Create a third-order plant model with two manipulated variables, two measured disturbances, and two measured outputs.

```
plant = rss(3,2,4);
plant.D = 0;
plant = setmpcsignals(plant,'mv',[1 2],'md',[3 4]);
```

Create an MPC controller for this plant.

```
MPCobj = mpc(plant,0.1);
```

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

Assume that you have three soft constraints.

$$u_1 + u_2 \le 5$$
$$y_1 + v_1 \le 10$$
$$y_2 + v_2 \le 12$$

Specify the constraint matrices.

```
E = [1 1; 0 0; 0 0];
F = [0 0; 1 0; 0 1];
G = [5;10;12];
S = [0 0; 1 0; 0 1];
```

Set the constraints in the MPC controller using the default value for `V`.

```
setconstraint(MPCobj,E,F,G,[],S)
```

Remove All Custom Constraints from MPC Controller

Define a plant model and create an MPC controller.

```
plant = rss(3,2,2);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Define controller custom constraints.

```
E = [-1 2; 1 -2];
F = [0 1; 0 -1];
G = [0; 10];
setconstraint(MPCobj,E,F,G)
```

Remove the custom constraints.

```
setconstraint(MPCobj)
```

```
-->Removing mixed input/output constraints.
```

Input Arguments

`MPCobj` — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`E` — Manipulated variable constraint constant

matrix of zeros (default) | matrix

Manipulated variable constraint constant, specified as an N_c-by-N_{mv} array, where N_c is the number of constraints, and N_{mv} is the number of manipulated variables.

F — Controlled output constraint constant

matrix of zeros (default) | matrix

Controlled output constraint constant, specified as an N_c-by-N_y array, where N_y is the number of controlled outputs (measured and unmeasured).

G — Mixed input/output constraint constant

column vector of zeros (default) | column vector

Mixed input/output constraint constant, specified as a column vector of length N_c.

V — Constraint softening constant

column vector of ones (default) | column vector

Constraint softening constant representing the equal concern for the relaxation (ECR), specified as a column vector of length N_c.

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

To make the i^{th} constraint hard, specify $V(i) = 0$.

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

In general, as $V(i)$ decreases, the controller hardens the constraints by decreasing the constraint violation that is allowed.

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

S — Measured disturbance constraint constant

matrix of zeros (default) | matrix

Measured disturbance constraint constant, specified as an N_c-by-N_v array, where N_v is the number of measured disturbances.

Tips

- The outputs, y, are being predicted using a model. If the model is imperfect, there is no guarantee that a constraint can be satisfied.
- Since the MPC controller does not optimize $u(k + p|k)$, the last constraint at time $k + p$ assumes that $u(k+p|k) = u(k+p–1|k)$.
- When simulating an MPC controller, you can update the `E`, `F`, `G`, and `S` constraint arrays at run time. For more information, see “Update Constraints at Run Time”.

See Also

`getconstraint` | `setterminal`

Topics

“Using Custom Input and Output Constraints”
“Nonlinear Blending Process with Custom Constraints”
“Constraints on Linear Combinations of Inputs and Outputs”
“Update Constraints at Run Time”

Introduced in R2011a

setEstimator

Modify a model predictive controller's state estimator

Syntax

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

Description

`setEstimator(MPCobj,L,M)` sets the gain matrices used for estimation of the states of an MPC controller. See "State Estimator Equations" on page 2-255. If `L` is empty, it defaults to `L = A*M`, where `A` is the state transition matrix defined in "State Estimator Equations" on page 2-255. 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 2-255. 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 = 6×1

    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 2-255.

If `L` is empty, it defaults to `L = A*M`, where `A` is the state transition matrix defined in "State Estimator Equations" on page 2-255.

`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 2-255.

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

Definitions

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.
$[j\|k]$	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.)

See Also

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

Introduced in R2014b

setindist

Modify unmeasured input disturbance model

Syntax

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

Description

`setindist(MPCobj,'model',model)` sets the input disturbance model used by the model predictive controller, `MPCobj`, to a custom model.

`setindist(MPCobj,'integrators')` sets the input disturbance model to its default value. Use this syntax if you previously set a custom input disturbance model and you want to change back to the default model. For more information on the default input disturbance model, see “MPC Modeling”.

Examples

Specify Input Disturbance Model Using Transfer Functions

Define a plant model with no direct feedthrough.

```
plant = rss(3,4,4);
plant.D = 0;
```

Set the first input signal as a manipulated variable and the remaining inputs as input disturbances.

```
plant = setmpcsignals(plant,'MV',1,'UD',[2 3 4]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
   for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Define disturbance models such that:

- Input disturbance 1 is random white noise with a magnitude of 2.
- Input disturbance 2 is random step-like noise with a magnitude of `0.5`.
- Input disturbance 3 is random ramp-like noise with a magnitude of 1.

```
mod1 = tf(2,1);
mod2 = tf(0.5,[1 0]);
mod3 = tf(1,[1 0 0]);
```

Construct the input disturbance model using the above transfer functions. Use a separate noise input for each input disturbance.

```
indist = [mod1 0 0; 0 mod2 0; 0 0 mod3];
```

Set the input disturbance model in the MPC controller.

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

View the controller input disturbance model.

```
getindist(MPCobj)
```

```
ans =

  A =
        x1   x2   x3
   x1    1    0    0
   x2    0    1    0
   x3    0  0.1    1

  B =
       Noise#1  Noise#2  Noise#3
   x1        0     0.05        0
   x2        0        0      0.1
   x3        0        0    0.005
```

```
  C =
        x1  x2  x3
   UD1   0   0   0
   UD2   1   0   0
   UD3   0   0   1

  D =
        Noise#1  Noise#2  Noise#3
   UD1        2        0        0
   UD2        0        0        0
   UD3        0        0        0

Sample time: 0.1 seconds
Discrete-time state-space model.
```

The controller converts the continuous-time transfer function model, `indist`, into a discrete-time state-space model.

Remove Input Disturbance for Particular Channel

Define a plant model with no direct feedthrough.

```
plant = rss(3,4,4);
plant.D = 0;
```

Set the first input signal as a manipulated variable and the remaining inputs as input disturbances.

```
plant = setmpcsignals(plant,'MV',1,'UD',[2 3 4]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

```
-->The "PredictionHorizon" property of "mpc" object is empty. Trying PredictionHorizon
-->The "ControlHorizon" property of the "mpc" object is empty. Assuming 2.
-->The "Weights.ManipulatedVariables" property of "mpc" object is empty. Assuming defau
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is empty. Assuming d
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assuming default 1
   for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Retrieve the default input disturbance model from the controller.

```
distMod = getindist(MPCobj);
```

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property of "mpc" object is empty:
   Assuming unmeasured input disturbance #2 is integrated white noise.
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming unmeasured input disturbance #4 is integrated white noise.
-->Assuming output disturbance added to measured output channel #1 is integrated white
   Assuming no disturbance added to measured output channel #2.
   Assuming no disturbance added to measured output channel #3.
   Assuming no disturbance added to measured output channel #4.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Remove the integrator from the second input disturbance. Construct the new input disturbance model by removing the second input channel and setting the effect on the second output by the other two inputs to zero.

```
distMod = sminreal([distMod(1,1) distMod(1,3); 0 0; distMod(3,1) distMod(3,3)]);
setindist(MPCobj,'model',distMod)
```

When removing an integrator from the input disturbance model in this way, use `sminreal` to make the custom model structurally minimal.

View the input disturbance model.

```
tf(getindist(MPCobj))
```

```
ans =

  From input "UD1-wn" to output...
          0.1
   UD1:  -----
         z - 1

   UD2:  0

   UD3:  0

  From input "UD3-wn" to output...
   UD1:  0

   UD2:  0

          0.1
   UD3:  -----
```

```
         z - 1

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The integrator has been removed from the second channel. The first and third channels of the input disturbance model remain at their default values as discrete-time integrators.

Set Input Disturbance Model to Default Value

Define a plant model with no direct feedthrough.

```
plant = rss(2,2,3);
plant.D = 0;
```

Set the second and third input signals as input disturbances.

```
plant = setmpcsignals(plant,'MV',1,'UD',[2 3]);
```

Create an MPC controller for the defined plant.

```
MPCobj = mpc(plant,0.1);
```

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

Set the input disturbance model to unity gain for both channels.

```
setindist(MPCobj,'model',tf(eye(2)))
```

Restore the default input disturbance model.

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

Input Arguments

`MPCobj` — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`model` — Custom input disturbance model

`[]` (default) | `ss` object | `tf` object | `zpk` object

Custom input disturbance model, specified as a state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) model. The MPC controller converts the model to a discrete-time, delay-free, state-space model. Omitting `model` or specifying `model` as `[]` is equivalent to using `setindist(MPCobj,'integrators')`.

The input disturbance model has:

- Unit-variance white noise input signals. For custom input disturbance models, the number of inputs is your choice.
- n_d outputs, where n_d is the number of unmeasured disturbance inputs defined in `MPCobj.Model.Plant`. Each disturbance model output is sent to the corresponding plant unmeasured disturbance input.

This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

`setindist` does not check custom input disturbance models for violations of state observability. This check is performed later in the MPC design process when the internal state estimator is constructed using commands such as `sim` or `mpcmove`. If the controller states are not fully observable, these commands generate an error.

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

Tips

- To view the current input disturbance model, use the `getindist` command.

See Also

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

Topics

“MPC Modeling”
“Controller State Estimation”
“Adjust Disturbance and Noise Models”

Introduced before R2006a

setmpcsignals

Set signal types in MPC plant model

Syntax

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

Description

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

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

Unambiguous abbreviations of signal types are also accepted.

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

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

Examples

Set MPC Signal Types and Create MPC Controller

Create a four-input, two output state-space plant model. By default all input signals are manipulated variables and all outputs are measured outputs.

```
plant = rss(3,2,4);
plant.D = 0;
```

Configure the plant input/output channels such that:

- The second and third inputs are measured disturbances.
- The fourth input is an unmeasured disturbance.
- The second output is unmeasured.

```
plant = setmpcsignals(plant,'MD',[2 3],'UD',4,'UO',2);
```

```
-->Assuming unspecified input signals are manipulated variables.
-->Assuming unspecified output signals are measured outputs.
```

Create an MPC controller.

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

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

See Also

`mpc` | `set`

Topics

“MPC Modeling”

Introduced before R2006a

setname

Set I/O signal names in MPC prediction model

Syntax

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

Description

`setname(MPCobj,'input',I,name)` changes the name of the `I`th input signal to `name`. This is equivalent to `MPCobj.Model.Plant.InputName{I}=name`. Specify `name` as a character vector or string. 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`. Specify `name` as a character vector or string. Note that `setname` also updates the read-only `Name` field of `MPCobj.OutputVariables`.

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

See Also

`getname` | `mpc` | `set`

Introduced before R2006a

setoutdist

Modify unmeasured output disturbance model

Syntax

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

Description

`setoutdist(MPCobj,'model',model)` sets the output disturbance model used by the model predictive controller, `MPCobj`, to a custom model.

`setoutdist(MPCobj,'integrators')` sets the output disturbance model to its default value. Use this syntax if you previously set a custom output disturbance model and you want to change back to the default model. For more information on the default output disturbance model, see “MPC Modeling”.

Examples

Specify Output Disturbance Model Using Transfer Functions

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Define disturbance models for each output such that the output disturbance for:

- Channel 1 is random white noise with a magnitude of 2.
- Channel 2 is random step-like noise with a magnitude of `0.5`.
- Channel 3 is random ramp-like noise with a magnitude of 1.

```
mod1 = tf(2,1);
mod2 = tf(0.5,[1 0]);
mod3 = tf(1,[1 0 0]);
```

Construct the output disturbance model using these transfer functions. Use a separate noise input for each output disturbance.

```
outdist = [mod1 0 0; 0 mod2 0; 0 0 mod3];
```

Set the output disturbance model in the MPC controller.

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

View the controller output disturbance model.

```
getoutdist(MPCobj)
```

```
ans =

  A =
        x1   x2   x3
   x1    1    0    0
   x2    0    1    0
   x3    0  0.1    1

  B =
       Noise#1  Noise#2  Noise#3
   x1        0     0.05        0
   x2        0        0      0.1
   x3        0        0    0.005

  C =
        x1  x2  x3
   MO1   0   0   0
   MO2   1   0   0
   MO3   0   0   1

  D =
```

```
       Noise#1  Noise#2  Noise#3
   MO1        2        0        0
   MO2        0        0        0
   MO3        0        0        0

Sample time: 0.1 seconds
Discrete-time state-space model.
```

The controller converts the continuous-time transfer function model, `outdist`, into a discrete-time state-space model.

Remove Output Disturbance from Particular Output Channel

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Retrieve the default output disturbance model from the controller.

```
distMod = getoutdist(MPCobj);
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output channel #1 is integrated white
-->Assuming output disturbance added to measured output channel #2 is integrated white
-->Assuming output disturbance added to measured output channel #3 is integrated white
-->The "Model.Noise" property of the "mpc" object is empty. Assuming white noise on eac
```

Remove the integrator from the second output channel. Construct the new output disturbance model by removing the second input channel and setting the effect on the second output by the other two inputs to zero.

```
distMod = sminreal([distMod(1,1) distMod(1,3); 0 0; distMod(3,1) distMod(3,3)]);
setoutdist(MPCobj,'model',distMod)
```

When removing an integrator from the output disturbance model in this way, use `sminreal` to make the custom model structurally minimal.

View the output disturbance model.

```
tf(getoutdist(MPCobj))
```

```
ans =

  From input "Noise#1" to output...
          0.1
   MO1:  -----
         z - 1

   MO2:  0

   MO3:  0

  From input "Noise#2" to output...
   MO1:  0

   MO2:  0

          0.1
   MO3:  -----
         z - 1

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The integrator has been removed from the second channel. The disturbance models for channels 1 and 3 remain at their default values as discrete-time integrators.

Remove Output Disturbances from All Output Channels

Define a plant model with no direct feedthrough and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.D = 0;
MPCobj = mpc(plant,1);
```

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

Set the output disturbance model to zero for all three output channels.

```
setoutdist(MPCobj,'model',tf(zeros(3,1)))
```

View the output disturbance model.

```
getoutdist(MPCobj)
```

```
ans =

  D =
        Noise#1
   MO1        0
   MO2        0
   MO3        0

Static gain.
```

A static gain of `0` for all output channels indicates that the output disturbances were removed.

Set Output Disturbance Model to Default Value

Define a plant model with no direct feedthrough and create an MPC controller for that plant.

```
plant = rss(2,2,2);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Remove the output disturbances for all channels.

```
setoutdist(MPCobj,'model',tf(zeros(2,1)))
```

Restore the default output disturbance model.

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

Input Arguments

`MPCobj` — Model predictive controller

MPC controller object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`model` — Custom output disturbance model

`[]` (default) | `ss` object | `tf` object | `zpk` object

Custom output disturbance model, specified as a state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) model. The MPC controller converts the model to a discrete-time, delay-free, state-space model. Omitting `model` or specifying `model` as `[]` is equivalent to using `setoutdist(MPCobj,'integrators')`.

The output disturbance model has:

- Unit-variance white noise input signals. For custom output disturbance models, the number of inputs is your choice.
- n_y outputs, where n_y is the number of plant outputs defined in `MPCobj.Model.Plant`. Each disturbance model output is added to the corresponding plant output.

This model, along with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see “MPC Modeling” and “Controller State Estimation”.

`setoutdist` does not check custom output disturbance models for violations of state observability. This check is performed later in the MPC design process when the internal state estimator is constructed using commands such as `sim` or `mpcmove`. If the controller states are not fully observable, these commands will generate an error.

Tips

- To view the current output disturbance model, use the `getoutdist` command.

See Also

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

Topics

“MPC Modeling”
“Controller State Estimation”
“Adjust Disturbance and Noise Models”

Introduced in R2006a

setterminal

Terminal weights and constraints

Syntax

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

Description

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

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

Input Arguments

`MPCobj`

MPC controller, specified as an MPC controller object on page 2-71

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 “Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”.

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 “Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”.

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

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

Default:

`Pt`

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

Default: Prediction horizon *p*

Examples

Specify Constraints and Penalty Weights at Last Prediction Horizon Step

Create an MPC controller for a plant with three output variables and two manipulated variables.

```
plant = rss(3,3,2);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Specify a prediction horizon of `8`.

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

Define the following penalty weights and constraints:

- Diagonal penalty weights of `1` and `10` on the first two output variables
- Lower bounds of `0` and `-1` on the first and third outputs respectively
- Upper bound of `2` on the second output
- Lower bound of `1` on the first manipulated variable

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

Specify the constraints and penalty weights at the last step of the prediction horizon.

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

Specify Terminal Constraints For Final Prediction Horizon Range

Create an MPC controller for a plant with three output variables and two manipulated variables.

```
plant = rss(3,3,2);
plant.D = 0;
MPCobj = mpc(plant,0.1);
```

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

Specify a prediction horizon of `10`.

```
MPCobj.PredictionHorizon = 10;
```

Define the following terminal constraints:

- Lower bounds of `0` and `-1` on the first and third outputs respectively
- Upper bound of `2` on the second output

- Lower bound of 1 on the first manipulated variable

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

Specify the constraints beginning at step 5 and ending at the last step of the prediction horizon.

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

Tips

- Advanced users can impose terminal polyhedral state constraints:

  $K_1 \le Hx \le K_2$.

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

See Also

`mpc` | `mpcprops` | `setconstraint`

Topics

“Provide LQR Performance Using Terminal Penalty Weights”
“Designing Model Predictive Controller Equivalent to Infinite-Horizon LQR”
“Terminal Weights and Constraints”

Introduced in R2011a

sim

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

Syntax

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

Description

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

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

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

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

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

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

Input Arguments

`MPCobj`

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

`T`

Number of simulation steps, specified as a positive integer.

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

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

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

`r`

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

Default: `MPCobj.Model.Nominal.Y`

`v`

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

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

`SimOptions`

Simulation options, specified as an options object you create using `mpcsimopt`.

Default: `[]`

Output Arguments

`y`

Sequence of controlled plant outputs, returned as a `T`-by-`Ny` array, where `T` is the number of simulation steps and `Ny` is the number of plant outputs. The values in `y` do not include additive measurement noise, if any).

`t`

Time sequence, returned as a `T`-by-1 array, where `T` is the number of simulation steps. The values in `t` are equally spaced by `MPCobj.Ts`.

`u`

Sequence of manipulated variables generated by the MPC controller, returned as a `T`-by-`Nu` array, where `T` is the number of simulation steps and `Nu` is the number of manipulated variables.

`xp`

Sequence of plant model states, `T`-by-`Nxp` array, where `T` is the number of simulation steps and `Nxp` is the number of states in the plant model. The plant model is either `MPCobj.Model` or `SimOptions.Model`, if the latter is specified.

`xmpc`

Sequence of MPC controller state estimates, returned as a `T`-by-1 structure array. Each entry in the structure array has the same fields as an `mpcstate` object. The state estimates include plant, disturbance, and noise model states at each time step.

`SimOptions`

Simulation options used, returned as a `mpcsimopt` object.

Examples

Simulate MPC Control of MISO Plant

Simulate the MPC control of a MISO system. The system has one manipulated variable, one measured disturbance, one unmeasured disturbance, and one output.

Create the continuous-time plant model. This plant will be used as the prediction model for the MPC controller.

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

Discretize the plant model using a sampling time of 0.2 units.

```
Ts = 0.2;
sysd = c2d(sys,Ts);
```

Specify the MPC signal type for the plant input signals.

```
sysd = setmpcsignals(sysd,'MV',1,'MD',2,'UD',3);
```

Create an MPC controller for the `sysd` plant model. Use default values for the weights and horizons.

```
MPCobj = mpc(sysd);
```

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

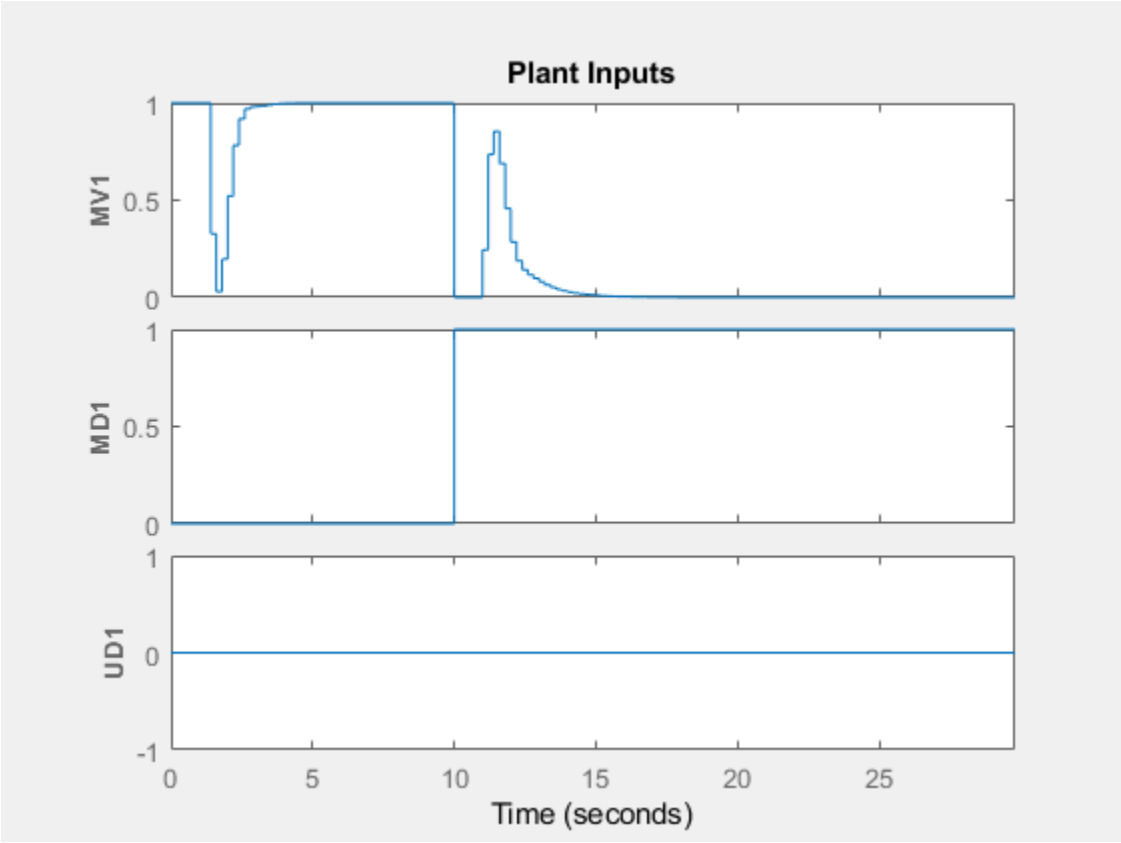
```
num_sim_steps = round(Tstop/Ts);
r = ones(num_sim_steps,1);
v = [zeros(num_sim_steps/3,1); ones(2*num_sim_steps/3,1)];
```

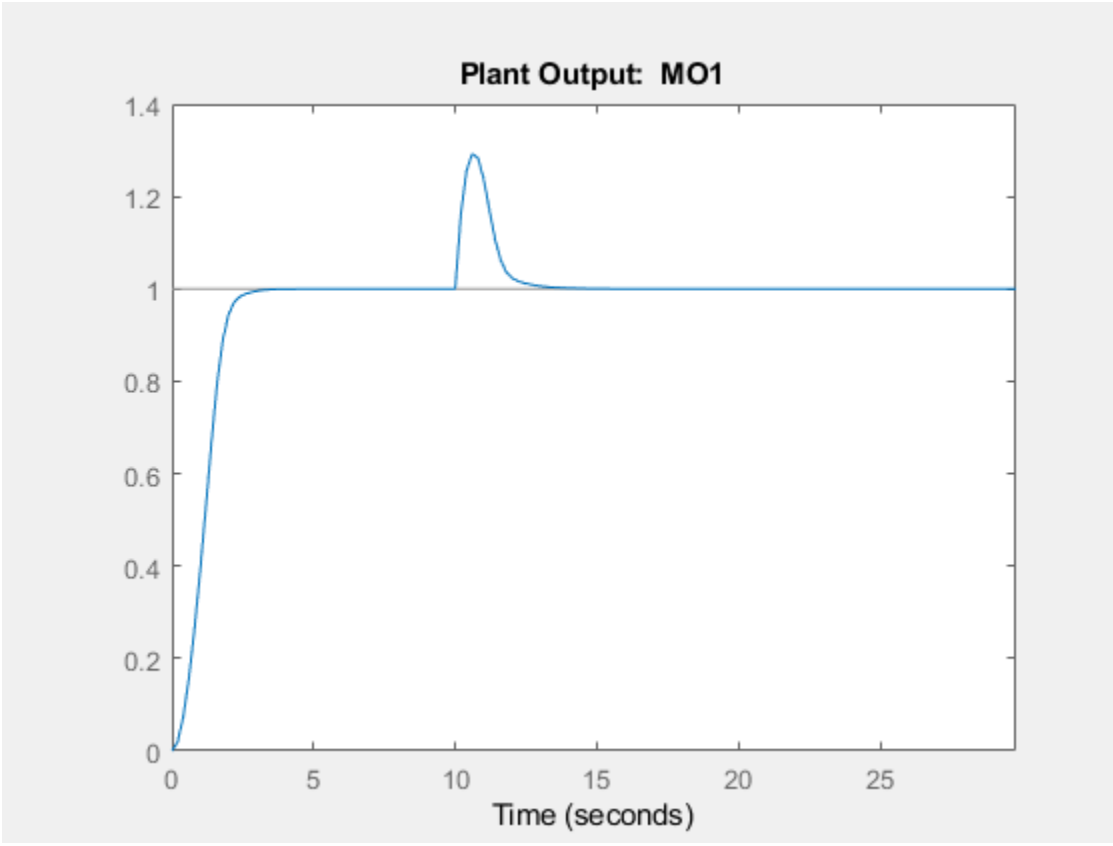
The reference signal, `r`, is a unit step. The measured disturbance signal, `v`, is a unit step, with a 10 unit delay.

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





See Also

`mpc` | `mpcmove` | `mpcsimopt`

Introduced before R2006a

simplify

Reduce explicit MPC controller complexity and memory requirements

Syntax

```
EMPCreduced = simplify(EMPCobj,'exact')
EMPCreduced = simplify(EMPCobj,'exact',uniteeps)
EMPCreduced = simplify(EMPCobj,'radius',r)
EMPCreduced = simplify(EMPCobj,'sequence',index)
simplify(EMPCobj,___)
```

Description

`EMPCreduced = simplify(EMPCobj,'exact')` attempts to reduce the number of piecewise affine (PWA) regions in an explicit MPC controller by merging regions that have identical controller gains and whose union is a convex set. Reducing the number of PWA regions reduces memory requirements of the controller. This command returns a reduced controller, `EMPCreduced`.

`EMPCreduced = simplify(EMPCobj,'exact',uniteeps)` specifies the tolerance for identifying regions that can be merged.

`EMPCreduced = simplify(EMPCobj,'radius',r)` retains only regions whose Chebyshev radius (the radius of the largest ball contained in the region) is larger than `r`.

`EMPCreduced = simplify(EMPCobj,'sequence',index)` eliminates all regions except those specified in an index vector.

`simplify(EMPCobj,___)` applies the reduction to the explicit MPC controller `EMPCobj`, rather than returning a new controller object. You can use this syntax with any of the previous reduction options.

Input Arguments

`EMPCobj` — Explicit MPC controller
explicit MPC controller object

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

`uniteeps` — Tolerance for joining regions
0.001 (default) | positive scalar

Tolerance for joining PWA regions, specified as a positive scalar.

`r` — Minimum Chebyshev radius
0 (default) | nonnegative scalar

Minimum Chebyshev radius for retaining PWA regions, specified as a nonnegative scalar. When you use the `'radius'` option, `simplify` keeps only the regions whose Chebyshev radius is larger than `r`. The default value is 0, which causes all regions to be retained.

`index` — Indices of PWA regions to retain
`1:nr` (default) | vector

Indices of PWA regions to retain, specified as a vector. The default value is `[1:nr]`, where `nr` is the number of PWA regions in `EMPCobj`. Thus, by default, all regions are retained. You can obtain a sequence of regions to retain by performing simulations using `EMPCobj` and recording the indices of regions actually encountered.

Output Arguments

`EMPCreduced` — Reduced MPC controller
explicit MPC controller object

Reduced MPC controller, returned as an Explicit MPC controller object.

See Also

`generateExplicitMPC`

Introduced in R2014b

size

Size and order of MPC Controller

Syntax

```
mpc_obj_size = size(MPCobj)
mpc_obj_size = size(MPCobj,signal_type)
size(MPCobj)
```

Description

`mpc_obj_size = size(MPCobj)` returns a row vector specifying the number of manipulated inputs and measured controlled outputs of an MPC controller. This row vector contains the elements [n_u n_{ym}], where n_u is the number of manipulated inputs and n_{ym} is the number of measured controlled outputs.

`mpc_obj_size = size(MPCobj,signal_type)` returns the number of signals of the specified type that are associated with the MPC controller.

You can specify `signal_type` as one of the following:

- `'uo'` — Unmeasured controlled outputs
- `'md'` — Measured disturbances
- `'ud'` — Unmeasured disturbances
- `'mv'` — Manipulated variables
- `'mo'` — Measured controlled outputs

`size(MPCobj)` displays the size information for all the signal types of the MPC controller.

See Also

`mpc` | `set`

Introduced before R2006a

ss

Convert unconstrained MPC controller to state-space linear system

Syntax

```
sys = ss(MPCobj)
sys = ss(MPCobj,signals)
sys = ss(MPCobj,signals,ref_preview,md_preview)
[sys,ut] = ss(MPCobj)
```

Description

The `ss` command returns a linear controller in the state-space form. The controller is equivalent to the traditional (implicit) MPC controller `MPCobj` when no constraints are active. You can then use Control System Toolbox software for sensitivity analysis and other diagnostic calculations.

`sys = ss(MPCobj)` returns the linear discrete-time dynamic controller `sys`.

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

$u(k) = Cx(k) + Dy_m(k)$

where y_m is the vector of measured outputs of the plant, and u is the vector of manipulated variables. The sampling time of controller `sys` is `MPCobj.Ts`.

Note Vector x includes the states of the observer (plant + disturbance + noise model states) and the previous manipulated variable $u(k\text{-}1)$.

`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 character vector or string with any combination that contains one or more of the following characters:

- `'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}, 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} and 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:

$x(k+1) = Ax(k) +...+ B_v[v(k);v(k+1);...;v(k+p)] +...$

$u(k) = Cx(k) +...+ D_v[v(k);v(k+1);...;v(k+p)] +...$

`[sys,ut] = ss(MPCobj)` also returns the input target values for the full form of the controller.

`ut` is returned as a vector of doubles, `[utarget(k); utarget(k+1); ... utarget(k+h)]`.

Here:

- h — Maximum length of previewed inputs; that is, `h = max(length(MPCobj.ManipulatedVariables(:).Target))`
- `utarget` — Difference between the input target and corresponding input offsets; that is, `MPCobj.ManipulatedVariables(:).Targets - MPCobj.Model.Nominal.U`

Examples

Convert Unconstrained MPC Controller to State-Space Model

To improve the clarity of the example, suppress messages about working with an MPC controller.

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

Create the plant model.

```
G = rss(5,2,3);
G.D = 0;
G = setmpcsignals(G,'mv',1,'md',2,'ud',3,'mo',1,'uo',2);
```

Configure the MPC controller with nonzero nominal values, weights, and input targets.

```
C = mpc(G,0.1);
C.Model.Nominal.U = [0.7 0.8 0];
```

```
C.Model.Nominal.Y = [0.5 0.6];
C.Model.Nominal.DX = rand(5,1);
C.Weights.MV = 2;
C.Weights.OV = [3 4];
C.MV.Target = [0.1 0.2 0.3];
```

`C` is an unconstrained MPC controller. Specifying `C.Model.Nominal.DX` as nonzero means that the nominal values are not at steady state. `C.MV.Target` specifies three preview steps.

Convert `C` to a state-space model.

```
sys = ss(C);
```

The output, `sys`, is a seventh-order SISO state-space model. The seven states include the five plant model states, one state from the default input disturbance model, and one state from the previous move, `u(k-1)`.

Restore `mpcverbosity`.

```
mpcverbosity(old_status);
```

See Also

`mpc` | `set` | `tf` | `zpk`

Introduced before R2006a

tf

Convert unconstrained MPC controller to linear transfer function

Syntax

```
sys=tf(MPCobj)
```

Description

The `tf` function computes the transfer function of the linear controller `ss(MPCobj)` as an LTI system in `tf` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `zpk`

Introduced before R2006a

trim

Compute steady-state value of MPC controller state for given inputs and outputs

Syntax

```
x = trim(MPCobj,y,u)
```

Description

The `trim` function finds a steady-state value for the plant state or the best approximation in a least squares sense 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 respectively.

`x` is returned as an `mpcstate` object. Specify `y` and `u` as doubles. `y` specifies the measured and unmeasured output values. `u` specifies the manipulated variable, measured disturbance, and unmeasured disturbance values. The values for unmeasured disturbances must be `0`.

`trim` assumes the disturbance model and measurement noise model to be zero when computing the steady-state value. The software uses the extended state vector to perform the calculation.

See Also

`mpc` | `mpcstate`

Introduced before R2006a

validateFcns

Examine prediction model and custom functions of `nlmpc` object for potential problems

`validateFunctions` tests the prediction model, custom cost, custom constraint, and Jacobian functions of a nonlinear MPC controller for potential problems. When you first design your nonlinear MPC controller, or when you make significant changes to an existing controller, it is best practice to validate your controller functions.

Syntax

```
validateFcns(nlmpcobj,x,mv)
validateFcns(nlmpcobj,x,mv,md)
validateFcns(nlmpcobj,x,mv,md,parameters)
validateFcns(nlmpcobj,x,mv,md,parameters,ref)
validateFcns(nlmpcobj,x,mv,md,parameters,ref,mvtarget)
```

Description

`validateFcns(nlmpcobj,x,mv)` tests the functions of nonlinear MPC controller `nlmpcobj` for potential problems. The functions are tested using specified nominal state values, `x`, and manipulated variable values, `mv`. Use this syntax if your controller has no measured disturbances and no parameters.

`validateFcns(nlmpcobj,x,mv,md)` specifies nominal measured disturbance values. If your controller has measured disturbance channels, you must specify `md`.

`validateFcns(nlmpcobj,x,mv,md,parameters)` specifies nominal parameter values. If your controller has parameters, you must specify `parameters`.

`validateFcns(nlmpcobj,x,mv,md,parameters,ref)` specifies nominal output references.

`validateFcns(nlmpcobj,x,mv,md,parameters,ref,mvtarget)` specifies nominal manipulated variable targets.

Examples

Validate Nonlinear MPC Prediction Model and Custom Functions

Create nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Validate the prediction model and custom functions at the initial states (`x0`) and initial inputs (`u0`) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Nonlinear MPC Controller with Discrete-Time Prediction Model

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nx = 4;
ny = 2;
nu = 1;
nlobj = nlmpc(nx,ny,nu);
```

```
In standard cost function, zero weights are applied by default to one or more OVs becau
```

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The discrete-time state function uses an optional parameter, the sample time `Ts`, to integrate the continuous-time model. Therefore, you must specify the number of optional parameters as `1`.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function for the controller. In this case, define the first and third states as outputs. Even though this output function does not use the optional sample time parameter, you must specify the parameter as an input argument (`Ts`).

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Validate the prediction model functions for nominal states `x0` and nominal inputs `u0`. Since the prediction model uses a custom parameter, you must pass this parameter to `validateFcns`.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Nonlinear MPC Controller with Measured and Unmeasured Disturbances

Create a nonlinear MPC controller with three states, one output, and four inputs. The first two inputs are measured disturbances, the third input is the manipulated variable, and the fourth input is an unmeasured disturbance.

```
nlobj = nlmpc(3,1,'MV',3,'MD',[1 2],'UD',4);
```

To view the controller state, output, and input dimensions and indices, use the `Dimensions` property of the controller.

```
nlobj.Dimensions
```

```
ans = struct with fields:
     NumberOfStates: 3
    NumberOfOutputs: 1
     NumberOfInputs: 4
            MVIndex: 3
            MDIndex: [1 2]
            UDIndex: 4
```

Specify the controller sample time and horizons.

```
nlobj.Ts = 0.5;
nlobj.PredictionHorizon = 6;
nlobj.ControlHorizon = 3;
```

Specify the prediction model state function, which is in the file `exocstrStateFcnCT.m`.

```
nlobj.Model.StateFcn = 'exocstrStateFcnCT';
```

Specify the prediction model output function, which is in the file `exocstrOutputFcn.m`.

```
nlobj.Model.OutputFcn = 'exocstrOutputFcn';
```

Validate the prediction model functions using the initial operating point as the nominal condition for testing and setting the unmeasured disturbance state, `x0(3)`, to `0`. Since the model has measured disturbances, you must pass them to `validateFcns`.

```
x0 = [311.2639; 8.5698; 0];
u0 = [10; 298.15; 298.15];
validateFcns(nlobj,x0,u0(3),u0(1:2)');
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Input Arguments

`nlmpcobj` — Nonlinear MPC controller

`nlmpc` object

Nonlinear MPC controller, specified as an `nlmpc` object.

`x` — Nominal state values

vector

Nominal state values, specified as a vector of length N_x, N_x is equal to `nlmpcobj.Dimensions.NumberOfStates`

`mv` — Nominal manipulated variable values

vector

Nominal manipulated variable values, specified as a vector of length N_{mv}, where N_{mv} is equal to the length of `nlmpcobj.Dimensions.MVIndex`.

`md` — Nominal measured disturbance values

`[]` (default) | vector

Nominal measured disturbance values, specified as a vector of length N_{md}, where N_{md} is equal to the length of `nlmpcobj.Dimensions.MDIndex`. If your controller has measured

disturbance channels, you must specify `md`. If your controller does not have measured disturbance channels, specify `md` as `[]`.

`parameters` — Nominal parameter values
`[]` (default) | cell array

Nominal parameter values used by the prediction model, custom cost function, and custom constraints, specified as a cell array of length N_p, where N_p is equal to `nlmpcobj.Model.NumberOfParameters`. The order of the parameters must match the order specified in the model functions, and each parameter must be a numeric parameter with the correct dimensions.

If your controller has parameters, you must specify `parameters`. If your controller does not have parameters, specify `parameters` as `[]`.

`ref` — Nominal output reference values
`[]` (default) | vector

Nominal output references values, specified as a vector of length N_y, where Ny is equal to `nlmpcobj.Dimensions.NumberOfOutputs`. `ref` is passed to the custom cost and constraint function. If you do not specify `ref`, the controller passes a vector of zeros to the custom functions.

`mvtarget` — Nominal manipulated variable targets
`[]` (default) | vector

Nominal manipulated variable targets, specified as a vector of length N_{mv}, where N_{mv} is equal to the length of `nlmpcobj.Dimensions.MVIndex`. `mvtarget` is passed to the custom cost and constraint function. If you do not specify `mvtarget`, the controller passes a vector of zeros to the custom functions.

Tips

- When you provide your own analytical Jacobian functions, it is especially important that these functions return valid Jacobian values. If `validateFunctions` detects large differences between the values returned by your user-defined Jacobian functions and the finite-difference approximation, verify the code in your Jacobian implementations.

Algorithms

For each controller function, `validateFunctions` checks whether the function:

- Exists on the MATLAB path
- Has the required number of input arguments
- Can be executed successfully without errors
- Returns the output arguments with the correct size and dimensions
- Returns valid numerical data; that is, it does not return `Inf` or `NaN` values

For Jacobian functions, `validateFunctions` checks whether the returned values are comparable to a finite-difference approximation of the Jacobian values. These finite-difference values are computed using numerical perturbation.

See Also

`nlmpc`

Topics

“Specify Prediction Model for Nonlinear MPC”
“Specify Cost Function for Nonlinear MPC”
“Specify Constraints for Nonlinear MPC”

Introduced in R2018b

zpk

Convert unconstrained MPC controller to zero/pole/gain form

Syntax

```
sys=zpk(MPCobj)
```

Description

The `zpk` function computes the zero-pole-gain form of the linear controller `ss(MPCobj)` as an LTI system in `zpk` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `tf`

Introduced before R2006a

3

Block Reference

MPC Controller

Compute MPC control law

Library

MPC Simulink Library

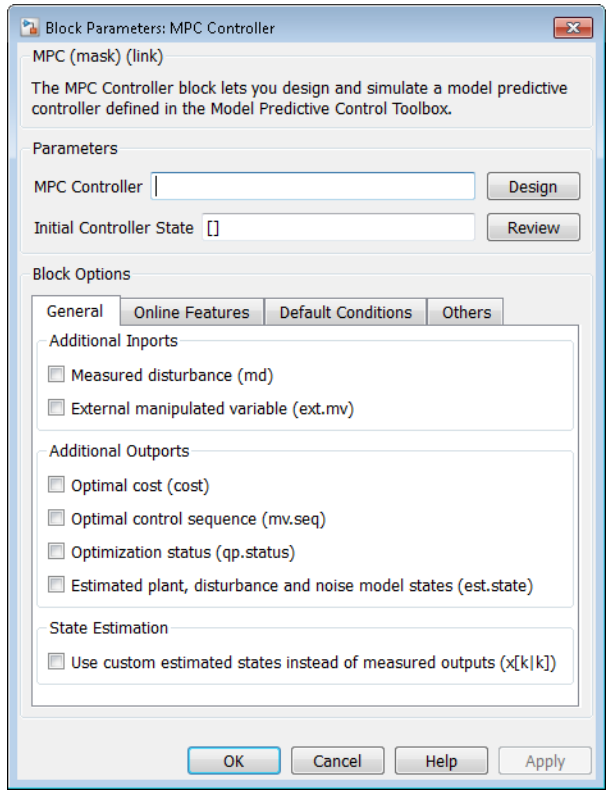
Description

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 programming problem using either the default KWIK solver or a custom QP solver. For more information, see “QP Solver”.

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 controls.

Because the MPC Controller block uses MATLAB Function blocks, it requires compilation each time you change the MPC object and block. Also, because MATLAB does not allow compiled code to reside in any MATLAB product folder, you must use a non-MATLAB folder to work on your Simulink model when you use MPC blocks.

Dialog Box



The MPC Controller block has the following parameter groupings:

- “Parameters” on page 3-4
- “Required Inports” on page 3-5
- “Required Outports” on page 3-6
- “Additional Inports (General Section)” on page 3-6
- “Additional Outports (General Section)” on page 3-9
- “State Estimation (General Section)” on page 3-11
- “Constraints (Online Features Section)” on page 3-11
- “Weights (Online Features Section)” on page 3-14

Parameters

You must provide an `mpc` object that defines an implicit MPC controller. To do so:

- Enter the name of an `mpc` object in the **MPC Controller** edit box. This object must be present in the MATLAB workspace.

  If you want to modify the controller settings in a graphical environment, open the **MPC Designer** app by clicking **Design**. For example, you can:

  - Import a new prediction model.
  - Change horizons, constraints, and weights.
  - Evaluate MPC performance with a linear plant.
  - Export the updated controller to the MATLAB workspace.

  To see how well the controller works for the nonlinear plant, run a closed-loop Simulink simulation.
- If you do not have an existing `mpc` object in the MATLAB workspace, leave the **MPC controller** field empty. With the MPC Controller block connected to the plant, click **Design** to open **MPC Designer**. Using the app, linearize the Simulink model at a specified operating point, and design your controller. For more information, see “Design MPC Controller in Simulink” and “Linearize Simulink Models Using MPC Designer”.

  To use this design approach, you must have Simulink Control Design software.

Once you specify a controller in the **MPC Controller** field, you can review your design for run-time stability and robustness issues by clicking **Review**. For more information, see “Review Model Predictive Controller for Stability and Robustness Issues”.

Specifies the initial controller state. If this parameter is left blank, the block uses the nominal values that are defined in the `Model.Nominal` property of the `mpc` object. To override the default, create an `mpcstate` object in your workspace, and enter its name in the field.

Required Inports

Measured output or **State estimate**

If your controller uses default state estimation, this inport is labeled `mo`. Connect this inport to the measured plant output signals. The MPC controller uses measured plant outputs to improve its state estimates.

To enable custom state estimation, in the **General** section, check **Use custom estimated states instead of measured outputs**. Checking this option changes the label on this inport to `x[k|k]`. Connect a signal that provides estimates of the controller state (plant, disturbance, and noise model states). Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

Reference

The `ref` dimension must not change from one control instant to the next. Each element must be a real number.

When `ref` is a 1-by-n_y signal, where n_y is the number of outputs, there is no reference signal previewing. The controller applies the current reference values across the prediction horizon.

To use signal previewing, specify `ref` as an N-by-n_y signal, where N is the number of time steps for which you are specifying reference values. Here, $1 < N \le p$, and p is the prediction horizon. Previewing usually improves performance, since the controller can anticipate future reference signal changes. The first row of `ref` specifies the n_y references for the first step in the prediction horizon (at the next control interval k = 1), and so on for N steps. If $N < p$, the last row designates constant reference values for the remaining p - N steps.

For example, suppose n_y = 2 and p = 6. At a given control instant, the signal connected to the `ref` 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

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables for controlling the plant. At each control instant, the controller updates its `mv` outport by solving a quadratic programming problem using either the default KWIK solver or a custom QP solver. For more information, see “QP Solver”.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution; that is, the controller output freezes.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

Additional Inports (General Section)

Add an inport (`md`) to which you connect a measured disturbance signal. The number of measured disturbances defined for your controller, $n_{md} \geq 1$, must match the dimensions of the connected disturbance signal.

The number of measured disturbances must not change from one control instant to the next, and each disturbance value must be a real number.

When `md` is a 1-by-n_{md} signal, there is no measured disturbance previewing. The controller applies the current disturbance values across the prediction horizon.

To use disturbance previewing, specify `md` as an N-by-n_{md} signal, where N is the number of time steps for which the measured disturbances are known. Here, $1 < N \le p+1$, and p is the prediction horizon. Previewing usually improves performance, since the controller can anticipate future disturbances. The first row of `md` specifies the n_{md} current disturbance values ($k = 1$), with other rows specifying disturbances for subsequent control intervals. If $N < p + 1$, the controller applies the last row for the remaining $p - N + 1$ steps.

For example, suppose $n_{md} = 2$ and $p = 6$. At a given control instant, the signal connected to the `md` 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 `MD` values are `2` and `5` at $k = 0$.
- The first `MD` remains at `2`, but the second increases gradually.
- The second `MD` becomes `8` at the beginning of the third step $k = 3$ in the prediction horizon.
- Both values remain constant at `2` and `8` respectively for steps 4–6 of the prediction horizon.

`mpcpreview` shows how to use `MD` previewing in a specific case.

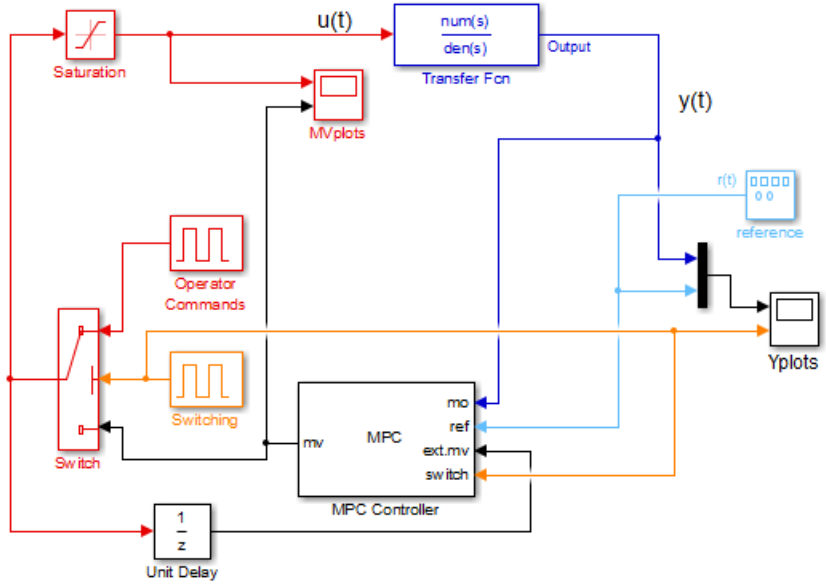
For calculation details, see “MPC Modeling” and “QP Matrices”.

Add an inport (`ext.mv`) to which you connect a vector signal that contains the actual n_u manipulated variables (MV) used in the plant. Use this option when the MV applied to the plant between time t_{k-1} and t_k is different than the optimal MV computed at the last control interval, for example due to signal saturation or an override condition. When enabled, the block uses this signal to correct controller state estimates at t_k.

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 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 Using this option can cause an algebraic loop in the Simulink model, since there is direct feedthrough from the `ext.mv` inport to the `mv` outport. To prevent such algebraic loops, insert a Memory block or Unit Delay block.

Additional Outports (General Section)

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`.)

Add an outport (`qp.status`) that allows you to monitor the status of the QP solver.

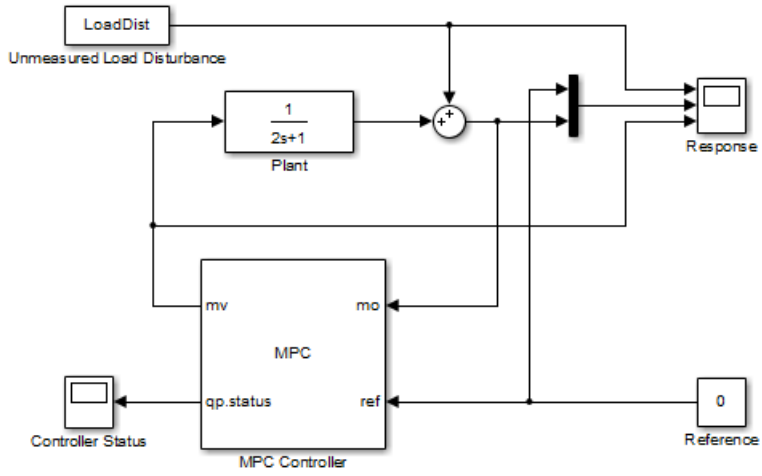
If a QP problem is solved successfully at a given control interval, the `qp.status` output returns the number of QP solver iterations used in computation. This value is a finite, positive integer and is proportional to the time required for the calculations. Thus, a large value means a relatively slow block execution for this time interval.

The QP solver can 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. In this case, if the `Optimizer.UseSuboptimalSolution` property of the MPC controller is `false`, the block holds its `mv` output at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.
- `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. In this case, the block holds its `mv` output at the most recent successful solution.
- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem. In this case, the block holds its `mv` output at the most recent successful solution.

In a real-time application, you can use `qp.status` to set an alarm or take other special action.

The following diagram shows how to use the status indicator to monitor the MPC Controller block in real time. For more information, see “Monitoring Optimization Status to Detect Controller Failures”.



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.

Add an outport (`mv.seq`) that provides the predicted optimal MV adjustments (moves) over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_u matrix, where and n_u is the number of manipulated variables.

`mv.seq` contains the calculated optimal MV moves at time `k+i-1`, for `i = 1,...,p`. The first row of `mv.seq` is identical to the `mv` outport signal, which is the current MV adjustment applied at time `k`. Since the controller does not calculate optimal control moves at time `k+p`, the last row of `mv.seq` duplicates the previous row.

Add an outport (`x.seq`) that provides the predicted optimal state variable sequence over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_x matrix, where n_x is the number of states in the plant and unmeasured disturbance models (states from noise models are not included).

`x.seq` contains the calculated optimal state values at time `k+i`, for `i = 1,...,p`. The first row of `x.seq` contains the current states at time `k` as determined by state estimation.

Add an outport (`y.seq`) that provides the predicted optimal output variable sequence over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_y matrix, where and n_y is the number of outputs.

`y.seq` contains the calculated optimal output values at time `k+i-1`, for `i = 1,...,p+1`. The first row of `y.seq` contains the current outputs at time `k` based on the estimated states and measured disturbances; it is not the measured output at time `k`.

State Estimation (General Section)

Replace `mo` with the `x[k|k]` inport for custom state estimation as described in “Required Inports” on page 3-5.

Constraints (Online Features Section)

Add inport `umin` that you can connect to a run-time constraint signal for manipulated variable lower bounds. This signal is a vector with n_u finite values. The `i`th element of `umin` replaces the `ManipulatedVariables(i).Min` property of the controller at run time.

If a manipulated variable does not have a lower bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `ManipulatedVariables(i).Min` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `umin` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inport `umax` that you can connect to a run-time constraint signal for manipulated variable upper bounds. This signal is a vector with n_u finite values. The `i`th element of

`umax` replaces the `ManipulatedVariables(i).Max` property of the controller at run time.

If a manipulated variable does not have an upper bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `ManipulatedVariables(i).Max` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `umax` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inport `ymin` that you can connect to a run-time constraint signal for output variable lower bounds. This signal is a vector with n_y finite values. The `i`th element of `ymin` replaces the `OutputVariables(i).Min` property of the controller at run time.

If an output variable does not have a lower bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `OutputVariables(i).Min` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `ymin` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inport `ymax` that you can connect to a run-time constraint signal for output variable upper bounds. This signal is a vector with n_y finite values. The `i`th element of `ymax` replaces the `OutputVariables(i).Max` property of the controller at run time.

If an output variable does not have an upper bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `OutputVariables(i).Max` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `ymax` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inports `E`, `F`, `G`, and `S` to the block that you can connect to the following run-time custom constraint matrix signals:

- `E` — Manipulated variable constraint matrix with size n_c-by-n_u, where n_c is the number of custom constraints
- `F` — Controlled output constraint matrix with size n_c-by-n_y
- `G` — Custom constraint matrix with size 1-by-n_c
- `S` — Measured disturbance constraint matrix, with size n_c-by-n_v, where n_v is the number of measured disturbances. `S` is added only if the `mpc` object has measured disturbances.

These constraints replace the custom constraints previously set using `setconstraint`.

If you define `E`, `F`, `G`, or `S` in the `mpc` object, you must connect a signal to the corresponding inport, and that signal must have the same dimensions as the array specified in the controller. If an array is not defined in the controller object, use a zero matrix with the correct size.

The custom constraints are of the form `E`u + `F`y + `S`v <= `G`, where:

- u is a vector of manipulated variable values.
- y is a vector of predicted plant output values.
- v is a vector of measured plant disturbance input values.

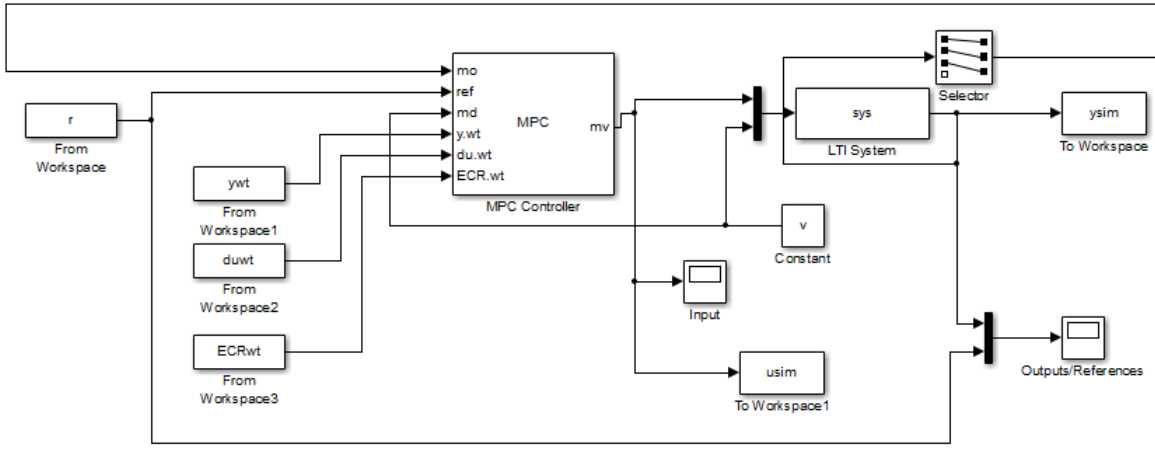
For more information, see “Constraints on Linear Combinations of Inputs and Outputs”.

Weights (Online Features Section)

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 shows three of the MPC Controller tuning inports. In this simulation context, the inports are tuned using pre-stored signals (the `ywt`, `duwt`, and `ECRwt` variables in the From Workspace blocks). In practice, you would connect a knob or similar manual adjustment.

Note You cannot specify time-varying weights at run time using a matrix signal.



Add an inport (`y.wt`) that you can connect to a run-time output variable (OV) weight signal. This signal overrides the `Weights.OV` property of the `mpc` object, which establishes the relative importance of OV reference tracking.

To use the same tuning weights over the prediction horizon, connect `y.wt` to a vector signal with n_y elements, where n_y is the number of controlled outputs. Each element specifies a nonnegative tuning weight for each controlled output variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `y.wt` to a matrix signal with n_y columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

If you do not connect a signal to the `y.wt` inport, the block uses the OV weights specified in your `mpc` object.

Add an inport (`u.wt`) that you can connect to a run-time manipulated variable (MV) weight signal. This signal overrides the `Weights.MV` property of the `mpc` object, which establishes the relative importance of MV target tracking.

To use the same tuning weights over the prediction horizon, connect `u.wt` to a vector signal with n_u elements, where n_u is the number of manipulated variables. Each element specifies a nonnegative tuning weight for each manipulated variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `u.wt` to a matrix signal with n_u columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

If you do not connect a signal to the `u.wt` inport, the block uses the MV weights specified in your `mpc` object.

Add an inport (`du.wt`) that you can connect to a run-time manipulated variable (MV) rate weight signal. This signal overrides the `Weights.MVrate` property of the `mpc` object, which establishes the relative importance of MV changes.

To use the same tuning weights over the prediction horizon, connect `du.wt` to a vector signal with n_u elements, where n_u is the number of manipulated variables. Each element specifies a nonnegative tuning weight for each manipulated variable rate. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `du.wt` to a matrix signal with n_u columns and up to p rows, where p is the prediction horizon. Each row contains

the tuning weights for one prediction horizon step. If you specify fewer than *p* rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

If you do not connect a signal to the `du.wt` inport, the block uses the MV rate weights specified in your `mpc` object.

Add an inport (`ecr.wt`), for a scalar nonnegative signal that overrides the `MPCobj.Weights.ECR` property of the `mpc` controller. 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 Section)

If you want one or more manipulated variables (MV) to track target values that change with time, use this option to add an `mv.target` inport. Connect this port to a target signal with dimension n_u, where n_u is the number of MVs.

For this to be effective, the corresponding MV(s) must have nonzero penalty weights (these weights are zero by default).

Default Conditions Section

Specify the default block sample time and signal dimensions for performing simulation, trimming, or linearization. In these cases, the `mv` output signal remains at zero. You must specify default condition values that are compatible with your Simulink model design.

Note These default conditions apply only if the **MPC Controller** field is empty. If you specify a controller from the MATLAB workspace, the sample time and signal sizes from the specified controller are used.

Specify the default controller sample time.

Specify the default signal dimensions for the following input signal types:

- Manipulated variables
- Unmeasured disturbances
- Measured disturbances

Note You can specify the measured disturbances signal dimension only if, on the **General** section, in the **Additional Inports** section, the **Measured disturbance** option is selected.

Specify the default signal dimensions for the following output signal types:

- Measured outputs
- Unmeasured outputs

Others Section

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

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

For an example of double-precision and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**.

Use the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results .

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” (Simulink).

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 is 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. However, the controller continues to update its internal state estimates in the usual way. Thus, it is ready to resume optimization calculations whenever the `switch` signal returns to zero. While controller optimization is 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 the value it had when optimization was disabled.

Compatibility Considerations

MPC Simulink block `mv.seq` output port signal dimensions have changed

Behavior changed in R2018b

The signal dimensions of the `mv.seq` output port of the MPC Controller block have changed. Previously, this signal was a p-by-N_{mv} matrix, where p is the prediction horizon and N_{mv} is the number of manipulated variables. Now, `mv.seq` is a (p+1)-by-N_{mv} matrix, where row p+1 duplicates row p.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks
Multiple MPC Controllers

Functions
`mpc` | `mpcstate`

Apps
MPC Designer

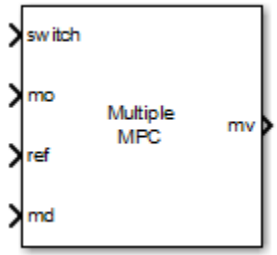
Topics

“MPC Modeling”
“Design MPC Controller in Simulink”
“Switching Controllers Based on Optimal Costs”
“Understanding Control Behavior by Examining Optimal Control Sequence”
“Simulation and Code Generation Using Simulink Coder”
“Simulation and Structured Text Generation Using PLC Coder”

Introduced before R2006a

Multiple MPC Controllers

Simulate switching between multiple implicit MPC controllers



Library

MPC Simulink Library

Description

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 a list of candidate MPC controllers designed at different operating points within the operating range. 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 enables you to achieve better control when operating conditions change. Using available measurements, you can detect the current operating region at run time and choose the appropriate active controller via the `switch` inport. Switching controllers for different operating regions is a common approach to solving nonlinear control problems using linear control techniques.

To improve efficiency, inactive controllers do not compute optimal control moves. However, to provide bumpless transfer between controllers, the inactive controllers continue to perform state estimation.

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 box; that is, there is no **Design** button. Design all candidate controllers before configuring the Multiple MPC Controllers block.
- Similarly, there is no **Review** button. Instead, use the `review` command or the **MPC Designer** app.
- You cannot update custom constraints on linear combinations of inputs and outputs at run time.

The Adaptive MPC Controller block compensates for operating point variations by modifying its prediction model. The advantages of the Multiple MPC Controllers block over Adaptive MPC Controller block are as follows:

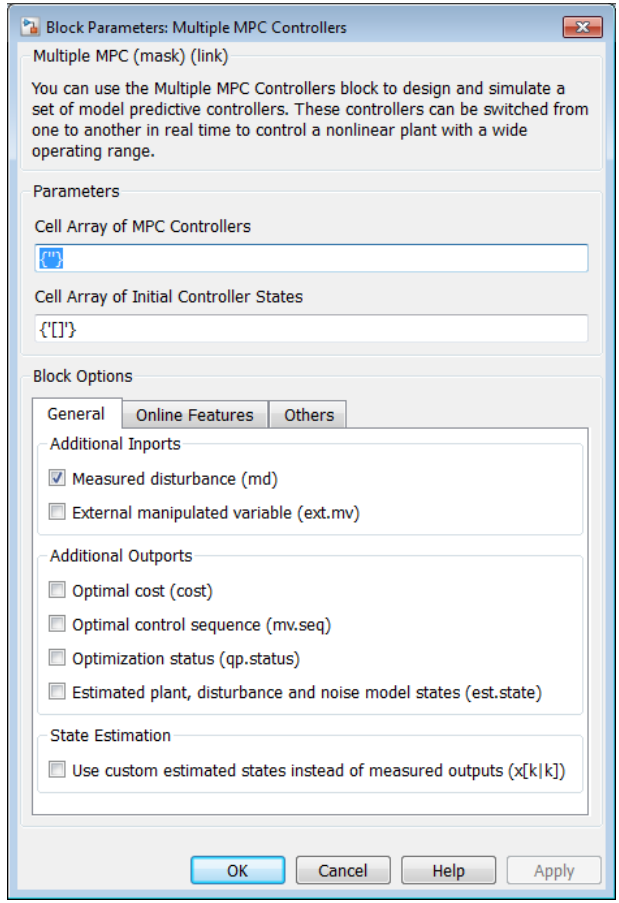
- Simpler configuration – There is no need to identify prediction model parameters using online data.
- Its candidate controllers form a limited set that you can test thoroughly.

Both the Multiple MPC Controllers block and the Adaptive MPC Controller block enable your control system to adapt to changing operating conditions at run time. The following table lists the advantages of using each block.

Block	Adaptive MPC Controller	Multiple MPC Controllers
Adaptation approach	Update prediction model for a single controller as operating conditions change	Switch between multiple controllers designed for different operating regions

Block	Adaptive MPC Controller	Multiple MPC Controllers
Advantages	• Only need to design a single controller offline • Less run-time computational effort and smaller memory footprint • More robust to real-life changes in plant conditions	• No need for online estimation of plant model • Controllers can have different sample time, horizons, and weights • Prediction models can have different orders or time domains • Finite set of candidate controllers can be tested thoroughly

Dialog Box



The Multiple MPC Controller block has the following parameter groupings:

- “Parameters” on page 3-24
- “Required Inports” on page 3-24
- “Required Outports” on page 3-26
- “Additional Inports (General Section)” on page 3-27
- “Additional Outports (General Section)” on page 3-28

Parameters

Cell Array of MPC Controllers

Candidate controllers, specified as:

- A cell array of `mpc` objects.
- A cell array of character vectors, where each element is the name of an `mpc` object in the MATLAB workspace.

The specified array must contain at least two candidate controllers. The first entry in the cell array is the controller that corresponds to a switch input value of 1, the second corresponds to a switch input value of 2, and so on.

Cell Array of Initial Controller States

Optional initial states for each candidate controller, specified as:

- A cell array of `mpcstate` objects.
- A cell array of character vectors, where each element is the name of an `mpcstate` object in the MATLAB workspace.
- `{[],[],...}` or `{'[]','[]',...}` — Use the nominal condition defined in `Model.Nominal` as the initial state for each controller.

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 specified candidate controllers. At each control instant, this signal designates the active controller. A switch value of 1 corresponds to the first entry in the cell array of candidate controllers, a value of 2 corresponds to the second controller, and so on.

If the `switch` signal is outside of the range 1 and n_c, the previous controller output is retained.

Measured output or **State estimate**

If candidate controllers use default state estimation, this inport is labeled `mo`. Connect this inport to the measured plant output signals.

If your candidate controllers use custom state estimation, check **Use custom estimated states instead of measured outputs** in the **General** section. Checking this 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, either default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Reference

The `ref` dimension must not change from one control instant to the next. Each element must be a real number.

When `ref` is a 1-by-n_y signal, where n_y is the number of outputs, there is no reference signal previewing. The controller applies the current reference values across the prediction horizon.

To use signal previewing, specify `ref` as an N-by-n_y signal, where N is the number of time steps for which you are specifying reference values. Here, $1 < N \le p$, and p is the prediction horizon. Previewing usually improves performance, since the controller can anticipate future reference signal changes. The first row of `ref` specifies the n_y references for the first step in the prediction horizon (at the next control interval k = 1), and so on for N steps. If $N < p$, the last row designates constant reference values for the remaining p - N steps.

For example, suppose $n_y = 2$ and $p = 6$. At a given control instant, the signal connected to the `ref` 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

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables for controlling the plant. The active controller updates its manipulated variable output by solving a quadratic programming problem using either the default KWIK solver or a custom QP solver. For more information, see “QP Solver”.

The Multiple MPC Controller block passes the output of the active controller to the `mv` outport.

If the active controller detects an infeasible QP problem or encounters numerical difficulties in solving an ill-conditioned QP problem, `mv` remains at its most recent successful solution, `x.LastMove`.

Otherwise, if the QP problem is feasible and the solver reaches the specified maximum number of iterations without finding a solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the active controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the active controller is `true`. For more information, see “Suboptimal QP Solution”.

Additional Inports (General Section)

Add an inport (`md`) to which you connect a measured disturbance signal. The number of measured disturbances defined for your controller, $n_{md} \geq 1$, must match the dimensions of the connected disturbance signal.

The number of measured disturbances must not change from one control instant to the next, and each disturbance value must be a real number.

When `md` is a 1-by-n_{md} signal, there is no measured disturbance previewing. The controller applies the current disturbance values across the prediction horizon.

To use disturbance previewing, specify `md` as an N-by-n_{md} signal, where N is the number of time steps for which the measured disturbances are known. Here, $1 < N \leq p+1$, and p is the prediction horizon. Previewing usually improves performance, since the controller can anticipate future disturbances. The first row of `md` specifies the n_{md} current disturbance values ($k = 1$), with other rows specifying disturbances for subsequent control intervals. If $N < p + 1$, the controller applies the last row for the remaining $p - N + 1$ steps.

For example, suppose $n_{md} = 2$ and $p = 6$. At a given control instant, the signal connected to the `md` 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 `MD` values are `2` and `5` at $k = 0$.
- The first `MD` remains at `2`, but the second increases gradually.
- The second `MD` becomes `8` at the beginning of the third step $k = 3$ in the prediction horizon.
- Both values remain constant at `2` and `8` respectively for steps 4–6 of the prediction horizon.

`mpcpreview` shows how to use `MD` previewing in a specific case.

For calculation details, see “MPC Modeling” and “QP Matrices”.

Add an inport (`ext.mv`) to which you connect a vector signal that contains the actual manipulated variables (MV) used in the plant. All candidate controllers use this signal to update their controller state estimates at each control interval. Using this inport improves state estimation accuracy when the MVs used in the plant differ from the MVs calculated by the block, for example due to signal saturation or an override condition.

For additional information, see the corresponding section of the MPC Controller block reference page.

Additional Outports (General Section)

You can configure several 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.

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`.)

Add an outport (`qp.status`) that allows you to monitor the status of QP solver for the active controller.

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 for this time interval.

The QP solver can 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. In this case, if the `Optimizer.UseSuboptimalSolution` property of the MPC controller is `false`, the block holds its `mv` output at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.
- `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. In this case, the block holds its `mv` output at the most recent successful solution.

- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem. In this case, the block holds its `mv` output at the most recent successful solution.

In a real-time application, you can use `qp.status` to set an alarm or take other special action.

Add an outport (`est.state`) for the active controller state estimates, `x[k|k]`, at each control instant. These estimates include the plant, disturbance, and noise model states.

Add an outport (`mv.seq`) that provides the predicted optimal MV adjustments (moves) over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_u matrix, where and n_u is the number of manipulated variables.

`mv.seq` contains the calculated optimal MV moves at time `k+i-1`, for `i = 1,...,p`. The first row of `mv.seq` is identical to the `mv` outport signal, which is the current MV adjustment applied at time `k`. Since the controller does not calculate optimal control moves at time `k+p`, the last row of `mv.seq` duplicates the previous row.

Add an outport (`x.seq`) that provides the predicted optimal state variable sequence over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_x matrix, where n_x is the number of states in the plant and unmeasured disturbance models (states from noise models are not included).

`x.seq` contains the calculated optimal state values at time `k+i`, for `i = 1,...,p`. The first row of `x.seq` contains the current states at time `k` as determined by state estimation.

Add an outport (`y.seq`) that provides the predicted optimal output variable sequence over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_y matrix, where and n_y is the number of outputs.

`y.seq` contains the calculated optimal output values at time `k+i-1`, for `i = 1,...,p+1`. The first row of `y.seq` contains the current outputs at time `k` based on the estimated states and measured disturbances; it is not the measured output at time `k`.

State Estimation (General Section)

Replace `mo` with the `x[k|k]` inport for custom state estimation as described in “Required Inports” on page 3-24. All candidate controllers must use the same state estimation option, either default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Constraints (Online Features Section)

At each control instant, the optional features described below apply to the active controller.

Add inport `umin` that you can connect to a run-time constraint signal for manipulated variable lower bounds. This signal is a vector with n_u finite values.

If a manipulated variable does not have a lower bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within the active controller.

Note You cannot specify time-varying constraints at run time using a matrix signal.

Add inport `umax` that you can connect to a run-time constraint signal for manipulated variable upper bounds. This signal is a vector with n_u finite values.

If a manipulated variable does not have an upper bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within the active controller.

Note You cannot specify time-varying constraints at run time using a matrix signal.

Add inport `ymin` that you can connect to a run-time constraint signal for output variable lower bounds. This signal is a vector with n_y finite values.

If an output variable does not have a lower bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within the active controller.

Note You cannot specify time-varying constraints at run time using a matrix signal.

Add inport `ymax` that you can connect to a run-time constraint signal for output variable upper bounds. This signal is a vector with n_y finite values.

If this parameter is not selected, the block uses the constant constraint values stored within the active controller.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

Weights (Online Features Section)

The optional inputs described below function as controller "tuning knobs." By default (or when a signal is unconnected), the stored tuning weights of the active controller apply.

When using these online tuning features, care must be taken to prevent an unexpected change in the active controller. Otherwise, settings intended for a particular candidate controller can instead retune another.

Add an inport (`y.wt`) that you can connect to a run-time output variable (OV) weight signal. This signal overrides the `Weights.OV` property of the `mpc` object, which establishes the relative importance of OV reference tracking.

To use the same tuning weights over the prediction horizon, connect `y.wt` to a vector signal with n_y elements, where n_y is the number of controlled outputs. Each element

specifies a nonnegative tuning weight for each controlled output variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `y.wt` to a matrix signal with n_y columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

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.

Add an inport (`u.wt`) that you can connect to a run-time manipulated variable (MV) weight signal. This signal overrides the `Weights.MV` property of the `mpc` object, which establishes the relative importance of MV target tracking.

To use the same tuning weights over the prediction horizon, connect `u.wt` to a vector signal with n_u elements, where n_u is the number of manipulated variables. Each element specifies a nonnegative tuning weight for each manipulated variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `u.wt` to a matrix signal with n_u columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

Add an inport (`du.wt`) that you can connect to a run-time manipulated variable (MV) rate weight signal. This signal overrides the `Weights.MVrate` property of the `mpc` object, which establishes the relative importance of MV changes.

To use the same tuning weights over the prediction horizon, connect `du.wt` to a vector signal with n_u elements, where n_u is the number of manipulated variables. Each element specifies a nonnegative tuning weight for each manipulated variable rate. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `du.wt` to a matrix signal with n_u columns and up to p rows, where p is the prediction horizon. Each row contains

the tuning weights for one prediction horizon step. If you specify fewer than *p* rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

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.

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 Section)

If you want one or more manipulated variables (MV) to track target values that change with time, use this option to add an `mv.target` inport. Connect this port to a target signal with 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 Section

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

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

For an example of double-precision and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**.

Use the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered

Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results .

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” (Simulink).

Compatibility Considerations

MPC Simulink block `mv.seq` output port signal dimensions have changed

Behavior changed in R2018b

The signal dimensions of the `mv.seq` output port of the Multiple MPC Controllers block have changed. Previously, this signal was a p-by-N_{mv} matrix, where p is the prediction horizon and N_{mv} is the number of manipulated variables. Now, `mv.seq` is a (p+1)-by-N_{mv} matrix, where row p+1 duplicates row p.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks
MPC Controller | Multiple Explicit MPC Controllers

Functions
`mpc` | `mpcmove` | `mpcstate`

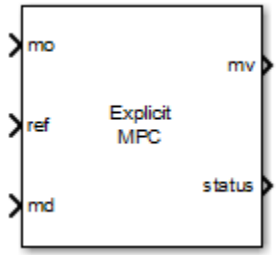
Topics

“Gain-Scheduled MPC”
“Design Workflow”
“Gain Scheduled Implicit and Explicit MPC Control of Mass-Spring System”
“Gain-Scheduled MPC Control of Nonlinear Chemical Reactor”
“Simulation and Code Generation Using Simulink Coder”
“Simulation and Structured Text Generation Using PLC Coder”

Introduced in R2008b

Explicit MPC Controller

Design and simulate explicit model predictive controller



Library

MPC Simulink Library

Description

The Explicit MPC Controller block uses the following input signals:

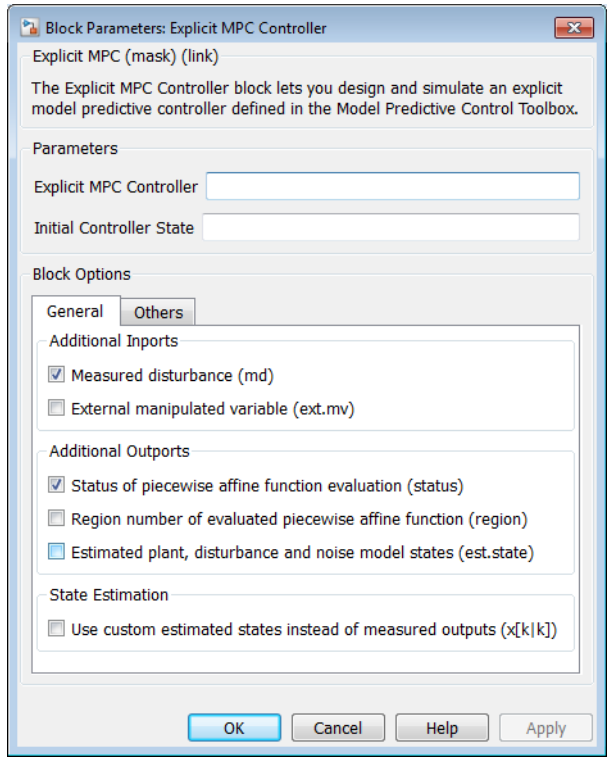
- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any

The key difference is that the Explicit MPC Controller block uses a table-lookup control law during each control interval rather than solving a quadratic program. The reduced online computational effort is advantageous in applications requiring a short control interval. The primary trade-off is a heavier offline computational effort 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.

The Explicit MPC Controller supports only a subset of optional MPC features, as outlined in the following table.

Supported Features	Unsupported Features
• 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	• Online tuning (penalty weight adjustments) • Online constraint adjustments • Online manipulated variable target adjustments • Reference and/or measured disturbance previewing

Dialog Box



The Explicit MPC Controller block has the following parameter groupings:

Parameters

An Explicit MPC controller on page 4-23 object containing the control law to be used. It must exist in the workspace. Use the `generateExplicitMPC` command to create this object.

An optional `mpcstate` object specifying the initial controller state. By default the block uses the `Model.Nominal` property of the controller object.

Required Inports

Measured output or **State estimate**

If your controller uses default state estimation, this inport is labeled `mo`. Connect this inport to the measured plant output signals. The MPC controller uses measured plant outputs to improve its state estimates.

To enable custom state estimation, in the **General** section, check **Use custom estimated states instead of measured outputs**. Checking this option changes the label on this inport to `x[k|k]`. Connect a signal that provides estimates of the controller state (plant, disturbance, and noise model states). Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables, where n_y is the total number of

outputs, including measured and unmeasured outputs. Since this block does not support reference previewing, `ref` cannot be defined as a matrix.

Required Outports

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables for controlling 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; that is, it is held at the previous successful result.

Additional Inports (General Section)

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.

Since this block does not support measured disturbance previewing, `md` cannot be defined as a matrix.

Add an inport (`ext.mv`) to which you connect a vector signal that contains the actual manipulated variables (MV) used in the plant. The controller uses this signal to update their controller state estimates at each control interval. Using this inport improves state estimation accuracy when the MVs used in the plant differ from the MVs calculated by the block, for example due to signal saturation or an override condition.

Note Using this option can cause an algebraic loop in the Simulink model, since there is direct feedthrough from the `ext.mv` inport to the `mv` outport. To prevent such algebraic loops, insert a Memory block or Unit Delay block.

For additional information, see the corresponding section of the MPC Controller block reference page.

Additional Outports (General Section)

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 control law parameters out of range.
- –1 — Undefined: Control law parameters were within the valid range but an extrapolation was necessary.

If `status` is either 0 or –1, the `mv` outport remains at the last known good value.

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.

Add an outport (`est.state`) for the controller state estimates, `x[k|k]`, at each control instant. These estimates include the plant, disturbance, and noise model states.

State Estimation (General Section)

Replace `mo` with the `x[k|k]` inport for custom state estimation as described in “Required Inports” on page 3-38.

Others Section

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

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

For an example of double-precision and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**.

Use the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results .

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” (Simulink).

Add an inport (`switch`) whose input specifies whether the controller evaluates its control law. If the input signal is zero, the controller behaves normally. If the input signal is nonzero, the Explicit MPC Controller block turns off controller evaluation. 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. However, the controller continues to update its internal state estimates in the usual way. Thus, it is ready to resume optimization calculations whenever the `switch` signal returns to zero. While controller evaluation is 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 the value it had when evaluation was disabled.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks

MPC Controller | Multiple Explicit MPC Controllers

Functions

`generateExplicitMPC` | `mpc` | `mpcmoveExplicit` | `mpcstate`

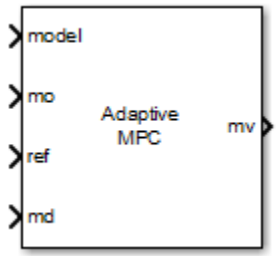
Topics

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

Introduced in R2014b

Adaptive MPC Controller

Design and simulate adaptive and time-varying model predictive controllers



Library

MPC Simulink Library

Description

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 use when computing the optimal plant manipulated variables `mv`. The linear prediction model can change at each control interval in response to changes in the real plant at run time. The prediction model can represent a single LTI plant used for all prediction steps (adaptive MPC mode) or an array of LTI plants for different prediction steps (time-varying MPC mode). Two common ways to modify this model are as follows:

- Given a nonlinear plant model, linearize it at the current operating point.
- Use plant data to estimate parameters in an empirical linear-time-varying (LTV) model.

By default, the block estimates its prediction model states. Since the prediction model parameters change at run time, the static Kalman filter used in the MPC Controller block

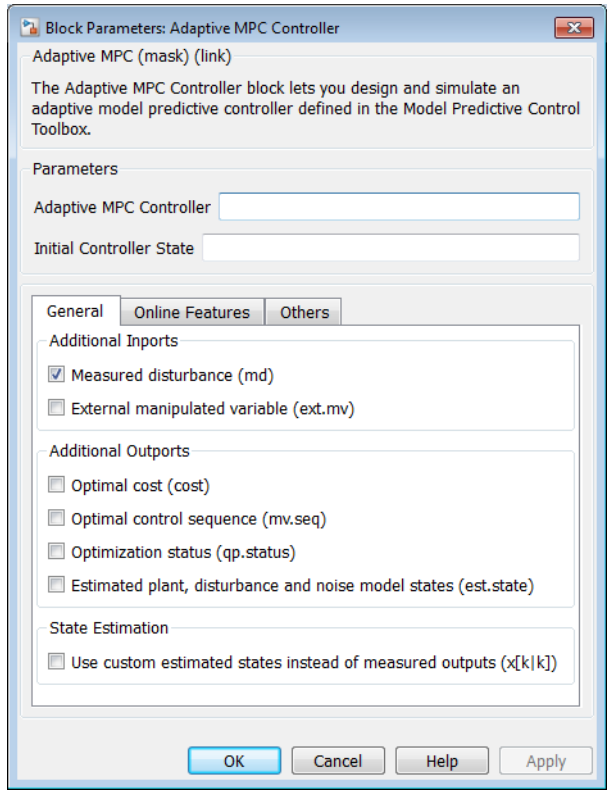
is inappropriate. Instead, the Adaptive MPC Controller block uses a linear-time-varying Kalman filter (LTVKF). For more information, see “Adaptive MPC”.

In all other ways, the Adaptive MPC Controller block mimics the MPC Controller block. Since the adaptive version involves additional overhead, use the MPC Controller block unless you need to control a nonlinear plant across a wide range of operating conditions where plant dynamics vary significantly.

Both the Adaptive MPC Controller block and the Multiple MPC Controllers block enable your control system to adapt to changing operating conditions at run time. The following table lists the advantages of using each block.

Block	Adaptive MPC Controller	Multiple MPC Controllers
Adaptation approach	Update prediction model for a single controller as operating conditions change	Switch between multiple controllers designed for different operating regions
Advantages	• Only need to design a single controller offline • Less run-time computational effort and smaller memory footprint • More robust to real-life changes in plant conditions	• No need for online estimation of plant model • Controllers can have different sample time, horizons, and weights • Prediction models can have different orders or time domains • Finite set of candidate controllers can be tested thoroughly

Dialog Box



The Adaptive MPC Controller block has the following parameter groupings:

- “Parameters” on page 3-46
- “Required Inports” on page 3-46
- “Required Outports” on page 3-49
- “Additional Inports (General Section)” on page 3-50
- “Additional Outports (General Section)” on page 3-51
- “State Estimation (General Section)” on page 3-53
- “Prediction Model (Online Features Section)” on page 3-53
- “Constraints (Online Features Section)” on page 3-53

Parameters

An `mpc` controller object designed at the nominal operating point. At run time, the controller replaces the original prediction model (`A`, `B`, `C`, `D`) and nominal values (`U`, `Y`, `X`, `DX`) with the data specified in the `model` inport at each control instant.

By default, the block assumes all other controller object properties (for example tuning weights, constraints) are constant. You can override this assumption using the options in the **Online Features** section.

The following restrictions apply to the `mpc` controller object:

- It must exist in the MATLAB 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. The dimensions of the `A`, `B`, `C`, and `D` matrices in the prediction determine the dimensions required by the `model` inport signal.

Specifies the initial controller state. If this parameter is left blank, the block uses the nominal values that are defined in the `Model.Nominal` property of the `mpc` object. To override the default, create an `mpcstate` object in your workspace, and enter its name in the field.

Required Inports

Model

Connect a bus signal to the `model` inport. This signal modifies the controller object `Model.Plant` and `Model.Nominal` properties at the beginning of each control interval.

The Adaptive MPC Controller requires `Model.Plant` to be an LTI discrete-time state-space object with no delays. The following command extracts the state-space matrices comprising such a model:

```
[A,B,C,D] = ssdata(MPCobj.Model.Plant)
```

The purpose of the `model` 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`.

When operating in:

- Adaptive MPC mode, 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_u matrix signal, where n_u 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_u matrix signal.
  - `X` — Vector signal of length n_x, replacing the controller `Model.Nominal.X` property.
  - `Y` — Vector signal of length n_y, replacing the controller `Model.Nominal.Y` property.
  - `U` — Vector signal of length n_u, replacing the controller `Model.Nominal.U` property.
  - `DX` — Vector signal of length n_x, replacing the controller `Model.Nominal.DX` property. It must be appropriate for use with a discrete-time model of the assumed control interval. For more information, see “Adaptive MPC”.
- Time-varying MPC mode, the bus you connect to the `model` inport must contain the following 3–dimensional bus signals:
  - `A` — n_x-by-n_x-by-(p+1) matrix signal
  - `B` — n_x-by-n_u-by-(p+1) matrix signal
  - `C` — n_y-by-n_x-by-(p+1)
  - `D` — n_y-by-n_u-by-(p+1) matrix signal
  - `X` — n_x-by-(p+1) matrix signal
  - `Y` — n_y-by-(p+1) matrix signal

- `U` — n_u-by-(p+1) matrix signal
- `DX` — n_x-by-(p+1) matrix signal

Here, p is the controller prediction horizon. For each signal, specify p+1 values representing the model and nominal conditions at each step of the prediction horizon. For more information, see “Time-Varying MPC”.

One way to form the bus is to use a Bus Creator block.

Measured output or **State estimate**

If your controller uses default state estimation, this inport is labeled `mo`. Connect this inport to the measured plant output signals. The MPC controller uses measured plant outputs to improve its state estimates.

To enable custom state estimation, in the **General** section, check **Use custom estimated states instead of measured outputs**. Checking this option changes the label on this inport to `x[k|k]`. Connect a signal that provides estimates of the controller state (plant, disturbance, and noise model states). Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

Reference

The `ref` dimension must not change from one control instant to the next. Each element must be a real number.

When `ref` is a 1-by-n_y signal, where n_y is the number of outputs, there is no reference signal previewing. The controller applies the current reference values across the prediction horizon.

To use signal previewing, specify `ref` as an N-by-n_y signal, where N is the number of

time steps for which you are specifying reference values. Here, $1 < N \le p$, and p is the prediction horizon. Previewing usually improves performance, since the controller can anticipate future reference signal changes. The first row of `ref` specifies the n_y references for the first step in the prediction horizon (at the next control interval k = 1), and so on for N steps. If $N < p$, the last row designates constant reference values for the remaining p - N steps.

For example, suppose n_y = 2 and p = 6. At a given control instant, the signal connected to the `ref` 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

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables for controlling the plant. At each control instant, the controller updates its `mv` outport by solving a quadratic programming problem using either the default KWIK solver or a custom QP solver. For more information, see “QP Solver”.

If the controller detects an infeasible optimization problem or encounters numerical difficulties in solving an ill-conditioned optimization problem, `mv` remains at its most recent successful solution; that is, the controller output freezes.

Otherwise, if the optimization problem is feasible and the solver reaches the specified maximum number of iterations without finding an optimal solution, `mv`:

- Remains at its most recent successful solution if the `Optimizer.UseSuboptimalSolution` property of the controller is `false`.
- Is the suboptimal solution reached after the final iteration if the `Optimizer.UseSuboptimalSolution` property of the controller is `true`. For more information, see “Suboptimal QP Solution”.

Additional Inports (General Section)

Add an inport (`md`) to which you connect a measured disturbance signal. The number of measured disturbances defined for your controller, $n_{md} \geq 1$, must match the dimensions of the connected disturbance signal.

The number of measured disturbances must not change from one control instant to the next, and each disturbance value must be a real number.

When `md` is a 1-by-n_{md} signal, there is no measured disturbance previewing. The controller applies the current disturbance values across the prediction horizon.

To use disturbance previewing, specify `md` as an N-by-n_{md} signal, where N is the number of time steps for which the measured disturbances are known. Here, $1 < N \leq p+1$, and p is the prediction horizon. Previewing usually improves performance, since the controller can anticipate future disturbances. The first row of `md` specifies the n_{md} current disturbance values ($k = 1$), with other rows specifying disturbances for subsequent control intervals. If $N < p + 1$, the controller applies the last row for the remaining $p - N + 1$ steps.

For example, suppose $n_{md} = 2$ and $p = 6$. At a given control instant, the signal connected to the `md` 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 `MD` values are `2` and `5` at $k = 0$.
- The first `MD` remains at `2`, but the second increases gradually.
- The second `MD` becomes `8` at the beginning of the third step $k = 3$ in the prediction horizon.
- Both values remain constant at `2` and `8` respectively for steps 4–6 of the prediction horizon.

`mpcpreview` shows how to use `MD` previewing in a specific case.

For calculation details, see “MPC Modeling” and “QP Matrices”.

Add an inport (`ext.mv`) to which you connect a vector signal that contains the actual manipulated variables (MV) used in the plant. The controller uses this signal to update their controller state estimates at each control interval. Using this inport improves state estimation accuracy when the MVs used in the plant differ from the MVs calculated by the block, for example due to signal saturation or an override condition.

Note Using this option can cause an algebraic loop in the Simulink model, since there is direct feedthrough from the `ext.mv` inport to the `mv` outport. To prevent such algebraic loops, insert a Memory block or Unit Delay block.

For additional information, see the corresponding section of the MPC Controller block reference page.

Additional Outports (General Section)

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`.)

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 for this time interval.

The QP solver can 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. In this case, if the `Optimizer.UseSuboptimalSolution` property of the MPC controller is `false`, the block holds its `mv` output at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.
- `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. In this case, the block holds its `mv` output at the most recent successful solution.

- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem. In this case, the block holds its `mv` output at the most recent successful solution.

In a real-time application, you can use `qp.status` to set an alarm or take other special action.

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.

Add an outport (`mv.seq`) that provides the predicted optimal MV adjustments (moves) over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_u matrix, where and n_u is the number of manipulated variables.

`mv.seq` contains the calculated optimal MV moves at time `k+i-1`, for `i = 1,...,p`. The first row of `mv.seq` is identical to the `mv` outport signal, which is the current MV adjustment applied at time `k`. Since the controller does not calculate optimal control moves at time `k+p`, the last row of `mv.seq` duplicates the previous row.

Add an outport (`x.seq`) that provides the predicted optimal state variable sequence over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_x matrix, where n_x is the number of states in the plant and unmeasured disturbance models (states from noise models are not included).

`x.seq` contains the calculated optimal state values at time `k+i`, for `i = 1,...,p`. The first row of `x.seq` contains the current states at time `k` as determined by state estimation.

Add an outport (`y.seq`) that provides the predicted optimal output variable sequence over the entire prediction horizon from `k` to `k+p`, where `k` is the current time and p is the prediction horizon. This signal is a (p+1)-by-n_y matrix, where and n_y is the number of outputs.

`y.seq` contains the calculated optimal output values at time `k+i-1`, for `i = 1,...,p+1`. The first row of `y.seq` contains the current outputs at time `k` based on the estimated states and measured disturbances; it is not the measured output at time `k`.

State Estimation (General Section)

Replace `mo` with the `x[k|k]` inport for custom state estimation as described in “Required Inports” on page 3-46.

Prediction Model (Online Features Section)

To operate your controller in time-varying MPC mode, select this option. When operating in this mode, connect a 3–dimensional bus signal to the `model` inport as described in “Required Inports” on page 3-46.

For an example, see “Time-Varying MPC Control of a Time-Varying Plant”.

Constraints (Online Features Section)

Add inport `umin` that you can connect to a run-time constraint signal for manipulated variable lower bounds. This signal is a vector with n_u finite values. The `i`th element of `umin` replaces the `ManipulatedVariables(i).Min` property of the controller at run time.

If a manipulated variable does not have a lower bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `ManipulatedVariables(i).Min` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `umin` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inport `umax` that you can connect to a run-time constraint signal for manipulated variable upper bounds. This signal is a vector with n_u finite values. The `i`th element of `umax` replaces the `ManipulatedVariables(i).Max` property of the controller at run time.

If a manipulated variable does not have an upper bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `ManipulatedVariables(i).Max` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `umax` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inport `ymin` that you can connect to a run-time constraint signal for output variable lower bounds. This signal is a vector with n_y finite values. The `i`th element of `ymin` replaces the `OutputVariables(i).Min` property of the controller at run time.

If an output variable does not have a lower bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `OutputVariables(i).Min` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `ymin` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inport `ymax` that you can connect to a run-time constraint signal for output variable upper bounds. This signal is a vector with n_y finite values. The `i`th element of `ymax` replaces the `OutputVariables(i).Max` property of the controller at run time.

If an output variable does not have an upper bound specified in the controller object, then the corresponding connected signal value is ignored.

If this parameter is not selected, the block uses the constant constraint values stored within its `mpc` object.

Note You cannot specify time-varying constraints at run time using a matrix signal.

If the `OutputVariables(i).Max` property of the controller is specified as a vector (that is, the constraint varies over the prediction horizon), the `i`th element of `ymax` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

Add inports `E`, `F`, `G`, and `S` to the block that you can connect to the following run-time custom constraint matrix signals:

- `E` — Manipulated variable constraint matrix with size n_c-by-n_u, where n_c is the number of custom constraints
- `F` — Controlled output constraint matrix with size n_c-by-n_y
- `G` — Custom constraint matrix with size 1-by-n_c
- `S` — Measured disturbance constraint matrix, with size n_c-by-n_v, where n_v is the number of measured disturbances. `S` is added only if the `mpc` object has measured disturbances.

These constraints replace the custom constraints previously set using `setconstraint`.

If you define `E`, `F`, `G`, or `S` in the `mpc` object, you must connect a signal to the corresponding inport, and that signal must have the same dimensions as the array specified in the controller. If an array is not defined in the controller object, use a zero matrix with the correct size.

The custom constraints are of the form `E`u + `F`y + `S`v <= `G`, where:

- u is a vector of manipulated variable values.
- y is a vector of predicted plant output values.
- v is a vector of measured plant disturbance input values.

For more information, see “Constraints on Linear Combinations of Inputs and Outputs”.

Weights (Online Features Section)

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.

Add an inport (`y.wt`) that you can connect to a run-time output variable (OV) weight signal. This signal overrides the `Weights.OV` property of the `mpc` object, which establishes the relative importance of OV reference tracking.

To use the same tuning weights over the prediction horizon, connect `y.wt` to a vector signal with n_y elements, where n_y is the number of controlled outputs. Each element specifies a nonnegative tuning weight for each controlled output variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `y.wt` to a matrix signal with n_y columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

If you do not connect a signal to the `y.wt` inport, the block uses the OV weights specified in your `mpc` object.

Add an inport (`u.wt`) that you can connect to a run-time manipulated variable (MV) weight signal. This signal overrides the `Weights.MV` property of the `mpc` object, which establishes the relative importance of MV target tracking.

To use the same tuning weights over the prediction horizon, connect `u.wt` to a vector signal with n_u elements, where n_u is the number of manipulated variables. Each element specifies a nonnegative tuning weight for each manipulated variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `u.wt` to a matrix signal with n_u columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

If you do not connect a signal to the `u.wt` inport, the block uses the MV weights specified in your `mpc` object.

Add an inport (`du.wt`) that you can connect to a run-time manipulated variable (MV) rate weight signal. This signal overrides the `Weights.MVrate` property of the `mpc` object, which establishes the relative importance of MV changes.

To use the same tuning weights over the prediction horizon, connect `du.wt` to a vector signal with n_u elements, where n_u is the number of manipulated variables. Each element specifies a nonnegative tuning weight for each manipulated variable rate. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon, connect `du.wt` to a matrix signal with n_u columns and up to p rows, where p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

If you do not connect a signal to the `du.wt` inport, the block uses the MV rate weights specified in your `mpc` object.

Add an inport (`ecr.wt`), for a scalar nonnegative signal that overrides the `MPCobj.Weights.ECR` property of the `mpc` controller. 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 Section)

If you want one or more manipulated variables (MV) to track target values that change with time, use this option to add an `mv.target` inport. Connect this port to a target signal with 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 Section

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

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

For an example of double-precision and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**.

Use the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results .

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” (Simulink).

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 is 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. However, the controller continues to update its internal state estimates in the usual way. Thus, it is ready to resume optimization calculations whenever the `switch` signal returns to zero. While controller optimization is 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 the value it had when optimization was disabled.

Compatibility Considerations

MPC Simulink block `mv.seq` output port signal dimensions have changed

Behavior changed in R2018b

The signal dimensions of the `mv.seq` output port of the Adaptive MPC Controller block have changed. Previously, this signal was a p-by-N_{mv} matrix, where p is the prediction horizon and N_{mv} is the number of manipulated variables. Now, `mv.seq` is a (p+1)-by-N_{mv} matrix, where row p+1 duplicates row p.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks
MPC Controller | Multiple MPC Controllers

Functions
`mpc` | `mpcmoveAdaptive` | `mpcstate`

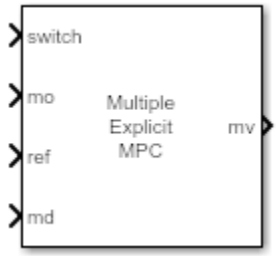
Topics

“Adaptive MPC”
“Time-Varying MPC”
“Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization”
“Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation”
“Time-Varying MPC Control of a Time-Varying Plant”

Introduced in R2014b

Multiple Explicit MPC Controllers

Simulate switching between multiple explicit MPC controllers



Library

MPC Simulink Library

Description

The Multiple Explicit MPC Controllers block uses the following input signals:

- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any
- Switching signal (`switch`)

The switching signal selects the *active controller* from among a list of two or more candidate controllers. However, for the Multiple Explicit MPC Controllers block, the candidates are explicit MPC controllers. These controllers reduce online computational effort by using a table-lookup control law during each control interval rather than solving a quadratic program. For more information, see Explicit MPC Controller.

The Multiple Explicit MPC Controllers block enables you to transition between multiple explicit MPC controllers in real time based on the current operating conditions. Typically, you design each controller for a particular region of the operating space. Using available

measurements, you detect the current operating region and select the appropriate active controller via the `switch` inport.

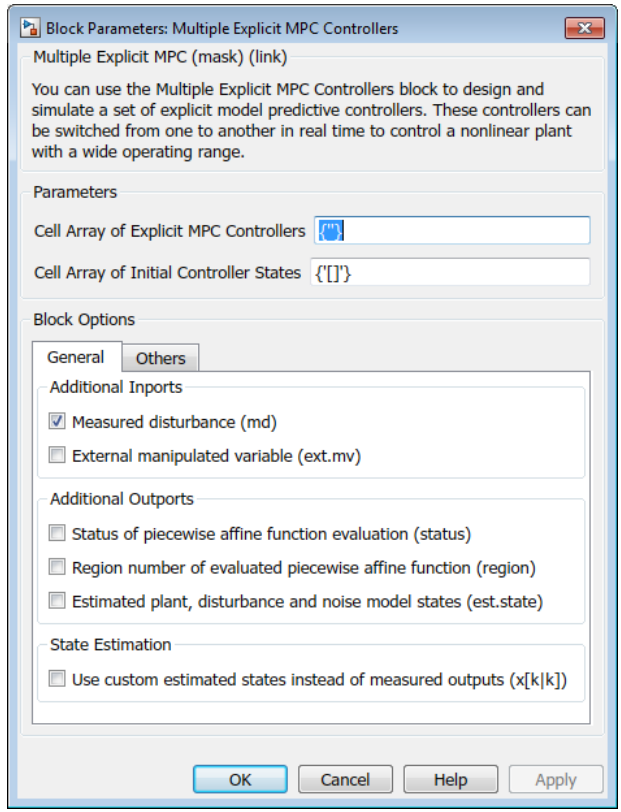
To improve efficiency, inactive controllers do not evaluate their control law. However, to provide bumpless transfer between controllers, the inactive controllers continue to perform state estimation.

Like for the Multiple MPC Controllers block, you cannot disable evaluation for the Multiple Explicit MPC Controllers. One controller must always be active.

Like the Explicit MPC Controller block, the Multiple Explicit MPC Controllers supports only a subset of optional MPC features, as outlined in the following table.

Supported Features	Unsupported Features
• 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	• Online tuning (penalty weight adjustments) • Online constraint adjustments • Online manipulated variable target adjustments • Reference and/or measured disturbance previewing

Dialog Box



The Multiple Explicit MPC Controller block has the following parameter groupings:

- “Parameters” on page 3-64
- “Required Inports” on page 3-64
- “Required Outports” on page 3-65
- “Additional Inports (General Section)” on page 3-65
- “Additional Outports (General Section)” on page 3-66
- “State Estimation (General Section)” on page 3-66
- “Others Section” on page 3-66

Parameters

Cell Array of Explicit MPC Controllers

Candidate controllers, specified as:

- A cell array of Explicit MPC controller on page 4-23 objects. Use the `generateExplicitMPC` command to create these objects.
- A cell array of character vectors, where each element is the name of an explicit MPC controller object in the MATLAB workspace.

The specified array must contain at least two candidate controllers. The first entry in the cell array is the controller that corresponds to a switch input value of 1, the second corresponds to a switch input value of 2, and so on.

Cell Array of Initial Controller States

Optional initial states for each candidate controller, specified as:

- A cell array of `mpcstate` objects.
- A cell array of character vectors, where each element is the name of an `mpcstate` object in the MATLAB workspace.
- `{[],[],...}` or `{'[]','[]',...}` — Use the nominal condition defined in `Model.Nominal` as the initial state for each controller.

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 specified candidate controllers. At each control instant, this signal designates the active controller. A switch value of 1 corresponds to the first entry in the cell array of candidate controllers, a value of 2 corresponds to the second controller, and so on.

If the `switch` signal is outside of the range 1 and n_c, the previous controller output is retained.

Measured output or **State estimate**

If candidate controllers use default state estimation, this inport is labeled `mo`. Connect this inport to the measured plant output signals.

If your candidate controllers use custom state estimation, check **Use custom estimated states instead of measured outputs** in the **General** section. Checking

this 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, either 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, where n_y is the total number of outputs, including measured and unmeasured outputs. Since this block does not support reference previewing, `ref` cannot be defined as a matrix.

Required Outports

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables for controlling the plant. The active controller updates its manipulated variable output at each control instant using the control law contained in its explicit MPC controller object. If the control law evaluation fails, this signal is unchanged; that is, it is held at the previous successful result. The Multiple Explicit MPC Controller block passes the output of the active controller to the `mv` outport.

Additional Inports (General Section)

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.

Since this block does not support measured disturbance previewing, `md` cannot be defined as a matrix.

Add an inport (`ext.mv`) to which you connect a vector signal that contains the actual manipulated variables (MV) used in the plant. All candidate controllers use this signal to update their controller state estimates at each control interval. Using this inport improves state estimation accuracy when the MVs used in the plant differ from the MVs calculated by the block, for example due to signal saturation or an override condition.

For additional information, see the corresponding section of the MPC Controller block reference page.

Additional Outports (General Section)

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 control law parameters out of range.
- –1 — Undefined: Control law parameters were within the valid range but an extrapolation was necessary.

If `status` is either 0 or –1, the `mv` outport remains at the last known good value.

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.

Add an outport (`est.state`) for the controller state estimates, `x[k|k]`, at each control instant. These estimates include the plant, disturbance, and noise model states.

State Estimation (General Section)

Replace `mo` with the `x[k|k]` inport for custom state estimation as described in “Required Inports” on page 3-64. All candidate controllers must use the same state estimation option, either default or custom. When you use custom state estimation, all candidate controllers must have the same dimension.

Others Section

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

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

For an example of double-precision and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**.

Use the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results .

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” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks
Explicit MPC Controller | Multiple MPC Controllers

Functions
`mpc` | `mpcmove` | `mpcstate`

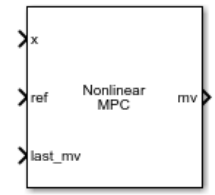
Topics

“Simulation and Code Generation Using Simulink Coder”
“Simulation and Structured Text Generation Using PLC Coder”

Introduced in R2016b

Nonlinear MPC Controller

Simulate nonlinear model predictive controllers
Library: Model Predictive Control Toolbox



Description

The Nonlinear MPC Controller block simulates a nonlinear model predictive controller. At each control interval, the block computes optimal control moves by solving a nonlinear programming problem. For more information on nonlinear MPC, see “Nonlinear MPC”.

To use this block, you must first create an `nlmpc` object in the MATLAB workspace.

Limitations

- None of the Nonlinear MPC Controller block parameters are tunable.

Ports

Input

Required Inputs

x — input
vector

Current prediction model states, specified as a vector signal of length N_x, where N_x is the number of prediction model states. Since the nonlinear MPC controller does not perform

state estimation, you must either measure or estimate the current prediction model states at each control interval.

`ref` — Model output reference values

row vector | matrix

Plant output reference values, specified as a row vector signal or matrix signal.

To use the same reference values across the prediction horizon, connect **ref** to a row vector signal with N_Y elements, where N_y is the number of output variables. Each element specifies the reference for an output variable.

To vary the references over the prediction horizon (previewing) from time k to time $k+p$-1, connect **ref** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the references for one prediction horizon step. If you specify fewer than p rows, the final references are used for the remaining steps of the prediction horizon.

`last_mv` — Control signals used in plant at previous control interval

vector

Control signals used in plant at previous control interval, specified as a vector signal of lengthN_{mv}, where N_{mv} is the number of manipulated variables.

Note Connect **last_mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode; that is, the controller output is not driving the plant, then feeding the actual control signal to **last_mv** can help achieve bumpless transfer when the controller is switched back online.

Additional Inputs

`md` — input

row vector | matrix

If your controller model has measured disturbances, enable this input port, and connect a row vector or matrix signal. If your controller has measured disturbances, you must enable this port.

To use the same measured disturbance values across the prediction horizon, connect **md** to a row vector signal with N_{md} elements, where N_{md} is the number of manipulated variables. Each element specifies the value for a measured disturbance.

To vary the disturbances over the prediction horizon (previewing) from time k to time $k+p$-1, connect **md** to a matrix signal with N_{md} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbances for one prediction horizon step. If you specify fewer than p rows, the final disturbances are used for the remaining steps of the prediction horizon.

`params` — Optional parameters

bus

If your controller uses optional parameters in its prediction model, custom cost function, or custom constraint functions, enable this input port, and connect a parameter bus signal with N_p elements, where N_p is the number of parameters. For more information on creating a parameter bus signal, see `createParameterBus`. The controller, passes these parameters to its model functions, cost function, constraint functions, and Jacobian functions.

If your controller does not use optional parameters, you must disable **params**.

Dependencies

To enable this port, select the **Model parameters** parameter.

`mv.target` — Manipulated variable targets

row vector | array

To specify manipulated variable targets, enable this input port, and connect a row vector or matrix signal.

To use the same manipulated variable targets across the prediction horizon, connect **mv.target** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies the target for a manipulated variable.

To vary the targets over the prediction horizon (previewing) from time k to time $k+p$-1, connect **mv.target** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the targets for one prediction horizon step. If you specify fewer than p rows, the final targets are used for the remaining steps of the prediction horizon.

Dependencies

To enable this port, select the **MV targets** parameter.

Online Constraints

`y.min` — Minimum output variable constraints

row vector | matrix

To specify run-time minimum output variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `OutputVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **y.min** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies the lower bound for an output variable.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, connect **y.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower OV limits** parameter.

`y.max` — Maximum output variable constraints

row vector | matrix

To specify run-time maximum output variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `OutputVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **y.max** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies the upper bound for an output variable.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, connect **y.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper OV limits** parameter.

`mv.min` — Minimum manipulated variable constraints

row vector | matrix

To specify run-time minimum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **mv.min** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the lower bound for a manipulated variable.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, connect **mv.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower MV limits** parameter.

`mv.max` — Maximum manipulated variable constraints

row vector | matrix

To specify run-time maximum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **mv.max** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the upper bound for a manipulated variable.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, connect **mv.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper MV limits** parameter.

`dmv.min` — Minimum manipulated variable rate constraints

row vector | matrix

To specify run-time minimum manipulated variable rate constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariable.RateMin` property of its controller object. **dmv.min** bounds must be nonpositive.

To use the same bounds over the prediction horizon, connect **dmv.min** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the lower bound for a manipulated variable rate of change.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, connect **dmv.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower MVRate limits** parameter.

`dmv.max` — Maximum manipulated variable rate constraints

row vector | matrix

To specify run-time maximum manipulated variable rate constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.RateMax` property of its controller object. **dmv.max** bounds must be nonnegative.

To use the same bounds over the prediction horizon, connect **dmv.max** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the upper bound for a manipulated variable rate of change.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, connect **dmv.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper MVRate limits** parameter.

`x.min` — Minimum state constraints

row vector | matrix

To specify run-time minimum state constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `States.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **x.min** to a row vector signal with N_x elements, where N_x is the number of outputs. Each element specifies the lower bound for a state.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, connect **x.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower state limits** parameter.

`x.max` — Maximum state constraints

row vector | matrix

To specify run-time maximum state constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `States.Max` property of its controller object.

To use the same bounds over the prediction horizon, connect **x.max** to a row vector signal with N_x elements, where N_x is the number of outputs. Each element specifies the upper bound for a state.

To vary the bounds over the prediction horizon from time k to time $k+p-1$, connect **x.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper state limits** parameter.

Online Tuning Weights

`y.wt` — Output variable tuning weights

row vector | matrix

To specify run-time output variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.OutputVariables` property of its controller object. These tuning weights penalize deviations from output references.

To use the same tuning weights over the prediction horizon, connect **y.wt** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies a nonnegative tuning weight for an output variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, connect **y.wt** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

Dependencies

To enable this port, select the **OV weights** parameter.

`mv.wt` — Manipulated variable tuning weights

row vector | matrix

To specify run-time manipulated variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariables` property of its controller object. These tuning weights penalize deviations from MV targets.

To use the same tuning weights over the prediction horizon, connect **mv.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, connect **mv.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

Dependencies

To enable this port, select the **MV weights** parameter.

`dmv.wt` — Manipulated variable rate tuning weights

row vector | matrix

To specify run-time manipulated variable rate tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariablesRate` property of its controller object. These tuning weights penalize large changes in control moves.

To use the same tuning weights over the prediction horizon, connect **dmv.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable rate. For more information on specifying tuning weights, see “Tune Weights”.

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, connect **dmv.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see “Time-Varying Weights and Constraints”.

Dependencies

To enable this port, select the **MVRate weights** parameter.

`ecr.wt` — Slack variable tuning weight

scalar

To specify a run-time slack variable tuning weight, enable this input port and connect a scalar signal. If this port is disabled, the block uses the tuning weight specified in the `Weights.ECR` property of its controller object.

The slack variable tuning weight 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.

Dependencies

To enable this port, select the **ECR weight** parameter.

Initial Guesses

`mv.init` — Initial guesses for the optimal manipulated variable solutions

vector | matrix

To specify initial guesses for the optimal manipulated variable solutions, enable this input port. If this port is disabled, the block uses the optimal control sequences calculated in the previous control interval as initial guesses.

To use the same initial guesses over the prediction horizon, connect **mv.init** to a vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies the initial guess for a manipulated variable.

To vary the initial guesses over the prediction horizon from time k to time $k+p-1$, connect **mv.init** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the guesses in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Initial guess** parameter.

`x.init` — Initial guesses for the optimal state variable solutions

vector | matrix

To specify initial guesses for the optimal state solutions, enable this input port. If this port is disabled, the block uses the optimal state sequences calculated in the previous control interval as intiial guesses.

To use the same initial guesses over the prediction horizon, connect **x.init** to a vector signal with N_x elements, where N_x is the number of states. Each element specifies the initial guess for a state.

To vary the initial guesses over the prediction horizon from time k to time $k+p-1$, connect **x.init** to a matrix signal with N_x columns and up to p rows. Here, k is the current time and

p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than *p* rows, the guesses in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Initial guess** parameter.

`e.init` — Initial guess for the slack variable at the solution

nonnegative scalar

To specify an initial guess for the slack variable at the solution, enable this input port and connect a nonnegative scalar signal. If this port is disabled, the block uses an initial guess of `0`.

Dependencies

To enable this port, select the **Initial guess** parameter.

Output

Required Output

`mv` — Optimal manipulated variable control action

column vector

Optimal manipulated variable control action, output as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the solver converges to a local optimum solution (**nlp.status** is positive), then **mv** contains the optimal solution.

If the solver reaches the maximum number of iterations without finding an optimal solution (**nlp.status** is zero) and the `Optimization.UseSuboptimalSolution` property of the controller is:

- `true`, then **mv** contains the suboptimal solution
- `false`, then **mv** is the same as **last_mv**

If the solver fails (**nlp.status** is negative), then **mv** is the same as **last_mv**.

Additional Outputs

`cost` — Objective function cost

nonnegative scalar

Objective function cost, output as a nonnegative scalar signal. The cost quantifies the degree to which the controller has achieved its objectives.

The cost value is only meaningful when the **nlp.status** output is nonnegative.

Dependencies

To enable this port, select the **Optimal cost** parameter.

`slack` — Slack variable

0 | nonnegative scalar

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

- ε = 0 — All soft constraints are satisfied over 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 the ECR values for each constraint).

Dependencies

To enable this port, select the **Slack variable** parameter.

`nlp.status` — Optimization status

scalar

Optimization status, output as one of the following:

- Positive Integer — Solver converged to an optimal solution
- `0` — Maximum number of iterations reached without converging to an optimal solution
- Negative integer — Solver failed

Dependencies

To enable this port, select the **Optimization status** parameter.

Optimal Sequences

`mv.seq` — Optimal manipulated variable sequence

matrix

Optimal manipulated variable sequence, returned as a matrix signal with p+1 rows and N_{mv} columns, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

The ith row of **mv.seq** contains the calculated optimal manipulated variable values at time k+i-1, for i = 1, ..., p+1, where k is the current time. The first row of **mv.seq** contains the current manipulated variables values (output **mv**). Since the controller does not calculate optimal control moves at time k+p, the final two rows of **mv.seq** are identical.

Dependencies

To enable this port, select the **Optimal control sequence** parameter.

`x.seq` — Optimal prediction model state sequence

matrix

Optimal prediction model state sequence, returned as a matrix signal with p+1 rows and N_x columns, where p is the prediction horizon and N_x is the number of states.

The ith row of **x.seq** contains the calculated state values at time k+i-1, for i = 1, ..., p+1, where k is the current time. The first row of **x.seq** contains the current state values (output **x**).

Dependencies

To enable this port, select the **Optimal state sequence** parameter.

`y.seq` — Optimal output variable sequence

matrix

Optimal output variable sequence, returned as a matrix signal with p+1 rows and N_y columns, where p is the prediction horizon and N_y is the number of output variables.

The ith row of **y.sseq** contains the calculated state values at time k+i-1, for i = 1, ..., p+1, where k is the current time. The first row of **y.seq** is computed based on the current states (output **x**) and the current measured disturbances (first row of input **md**).

Dependencies

To enable this port, select the **Optimal output sequence** parameter.

Parameters

`Nonlinear MPC Controller` — Parameter

`nlmpc` object name

You must provide an `nlmpc` object that defines a nonlinear MPC controller. To do so, enter the name of an `nlmpc` object in the MATLAB workspace.

Programmatic Use

Block Parameter: `nlmpcobj`
Type: character vector
Values: name of `nlmpc` object
Default: `''`

`Use prediction model sample time` — Parameter

on (default) | off

Select this parameter to run the controller using the same sample time as its prediction model. To use a different controller sample time, clear this parameter, and specify the sample time using the **Make block run at a different sample time** parameter.

Programmatic Use

Block Parameter: `UseObjectTs`
Type: character vector
Values: `'on'`, `'off'`
Default: `'on'`

`Make block run at a different sample time` — Controller sample time

positive finite scalar

Specify this parameter to run the controller using a different sample time from its prediction model.

Dependencies

To enable this parameter, clear the **Use prediction model sample time** parameter.

Programmatic Use
Block Parameter: `TsControl`
Type: character vector
Values: positive finite scalar
Default: `''`

General Tab

`Measured disturbances` — Add measured disturbance input port

off (default) | on

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: `md_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Model parameters` — Add model parameters input port

off (default) | on

If your controller uses optional parameters, you must select this parameter to add the **params** output port to the block.

For more information on creating a parameter bus signal, see `createParameterBus`.

Programmatic Use
Block Parameter: `param_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Optimal cost` — Odd optimal cost output port

off (default) | on

Select this parameter to add the **cost** output port to the block.

Programmatic Use
Block Parameter: `cost_enabled`
Type: character vector

Values: `'off'`, `'on'`
Default: `'off'`

`Optimal control sequence` — Add optimal control sequence output port
off (default) | on

Select this parameter to add the **mv.seq** output port to the block.

Programmatic Use
Block Parameter: `mvseq_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Optimal state sequence` — Add optimal state sequence output port
off (default) | on

Select this parameter to add the **x.seq** output port to the block.

Programmatic Use
Block Parameter: `stateseq_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Optimal output sequence` — Add optimal output sequence output port
off (default) | on

Select this parameter to add the **y.seq** output port to the block.

Programmatic Use
Block Parameter: `ovseq_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Slack variable` — Add slack variable output port
off (default) | on

Select this parameter to add the **slack** output port to the block.

Programmatic Use
Block Parameter: `slack_enabled`

Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Optimization status` — Add optimization status output port
off (default) | on

Select this parameter to add the **nlp.status** output port to the block.

Programmatic Use
Block Parameter: `status_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

Online Features Tab

`Lower OV limits` — Add minimum OV constraint input port
off (default) | on

Select this parameter to add the **ov.min** input port to the block.

Programmatic Use
Block Parameter: `ov_min`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Upper OV limits` — Add maximum OV constraint input port
off (default) | on

Select this parameter to add the **ov.max** input port to the block.

Programmatic Use
Block Parameter: `ov_max`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Lower MV limits` — Add minimum MV constraint input port
off (default) | on

Select this parameter to add the **mv.min** input port to the block.

Programmatic Use
Block Parameter: `mv_min`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Upper MV limits` — Add maximum MV constraint input port

off (default) | on

Select this parameter to add the **mv.max** input port to the block.

Programmatic Use
Block Parameter: `mv_max`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Lower MVRate limits` — Add minimum MV rate constraint input port

off (default) | on

Select this parameter to add the **dmv.min** input port to the block.

Programmatic Use
Block Parameter: `mvrate_min`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Upper MVRate limits` — Add maximum MV rate constraint input port

off (default) | on

Select this parameter to add the **dmv.max** input port to the block.

Programmatic Use
Block Parameter: `mvrate_max`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Lower state limits` — Add minimum state constraint input port

off (default) | on

Select this parameter to add the **x.min** input port to the block.

Programmatic Use
Block Parameter: `state_min`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Upper state limits` — Add maximum state constraint input port

off (default) | on

Select this parameter to add the **x.max** input port to the block.

Programmatic Use
Block Parameter: `state_max`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`OV weights` — Add OV tuning weights input port

off (default) | on

Select this parameter to add the **y.wt** input port to the block.

Programmatic Use
Block Parameter: `ov_weight`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`MV weights` — Add MV tuning weights input port

off (default) | on

Select this parameter to add the **mv.wt** input port to the block.

Programmatic Use
Block Parameter: `mv_weight`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`MVRate weights` — Add MV rate tuning weights input port

off (default) | on

Select this parameter to add the **dmv.wt** input port to the block.

Programmatic Use
Block Parameter: `mvrate_weight`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`ECR weight` — Add ECR tuning weight input port

off (default) | on

Select this parameter to add the **ecr.wt** input port to the block.

Programmatic Use
Block Parameter: `ecr_weight`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`MV targets` — Add manipulated variable target input port

off (default) | on

Select this parameter to add the **mv.target** input port to the block.

Programmatic Use
Block Parameter: `mvtarget_enabled`
Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

`Initial guess` — Add initial guess input ports

off (default) | on

Select this parameter to add the **mv.init**, **x.init**, and **e.init** input ports to the block.

Note By default, the Nonlinar MPC Controller block uses the calculated optimal manipulated variable and state trajectories from one control interval as the initial guesses for the next control interval.

Enable the initial guess ports only if it is necessary for your application.

Programmatic Use
Block Parameter: `nlp_initialize`

Type: character vector
Values: `'off'`, `'on'`
Default: `'off'`

See Also

`createParameterBus` | `nlmpc` | `nlmpcmove`

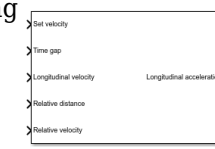
Topics

“Nonlinear MPC”

Introduced in R2018b

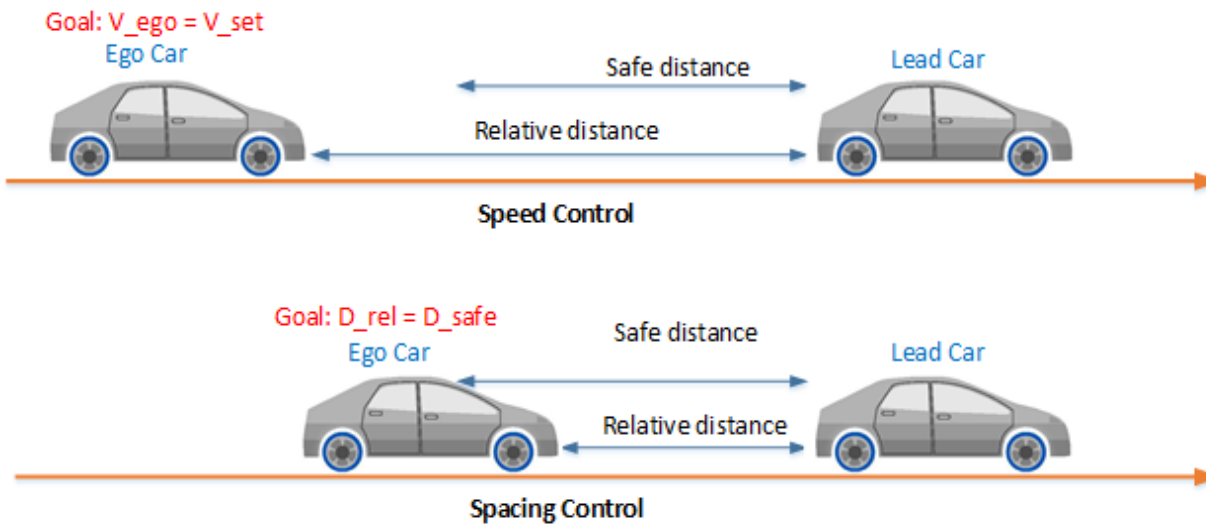
Adaptive Cruise Control System

Simulate adaptive cruise control using model predictive controller
Library: Model Predictive Control Toolbox / Automated Driving



Description

The Adaptive Cruise Control System block simulates an adaptive cruise control (ACC) system that tracks a set velocity and maintains a safe distance from a lead vehicle by adjusting the longitudinal acceleration of an ego vehicle. The block computes optimal control actions while satisfying safe distance, velocity, and acceleration constraints using model predictive control (MPC).



To customize your controller, for example to use advanced MPC features or modify controller initial conditions, click **Create ACC subsystem**.

Ports

Input

`Set velocity` — Ego vehicle velocity setpoint
nonnegative scalar

Ego vehicle velocity setpoint in meters per second. When there is no lead vehicle, the controller tracks this velocity.

`Time gap` — Safe time gap
nonnegative scalar

Safe time gap in seconds between the lead vehicle and the ego vehicle. This time gap is used to calculate the minimum safe following distance constraint. For more information, see “Safe Following Distance” on page 3-96.

`Longitudinal velocity` — Ego vehicle velocity
nonnegative scalar

Ego vehicle velocity in meters per second.

`Relative distance` — Distance between lead vehicle and ego vehicle
positive scalar

Distance in meters between lead vehicle and ego vehicle. To calculate this signal, subtract the ego vehicle position from the lead vehicle position.

`Relative velocity` — Velocity difference between lead vehicle and ego vehicle
scalar

Velocity difference in meters per second between lead vehicle and ego vehicle. To calculate this signal, subtract the ego vehicle velocity from the lead vehicle velocity.

`Minimum longitudinal acceleration` — Minimum ego vehicle acceleration
`-3` (default) | negative scalar

Minimum ego vehicle longitudinal acceleration constraint in m/s^2. Use this input port when the minimum acceleration varies at run time.

Dependencies

To enable this port, select **Use external source** for the **Minimum longitudinal acceleration** parameter.

`Maximum longitudinal acceleration` — Maximum ego vehicle acceleration
2 (default) | positive scalar

Maximum ego vehicle longitudinal acceleration constraint in m/s^2. Use this input port when the maximum acceleration varies at run time.

Dependencies

To enable this port, select **Use external source** for the **Maximum longitudinal acceleration** parameter.

`Enable optimization` — Controller optimization enable signal
scalar

Controller optimization enable signal. When this signal is:

- Nonzero, the controller performs optimization calculations and generates a **Longitudinal acceleration** control signal.
- Zero, the controller does not perform optimization calculations. In this case, the **Longitudinal acceleration** output signal remains at the value it had when the optimization was disabled. The controller continues to update its internal state estimates.

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

`External control signal` — Longitudinal acceleration applied to ego vehicle
scalar

Actual longitudinal acceleration in m/s^2 applied to the ego vehicle.

The model predictive controller uses this signal to estimate the ego vehicle model states when the Adaptive Cruise Control System is not the active controller. Doing so prevents bumps in the control signal when the controller becomes active.

Dependencies

To enable this port, select the **Use external control signal for bumpless transfer** parameter.

Output

`Longitudinal acceleration` — Acceleration control signal

scalar

Acceleration control signal in m/s^2 generated by the controller.

Parameters

Parameters Tab

Ego Vehicle Model

`Linear model from longitudinal acceleration to longitudinal velocity` — Ego vehicle model

`tf(1,[0.5,1,0])` (default) | LTI model | linear System Identification Toolbox model

The linear model from the ego vehicle longitudinal acceleration to its longitudinal velocity, specified as an LTI model or a linear System Identification Toolbox model. The controller creates its internal predictive model by augmenting the ego vehicle dynamic model.

`Initial condition for longitudinal velocity` — Initial velocity of the ego vehicle model

`20` (default) | nonnegative scalar

Initial velocity in m/s of the ego vehicle model, which can differ from the actual ego vehicle initial velocity.

This value is used to configure the initial conditions of the model predictive controller. For more information, see “Initial Conditions” on page 3-97.

`Default spacing` — Minimum spacing to lead vehicle

`10` (default) | nonnegative scalar

Minimum spacing in meters between the lead vehicle and the ego vehicle. This value corresponds to the target relative distance between the ego and lead vehicles when the ego vehicle velocity is zero.

This value is used to calculate the:

- Minimum safe following distance. For more information, see “Safe Following Distance” on page 3-96.
- Controller initial conditions. For more information, see “Initial Conditions” on page 3-97.

`Maximum velocity` — Maximum longitudinal velocity
`50` (default) | positive scalar

Maximum ego vehicle longitudinal velocity in m/s.

Adaptive Cruise Controller Constraints

`Minimum longitudinal acceleration` — Minimum ego vehicle acceleration
`-3` (default) | negative scalar

Minimum ego vehicle longitudinal acceleration constraint in m/s^2

If the minimum acceleration varies over time, select **Use external source** to add the **Minimum longitudinal acceleration** input port to the block.

`Maximum longitudinal acceleration` — Maximum ego vehicle acceleration
`2` (default) | nonnegative scalar

Maximum ego vehicle longitudinal acceleration constraint in m/s^2.

If the maximum acceleration varies over time, select **Use external source** to add the **Maximum longitudinal acceleration** input port to the block.

Model Predictive Controller Settings

`Sample time` — Controller sample time
`0.1` (default) | positive scalar

Controller sample time in seconds.

`Prediction horizon` — Controller prediction horizon
`30` (default) | positive integer

Controller prediction horizon steps. The controller prediction time is the product of the sample time and the prediction horizon.

`Controller behavior` — Closed-loop controller performance
`0.5` (default) | scalar between `0` and `1`

Closed-loop controller performance. The default parameter value provides a balanced controller design. Specifying a:

- Smaller value produces a more robust controller with smoother control actions.
- Larger value produces a more aggressive controller with a faster response time.

When you modify this parameter, the change is applied to the controller immediately.

Block Tab

`Use suboptimal solution` — Apply suboptimal solution after specified number of iterations
`off` (default) | `on`

Configure the controller to apply a suboptimal solution after a specified maximum number of iterations, which guarantees the worst-case execution time for your controller.

For more information, see “Suboptimal QP Solution”.

Dependencies

After selecting this parameter, specify the **Maximum iteration number** parameter.

`Maximum iteration number` — Maximum optimization iterations
`10` (default) | positive integer

Maximum number of controller optimization iterations.

Dependencies

This parameter is enabled when the **Use suboptimal solution** parameter is selected.

`Use external signal to enable or disable optimization` — Add port for enabling optimization
`off` (default) | `on`

Select this parameter to add the **Enable optimization** input port to the block.

`Use external signal for bumpless transfer` — Add external control signal input port
`off` (default) | `on`

Select this parameter to add the **External control signal** input port to the block.

`Create ACC subsystem` — Create custom controller
button

Generate a custom ACC subsystem, which you can modify for your application. The configuration data for the custom controller is exported to the MATLAB workspace as a structure.

You can modify the custom controller subsystem to:

- Modify default MPC settings or use advanced MPC features.
- Modify the default controller initial conditions.
- Use different application settings, such as a custom safe following distance definition.

Algorithms

Safe Following Distance

By default, the model predictive controller computes the safe following distance constraint; that is, the minimum relative distance between the lead and ego vehicle, as:

$$D_R = D_S + G_T * V_E$$

Here:

- D_S is the **Default spacing** parameter.
- G_T is the **Time gap** input signal.
- V_E is the **Longitudinal velocity** input signal.

To define a different safe following distance constraint, create a custom cruise control system by, on the **Block** tab, clicking **Create ACC subsystem**.

Initial Conditions

By default, the model predictive controller assumes the following initial conditions:

- Longitudinal velocity of both the ego vehicle and the lead vehicle equal the **Initial condition for longitudinal velocity** parameter value.
- Ego vehicle longitudinal acceleration is zero.
- Relative distance between the lead vehicle and ego vehicle is:

  $$D_R = D_S + G_T * V_E$$

  Here:

  - D_S is the **Default spacing** parameter.
  - G_T is the time gap and is assumed to be `1.4`.
  - V_E is the ego vehicle velocity, which is equal to the **Initial condition for longitudinal velocity** parameter.

If the initial conditions in your model do not match these conditions, the **Longitudinal acceleration** output can exhibit an initial bump at the start of the simulation.

To modify the controller initial conditions to match your simulation, create a custom cruise control system by, on the **Block** tab, clicking **Create ACC subsystem**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

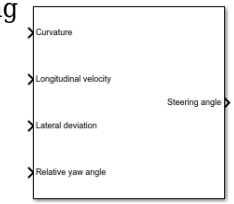
See Also

Blocks
MPC Controller

Introduced in R2018a

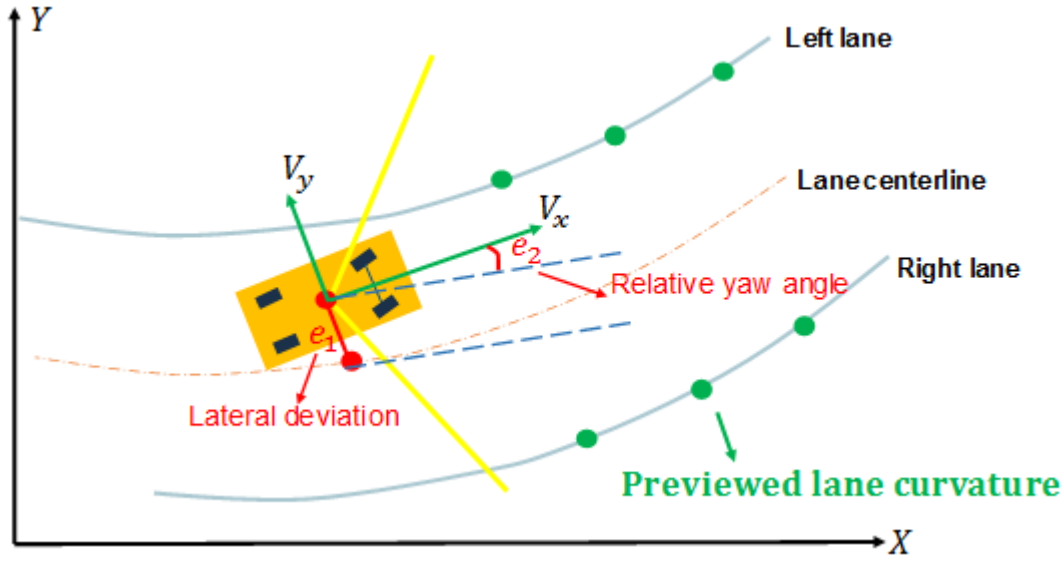
Lane Keeping Assist System

Simulate lane-keeping assistance using adaptive model predictive controller
Library: Model Predictive Control Toolbox / Automated Driving



Description

The Lane Keeping Assist System block simulates a lane keeping assist (LKA) system that keeps an ego vehicle traveling along the center of a straight or curved road by adjusting the front steering angle. The controller reduces the lateral deviation and relative yaw angle of the ego vehicle with respect to the lane centerline. The block computes optimal control actions while satisfying steering angle constraints using adaptive model predictive control (MPC).



To customize your controller, for example to use advanced MPC features or modify controller initial conditions, click **Create LKA subsystem**.

Ports

Input

`Curvature` — Road curvature

scalar

Road curvature, specified as $1/R$, where R is the radius of the curve in meters.

The road curvature is:

- Positive when the road curves toward the positive Y axis of the global coordinate system.
- Negative when the road curves toward the negative Y axis of the global coordinate system.
- Zero for a straight road.

The controller models the road curvature as a measured disturbance with previewing. You can specify the curvature as a:

- Scalar signal — Specify the curvature for the current control interval. The controller uses this curvature value across the prediction horizon.
- Vector signal with length less than or equal to the **Prediction Horizon** — Specify the current and predicted curvature values across the prediction horizon. If the length of the vector is less than the prediction horizon, then the controller uses the final curvature value in the vector for the remainder of the prediction horizon.

`Longitudinal velocity` — Ego vehicle longitudinal velocity
positive scalar

Ego vehicle longitudinal velocity in m/s.

`Lateral deviation` — Ego vehicle lateral deviation
scalar

Ego vehicle lateral deviation in meters from the centerline of the lane.

`Relative yaw angle` — Angle from lane centerline
scalar

Ego vehicle longitudinal axis angle in radians from the centerline of the lane.

`Minimum steering angle` — Minimum front steering angle
scalar

Minimum front steering angle constraint in radians. Use this input port when the minimum steering angle varies at run time.

Dependencies

To enable this port, select **Use external source** for the **Minimum steering angle** parameter.

`Maximum steering angle` — Maximum front steering angle
scalar

Maximum front steering angle constraint in radians. Use this input port when the maximum steering angle varies at run time.

Dependencies

To enable this port, select **Use external source** for the **Maximum steering angle** parameter.

`Enable optimization` — Controller optimization enable signal

scalar

Controller optimization enable signal. When this signal is:

- Nonzero, the controller performs optimization calculations and generates a **Steering angle** control signal.
- Zero, the controller does not perform optimization calculations. In this case, the **Steering angle** output signal remains at the value it had when the optimization was disabled. The controller continues to update its internal state estimates.

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

`External control signal` — Steering angle applied to ego vehicle

scalar

Actual steering angle in radians applied to the ego vehicle.

The model predictive controller uses this signal to estimate the ego vehicle model states when the Lane Keeping Assist System is not the active controller. Doing so prevents bumps in the control signal when the controller becomes active.

Dependencies

To enable this port, select the **Use external control signal for bumpless transfer** parameter.

`Vehicle dynamics matrix A` — State matrix of ego vehicle predictive model

square matrix

State matrix of ego vehicle predictive model.

Dependencies

- To enable this port, select the **Use vehicle parameters** parameter.

- The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

`Vehicle dynamics matrix B` — Input-to-state matrix of ego vehicle predictive model

column vector

Input-to-state matrix of ego vehicle predictive model. The number of rows in this signal must match the number of rows in **Vehicle dynamics matrix A**.

Dependencies

- To enable this port, select the **Use vehicle parameters** parameter.
- The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

`Vehicle dynamics matrix C` — State-to-output matrix of ego vehicle predictive model

matrix with two rows

State-to-output matrix of ego vehicle predictive model. The number of columns in this signal must match the number of rows in **Vehicle dynamics matrix A**.

Dependencies

- To enable this port, select the **Use vehicle parameters** parameter.
- The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Output

`Steering angle` — Front steering angle control signal

scalar

Front steering angle control signal in radians generated by the controller. The front steering angle is the angle of the front tires from the longitudinal axis of the vehicle. The steering angle is positive towards the positive lateral axis of the ego vehicle.

Parameters

Parameters Tab

Ego Vehicle

`Use vehicle parameters` — Define ego vehicle model using vehicle properties
`off` (default) | `on`

Select this parameter to define the ego vehicle model used by the MPC controller by specifying properties of the ego vehicle. The ego vehicle model is the linear model from the front steering angle to the lateral velocity and yaw angle rate. For more information, see “Ego Vehicle Predictive Model” on page 3-108.

Vehicle Property	Description	Units	Default Value
Total mass	Ego vehicle mass	kg	1575
Yaw moment of inertia	Moment of inertia about the ego vehicle vertical axis	mN/s^2	2875
Longitudinal distance from center of gravity to front tires	Distance from the ego vehicle center of mass to its front tires, measured along the longitudinal axis of the vehicle	m	1.2
Longitudinal distance from center of gravity to rear tires	Distance from the ego vehicle center of mass to its rear tires, measured along the longitudinal axis of the vehicle	m	1.6

Vehicle Property	Description	Units	Default Value
Cornering stiffness of front tires	Relationship between the side force on the front tires and the angle of the tires to the longitudinal axis of the vehicle	N/rad	`19000`
Cornering stiffness of rear tires	Relationship between the side force on the rear tires and the angle of the tires to the longitudinal axis of the vehicle	N/rad	`33000`

`Use vehicle model` — Define ego vehicle model using state-space matrices
`off` (default) | `on`

Select this parameter to define the state-space matrices of the ego vehicle model used by the MPC controller. This model is the linear model from the front steering angle in radians to the lateral velocity in meters per second and yaw angle rate in radians per second. For more information on the ego vehicle model, see “Ego Vehicle Predictive Model” on page 3-108.

To define the initial internal model, specify the **A**, **B**, and **C** state-space matrices. The internal model must be a minimal realization with no direct feedthrough, and the dimensions of **A**, **B**, and **C** must be consistent.

Typically, the ego vehicle steering model is velocity-dependent, and therefore, it varies over time. To update the internal model at run time, use the **Vehicle dynamics A**, **Vehicle dynamics B**, and **Vehicle dynamics C** input ports.

`Initial longitudinal velocity` — Initial velocity of the ego vehicle
15 (default) | positive scalar

Initial velocity of the ego vehicle model when the lane-keeping assist is enabled in meters per second. This velocity can differ from the actual ego vehicle initial velocity.

`Transport lag between model inputs and outputs` — Total transport lag in ego vehicle model
`0` (default) | nonnegative scalar

Total transport lag, τ, in ego vehicle model in seconds. This lag includes actuator, sensor, and communication lags. For each input-output channel, the transport lag is approximated by:

$$\frac{1}{\tau s + 1}$$

Lane Keeping Controller Constraints

`Minimum steering angle` — Minimum front steering angle
`-0.26` (default) | scalar between `-pi/2` and `pi/2`

Minimum front steering angle constraint in radians.

If the minimum steering angle varies over time, select **Use external source** to add the **Minimum steering angle** input port to the block.

Dependencies

This parameter must be less than the **Maximum steering angle** parameter.

`Maximum steering angle` — Maximum front steering angle
`0.26` (default) | scalar between `-pi/2` and `pi/2`

Maximum front steering angle constraint in radians.

If the maximum steering angle varies over time, select **Use external source** to add the **Maximum steering angle** input port to the block.

Dependencies

This parameter must be greater than the **Minimum steering angle** parameter.

Model Predictive Controller Settings

`Sample time` — Controller sample time
`0.1` (default) | positive scalar

Controller sample time in seconds.

`Prediction horizon` — Controller prediction horizon
`10` (default) | positive integer

Controller prediction horizon steps. The controller prediction time is the product of the sample time and the prediction horizon.

`Controller behavior` — Closed-loop controller performance
`0.5` (default) | scalar between `0` and `1`

Closed-loop controller performance. The default parameter value provides a balanced controller design. Specifying a:

- Smaller value produces a more robust controller with smoother control actions.
- Larger value produces a more aggressive controller with a faster response time.

When you modify this parameter, the change is applied to the controller immediately.

Block Tab

`Use suboptimal solution` — Apply suboptimal solution after specified number of iterations
`off` (default) | `on`

Configure the controller to apply a suboptimal solution after a specified maximum number of iterations, which guarantees the worst-case execution time for your controller.

For more information, see “Suboptimal QP Solution”.

Dependencies

Once you select this parameter, specify the **Maximum iteration number** parameter.

`Maximum iteration number` — Maximum optimization iterations
`10` (default) | positive integer

Maximum number of controller optimization iterations.

Dependencies

This parameter is enabled when the **Use suboptimal solution** parameter is selected.

`Use external signal to enable or disable optimization` — Add port for enabling optimization
`off` (default) | `on`

Select this parameter to add the **Enable optimization** input port to the block.

`Use external signal for bumpless transfer` — Add external control signal input port
`off` (default) | `on`

Select this parameter to add the **External control signal** input port to the block.

`Create LKA subsystem` — Create custom controller
button

Generate a custom LKA subsystem, which you can modify for your application. The controller configuration data for the custom controller is exported to the MATLAB workspace as a structure.

You can modify the custom controller subsystem to:

- Modify default MPC settings or use advanced MPC features.
- Modify the default controller initial conditions.

Algorithms

Ego Vehicle Predictive Model

The default ego vehicle predictive model is the following state-space model:

$$A = \begin{bmatrix} -2(C_F + C_R)/m/V_X & -V_X - 2(C_F L_F - C_R L_R)/m/V_X \\ -2(C_F L_F - C_R L_R)/I_Z/V_X & -2\left(C_F L_F^2 + C_R L_R^2\right)/I_Z/V_X \end{bmatrix}$$

$$B = 2C_F \begin{bmatrix} 1/m \\ L_F/I_Z \end{bmatrix}$$

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

$$D = 0$$

Here:

- V_X is the longitudinal velocity of the car. At the start of the simulation, this velocity is equal to the **Initial condition for longitudinal velocity** parameter. At run time, this velocity is equal to the **Longitudinal velocity** input signal.

- m is the **Total mass** parameter.
- I_Z is the **Yaw moment of inertia** parameter.
- L_F is the **Longitudinal distance from center of gravity to front tires** parameter.
- L_R is the **Longitudinal distance from center of gravity to rear tires** parameter.
- C_F is the **Cornering stiffness of front tires** parameter.
- C_R is the **Cornering stiffness of rear tires** parameter.

The input to this model is the steering angle in radians, and the outputs are the lateral velocity in meters per second and yaw angle rate in radians per second.

To define a different ego vehicle predictive model, select the **Use vehicle model** parameter, and specify the initial state-space model. Then, specify the run-time values of the state-space matrices using the **Vehicle dynamics A**, **Vehicle dynamics B**, and **Vehicle dynamics C** input signals.

The controller creates its internal predictive model by augmenting the ego vehicle dynamic model. The augmented model includes the road curvature as a measured disturbance input signal.

Initial Conditions

By default, the model predictive controller assumes the following initial conditions for the ego vehicle:

- Longitudinal velocity is equal to the **Initial longitudinal velocity** parameter.
- Lateral velocity is zero.
- Steering angle is zero.
- Yaw angle rate is zero.

If the initial conditions in your model do not match these conditions, the **Steering angle** output can exhibit an initial bump at the start of the simulation.

To modify the controller initial conditions to match your simulation, create a custom lane-keeping control system by, on the **Block** tab, clicking **Create LKA subsystem**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks

Adaptive MPC Controller

Introduced in R2018a

Object Reference

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 vector)	`InputName` of LTI plant model
`Units`	Read-only MV signal units (character vector)	`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 vector)	`OutputName` of LTI plant model
`Units`	Read-only OV signal units (character vector)	`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 vector)	`InputName` of LTI plant model
`Units`	Read-only DV signal units (character vector)	`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)`
`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` = $R_{\Delta \mathrm{u}}$ 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.)

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

Ts

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

Optimizer

`Optimizer` is a structure with fields that contain parameters for the QP optimization.

Optimizer Properties

Field	Description	Default
`MaxIter`	Maximum number of iterations allowed in the QP solver, specified as one of the following: • `'Default'` — The MPC controller automatically computes the maximum number of QP solver iterations as: $\mathrm{MaxIter} = 4\left(n_c + n_v\right)$ where • n_{cy} is the total number of constraints across the prediction horizon. • n_v is the total number of optimization variables across the control horizon. The default `MaxIter` value has a lower bound of `120`. • Positive integer — The QP solver stops after `MaxIter` iterations. If the solver fails to converge in the final iteration, the controller: • Freezes the controller movement if `UseSuboptimalSolution` is `false`. • Applies the suboptimal solution reached after the final iteration if `UseSuboptimalSolution` is `true`. If `CustomSolver` or `CustomSolverCodeGen` is `true`, the controller does not require the custom solver to honor `MaxIter`.	`'Default'`

Field	Description	Default
`MinOutputECR`	Minimum value allowed for `OutputMinECR` and `OutputMaxECR`, specified as a nonnegative scalar. A value of 0 indicates that hard output constraints are allowed. If either of the `OutputVariables.MinECR` or `OutputVariables.MaxECR` properties of an MPC controller are less than `MinOutputECR`, a warning is displayed and the value is raised to `MinOutputECR` during computation.	`0`
`UseSuboptimalSolution`	Flag indicating whether to apply a suboptimal solution after the number of optimization iterations exceeds `MaxIter`, specified as a logical value.	`false`
`UseWarmStart`	Flag indicating whether to *warm start* each QP solver iteration by passing in a list of active inequalities from the previous iteration, specified as a logical value. Inequalities are active when their equal portion is true. If `CustomSolver` or `CustomSolverCodeGen` is `true`, the controller does not require the custom solver to honor `UseWarmStart`.	`true`

Field	Description	Default
`CustomSolver`	Flag indicating whether to use a custom QP solver for simulation, specified as a logical value. If `CustomSolver` is `true`, the user must provide an `mpcCustomSolver` function on the MATLAB path. This custom solver is not used for code generation. To generate code for a controller with a custom solver, use `CustomSolverCodeGen`. If `CustomSolver` is `true`, the controller does not require the custom solver to honor `MaxIter` and `UseWarmStart`. For more information on specifying custom solvers, see “Custom QP Solver”.	`false`
`CustomSolverCodeGen`	Flag indicating whether to use a custom QP solver for code generation, specified as a logical value. If `CustomSolverCodeGen` is `true`, the user must provide an `mpcCustomSolverCodeGen` function on the MATLAB path. This custom solver is not used for simulation. To simulate a controller with a custom solver, use `CustomSolver`. If `CustomSolverCodeGen` is `true`, the controller does not require the custom solver to honor `MaxIter` and `UseWarmStart`. For more information on specifying custom solvers, see “Custom QP Solver”.	`false`

Note The default `MaxIter` value can be very large for some controller configurations, such as those with large prediction and control horizons. When simulating such

controllers, if the QP solver cannot find a feasible solution, the simulation can appear to stop responding, since the solver continues searching for `MaxIter` iterations.

PredictionHorizon

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

ControlHorizon

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

History

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

Notes

`Notes` stores text or comments as a cell array of character vectors.

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'`
`MDLookAhead`	Preview on measured disturbance signal (`'on'` or `'off'`).
`Constraints`	Use MPC constraints (`'on'` or `'off'`). Default: `'on'`

Property	Description
`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 `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.

Property	Description
`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).

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 4-2 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.

Property	Description
`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 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.