Model Predictive Control Toolbox™

Reference

R2014**a**

Alberto Bemporad
Manfred Morari
N. Lawrence Ricker

MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Model Predictive Control Toolbox™ Reference

© COPYRIGHT 2005–2014 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)

Contents

Functions – Alphabetical List

Purpose

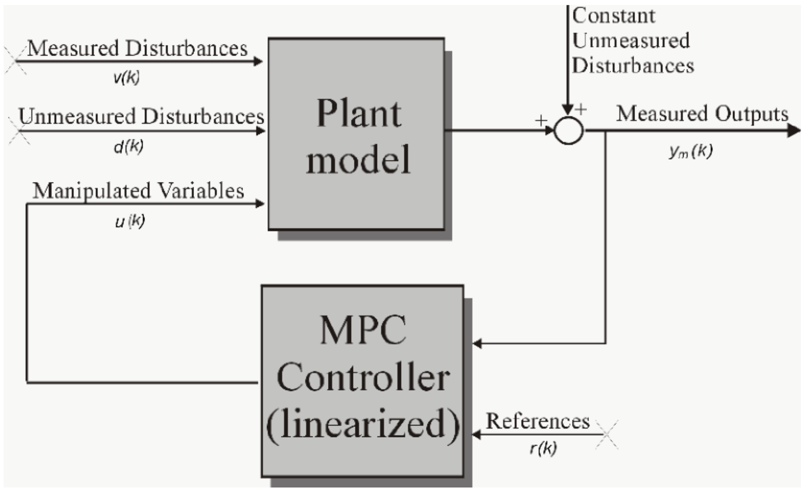
Compute MPC closed-loop DC gain from output disturbances to measured outputs assuming constraints are inactive at steady state

Syntax

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

Description

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



Computing the Effect of Output Disturbances

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

`DCgain=cloffset(MPCobj)` returns an n_{ym}-by-n_{ym} DC gain matrix `DCgain`, where n_{ym} is the number of measured plant outputs. `MPCobj` is the MPC object specifying the controller for which the closed-loop gain is calculated. `DCgain(i,j)` represents the gain from an additive (constant) disturbance on output `j` to measured output `i`. If row `i` contains all zeros, there will be no steady-state offset on output `i`.

See Also

`mpc` | `ss`

Related Examples

- "Compute Steady-State Gain"

compare

Purpose 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` | `pack`

Purpose Change MPC controller's sampling time

Syntax

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

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

See Also `mpc` | `set`

get

Purpose

MPC property values

Syntax

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

Description

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

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

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

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`

Purpose

Model Predictive Control custom constraint definitions

Syntax

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

Description

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

$$Eu(k+j)+Fy(k+j)+Sv(k+j)\leq G+\varepsilon V \qquad \textbf{(1-1)}$$

where:

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

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

Input Arguments

mpcobj

MPC controller, specified as an `mpc` object.

Output Arguments

E

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

`[]` if `mpcobj` contains no custom constraints.

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

F

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

`[]` if `mpcobj` contains no custom constraints.

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

G

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

`[]` if `mpcobj` contains no custom constraints.

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

V

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

`[]` if `mpcobj` contains no custom constraints.

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

If

- `V(i) = 0` — the i^{th} constraint is hard
- `V(i) > 0` — the i^{th} constraint is soft

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

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

S

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

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

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

Examples

Obtain the constraints associated with an MPC controller.

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

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

Assume that you have two soft constraints.

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

Set the constraints for the `mpc` object.

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

Obtain the constraints for `c`.

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

     1     1
     0     0


F =

     0     0
     0     1
```

```
G =

     5
    10


V =

     1
     1


S =

     0
     1
```

See Also `setconstraint`

Purpose

Model and gain for observer design

Syntax

```
M = getestim(MPCobj)
[M,A,Cm] = getestim(MPCobj)
[M,A,Cm,Bu,Bv,Dvm] = getestim(MPCobj)
[M,model,Index] = getestim(MPCobj,'sys')
```

Description

`M = getestim(MPCobj)` extracts the estimator gain `M` used by the MPC controller `MPCobj` for observer design. The observer is based on the models specified in `MPCobj.Model.Plant`, in `MPCobj.Model.Disturbance`, by the output disturbance model (default is integrated white noise, see "Output Disturbance Model"), `and by` `MPCobj.Model.Noise`.

The state estimator is based on the linear model (see "State Estimation")

$x(k + 1) = Ax(k) + B_u u(k) + B_v v(k)$

$y_m(k) = C_m x(k) + D_{vm} v(k)$

where $v(k)$ are the measured disturbances, $u(k)$ are the manipulated plant inputs, $y_m(k)$ are the measured plant outputs, and $x(k)$ is the overall state vector collecting states of plant, unmeasured disturbance, and measurement noise models.

The estimator used in the Model Predictive Control Toolbox™ software is described in "State Estimation". The estimator's equations are

Predicted Output Computation:

$$\hat{y}_m(k|k-1) = C_m \hat{x}(k|k-1) + D_{vm} v(k)$$

Measurement Update:

$$\hat{x}(k|k) = \hat{x}(k|k-1) + M\left(y_m(k) - \hat{y}_m(k|k-1)\right)$$

Time Update:

$$\hat{x}(k+1|k) = A\hat{x}(k|k) + B_u u(k) + B_v v(k)$$

By combining these three equations, the overall state observer is

$$\hat{x}(k+1|k) = (A - LC_m)\hat{x}(k|k-1) + Ly_m(k) + B_u u(k) + (B_v - LD_{vm})v(k)$$

where L=AM.

`[M,A,Cm] = getestim(MPCobj)` also returns matrices A,C_m used for observer design. This includes the plant model, disturbance model, noise model, and offsets. The extended state is

x=[plant states; disturbance models states; noise model states]

`[M,A,Cm,Bu,Bv,Dvm] = getestim(MPCobj)` retrieves the whole linear system used for observer design.

`[M,model,Index] = getestim(MPCobj,'sys')` retrieves the overall model used for observer design (specified in the `Model` field of the MPC object) as an LTI state-space object, and optionally a structure `Index` summarizing I/O signal types.

The extended input vector of model `model` is

u=[manipulated vars;measured disturbances; 1; noise exciting disturbance model;noise exciting noise model]

Model `model` has an extra measured disturbance input v=1 used for handling possible nonequilibrium nominal values (see "Offsets").

Input, output, and state names and input/output groups are defined for model `model`.

The structure `Index` has the fields detailed in the following table.

Field Name	Description
`ManipulatedVariables`	Indices of manipulated variables within input vector
`MeasuredDisturbances`	Indices of measured disturbances within input vector (not including offset=1)

Field Name	Description
`Offset`	Index of offset=1
`WhiteNoise`	Indices of white noise signals within input vector
`MeasuredOutputs`	Indices of measured outputs within output vector
`UnmeasuredOutputs`	Indices of unmeasured outputs within output vector

To improve the solvability of the Kalman filter design, the software adds white noise to the manipulated variables and measured disturbances, as described in “State Observer”. The model returned by `getestim` does not include this additional white noise.

See Also

`setestim` | `mpc` | `mpcstate`

getindist

Purpose

Unmeasured input disturbance model

Syntax

```
model=getindist(MPCobj)
```

Description

`model=getindist(MPCobj)` retrieves the linear discrete-time transfer function used to model unmeasured input disturbances in the MPC setup described by the MPC object `MPCobj`. Model `model` is an LTI object with as many outputs as the number of unmeasured input disturbances, and as many inputs as the number of white noise signals driving the input disturbance model.

For details about the overall model used in the MPC algorithm for state estimation purposes, see "State Estimation".

See Also

`mpc` | `setindist` | `setestim` | `getestim` | `getoutdist`

Purpose

Private MPC data structure

Note `getmpcdata` will be removed in a future version. Use `get`, `getconstraint`, `getestim`, `getindist` and`getoutdist` instead.

Syntax

```
mpcdata=getmpcdata(MPCobj)
```

Description

`mpcdata=getmpcdata(MPCobj)` returns the private field `MPCData` of the MPC object `MPCobj`. Here, all internal QP matrices, models, estimator gains are stored at initalization of the object. You can manually change the private data structure using the `setmpcdata` command, although you may only need this for very advanced use of Model Predictive Control Toolbox software.

Note Changes to the data structure may easily lead to unpredictable results.

See Also

`setmpcdata` | `set` | `get`

getname

Purpose

I/O signal names in MPC prediction model

Syntax

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

Description

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

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

See Also

`setname | mpc | set`

Purpose

Unmeasured output disturbance model

Syntax

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

Description

`outdist=getoutdist(MPCobj)` retrieves the linear discrete-time transfer function used to model output disturbances in the MPC setup described by the MPC object `MPCobj`. Model `outdist` is an LTI object with as many outputs as the number of measured + unmeasured outputs, and as many inputs as the number of white noise signals driving the output disturbance model.

For details about the overall model used in the MPC algorithm for state estimation purposes, see “State Estimation”.

`[outdist,channels]=getoutdist(MPCobj)` also returns the output channels where integrated white noise was added as an output disturbance model. This is only meaninful when the default output disturbance model is used, namely when `MPCobj.OutputVariables(i).Integrators` is empty for all channels `i`. The array `channels` is empty for output disturbance models.

See Also

`mpc` | `setoutdist` | `setestim` | `getestim` | `getindist`

gpc2mpc

Purpose

Generate MPC controller using generalized predictive controller (GPC) settings

Syntax

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

Description

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

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

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

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.

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.

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);
```

References

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

See Also

"MPC Controller Object" on page 3-2

- "Design Controller Using the Design Tool"
- "Design Controller at the Command Line"

Purpose

Create MPC controller

Syntax

```
MPCobj=mpc(plant)
MPCobj=mpc(plant,ts)
MPCobj=mpc(plant,ts,p)
MPCobj=mpc(plant,ts,p,m)
MPCobj=mpc(plant,ts,p,m,W)
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 an MPC controller based on the discrete-time model `plant`. The model can be specified either as an LTI object, or as an object in System Identification Toolbox™ format. See "Identify Plant from Data".

`MPCobj=mpc(plant,ts)` specifies the sampling time `ts` for the MPC controller. A continuous-time `plant` is discretized with sampling time `ts`. A discrete-time `plant` is resampled if its sampling time is different than the controller's sampling time `ts`. If `plant` is a discrete-time model with unspecified sampling time, namely `plant.ts=-1`, then Model Predictive Control Toolbox software assumes that the plant is sampled with the controller's sampling time `ts`. `ts` has the same unit as the internal predictive plant model, i.e., `plant.TimeUnit`.

`MPCobj=mpc(plant,ts,p)` specifies the prediction horizon `p`.

`MPCobj=mpc(plant,ts,p,m)` specifies the control horizon `m`.

`MPCobj=mpc(plant,ts,p,m,W)` also specifies the structure `W` of input, input increments, and output weights (see "Weights" on page 3-7).

`MPCobj=mpc(plant,ts,p,m,W,MV,OV,DV)` also specifies limits on manipulated variables (`MV`) and output variables (`OV`), as well as equal concern relaxation values, units, etc. Names and units of input disturbances can be also specified in the optional input `DV`. The fields of structures `MV`, `OV`, and `DV` are described in "ManipulatedVariables" on page 3-3, in "OutputVariables" on page 3-4, and in "DisturbanceVariables" on page 3-6, respectively).

`MPCobj=mpc(models,ts,p,m,W,MV,OV,DV)` where `model` is a structure containing models for plant, unmeasured disturbances, measured disturbances, and nominal linearization values, as described in "Model" on page 3-9.

Note Other MPC properties are specified by using `set(MPCobj,Property1, Value1,Property2,Value2,...)` or `MPCobj.Property=Value`.

Construction and Initialization

An MPC controller is built in two steps. The first step happens *at construction* when the object constructor `mpc` is invoked, or properties are changed by a `set` command. At this first stage, only basic consistency checks are performed, such as dimensions of signals, weights, constraints, etc. The second step happens *at initialization*, namely when the object is used for the first time by methods such as `mpcmove` and `sim`, that require the full computation of the QP matrices and the estimator gain. At this second stage, further checks are performed, such as a test of observability of the overall extended model.

Informative messages are displayed in the command window in both phases, you can turn them on or off using the `mpcverbosity` command.

Properties

All the parameters defining the MPC control law (prediction horizon, weights, constraints, etc.) are stored in an MPC object, whose properties are listed in the following table (MPC Controller Object on page 3-2).

MPC Controller Object

Property	Description
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	Input and input-rate upper and lower bounds, ECR values, names, units, and input target
`OutputVariables` (or `OV` or `Controlled` or `Output`)	Output upper and lower bounds, ECR values, names, units

MPC Controller Object (Continued)

Property	Description
`DisturbanceVariables` (or `DV` or `Disturbance`)	Disturbance names and units
`Weights`	Weights defining the performance function
`Model`	Plant, input disturbance, and output noise models, and nominal conditions.
`Ts`	Controller's sampling time
`Optimizer`	Parameters for 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
`MPCData` (private)	Matrices for the QP problem and other accessorial data
`Version` (private)	Model Predictive Control Toolbox version number

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 (Structure ManipulatedVariables on page 3-3), where p denotes the prediction horizon.

Structure ManipulatedVariables

Field Name	Content	Default
`Min`	1 to p dimensional vector of lower constraints on a manipulated variable u	`-Inf`
`Max`	1 to p dimensional vector of upper constraints on a manipulated variable u	`Inf`
`MinECR`	1 to p dimensional vector describing the equal concern for the relaxation of the lower constraints on u	`0`
`MaxECR`	1 to p dimensional vector describing the equal concern for the relaxation of the upper constraints on u	`0`
`Target`	1 to p dimensional vector of target values for the input variable u	`'nominal'`
`RateMin`	1 to p dimensional vector of lower constraints on the rate of a manipulated variable u	`-Inf` if problem is unconstrained, otherwise `-10`
`RateMax`	1 to p dimensional vector of upper constraints on the rate of a manipulated variable u	`Inf`
`RateMinECR`	1 to p dimensional vector describing the equal concern for the relaxation of the lower constraints on the rate of u	`0`

Structure ManipulatedVariables (Continued)

Field Name	Content	Default
`RateMaxECR`	1 to p dimensional vector describing the equal concern for the relaxation of the upper constraints on the rate of u	`0`
`Name`	Name of input signal. This is inherited from `InputName` of the LTI plant model.	`InputName` of LTI plant model
`Units`	String specifying the measurement units for the manipulated variable	`''`

Note Rates refer to the difference $\Delta u(k)=u(k)-u(k\text{-}1)$. 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 (Structure OutputVariables on page 3-5), where p denotes the prediction horizon.

Structure OutputVariables

Field Name	Content	Default
`Min`	1 to p dimensional vector of lower constraints on an output y	`-Inf`
`Max`	1 to p dimensional vector of upper constraints on an output y	`Inf`

Structure OutputVariables (Continued)

Field Name	Content	Default
`MinECR`	1 to p dimensional vector describing the equal concern for the relaxation of the lower constraints on an output y	`1`
`MaxECR`	1 to p dimensional vector describing the equal concern for the relaxation of the upper constraints on an output y	`1`
`Name`	Name of output signal. This is inherited from `OutputName` of the LTI plant model.	`OutputName` of LTI plant model
`Units`	String specifying the measurement units for the measured output	`''`
`Integrator`	Magnitude of integrated white noise on the output channel (0=no integrator)	`[]`

In order to reject constant disturbances due for instance to gain nonlinearities, the default output disturbance model used in Model Predictive Control Toolbox software is a collection of integrators driven by white noise on measured outputs (see “Output Disturbance Model”). Output integrators are added according to the following rule:

1 Measured outputs are ordered by decreasing output weight (in case of time-varying weights, the sum of the absolute values over time is considered for each output channel, and in case of equal output weight, the order within the output vector is followed).

2 By following such order, an output integrator is added per measured outputs, unless there is a violation of observability, or you force it by zeroing the corresponding value in `OutputVariables.Integrators`).

By default, `OutputVariables.Integrators` is empty on all outputs. This enforces the default action of Model Predictive Control Toolbox software, namely add integrators on measured outputs, do not add integrators on unmeasured outputs. By setting the entry of `OutputVariables(i).Integrators` to zero, no attempt will be made to add integrated white noise on the `i`-th output . On the contrary, by setting the entry of `OutputVariables(i).Integrators` to one, an attempt will be made to add integrated white noise on the `i`-th output (see `getoutdist`).

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), one per input disturbance. Each structure has the fields described in the following table (Structure DisturbanceVariables on page 3-6).

Structure DisturbanceVariables

Field Name	Content	Default
`Name`	Name of input signal. This is inherited from `InputName` of the LTI plant model.	`InputName` of LTI plant model
`Units`	String specifying the measurement units for the manipulated variable	`''`

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

Note The `Name` properties of `ManipulatedVariables`, `OutputVariables`, and `DisturbanceVariables` are read only. You can set signal names in the `Model.Plant.InputName` and `Model.Plant.OutputName`properties of the MPC object, for instance by using the method `setname`.

Weights

`Weights` is the structure defining the QP weighting matrices. Unlike the `InputSpecs` and `OutputSpecs`, which are arrays of structures, `W` is a single structure containing four fields. The values of these fields depend on whether you are using the standard quadratic cost function (see "Standard Form") or the alternative cost function (see "Alternative Cost Function").

Standard Cost Function

The table below, Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7, lists the content of the four fields where p denotes the prediction horizon, n_u the number of manipulated variables, n_y the number of output variables.

The fields `ManipulatedVariables`, `ManipulatedVariablesRate`, and `OutputVariables` are arrays with n_u, n_u, and n_y columns, respectively. If weights are time invariant, then `ManipulatedVariables`, `ManipulatedVariablesRate`, and `OutputVariables` are row vectors. However, for time-varying weights, each field is a matrix with up to p rows. If the number of rows is less than the prediction horizon, p, the object constructor duplicates the last row to create a matrix with p rows.

Weights for the Standard Cost Function (MATLAB Structure)

Field Name	Content	Default
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	(1 to p)-by-n_u dimensional array of input weights	`zeros(1,nu)`
`ManipulatedVariablesRate` (or `MVRate` or `ManipulatedRate` or `InputRate`)	(1 to p)-by-n_u dimensional array of input-rate weights	`0.1*ones(1,nu)`

Weights for the Standard Cost Function (MATLAB Structure) (Continued)

Field Name	Content	Default
`OutputVariables` (or `OV` or `Controlled` or `Output`)	(1 to p)-by-n_y dimensional array of output 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 are weighted with unit weight (with preference given to measured outputs), while the remaining outputs receive zero weight.)
`ECR`	Weight on the slack variable ε used for softening the constraints	1e5*(max weight)

The default ECR weight is 10^5 times the largest weight specified in `ManipulatedVariables`, `ManipulatedVariablesRate`, and `OutputVariables`.

Note All weights must be greater than or equal to zero. If all weights on manipulated variable increments are strictly positive, the resulting QP problem is always strictly convex. If some of those weights are zero, the Hessian matrix of the QP problem may become only positive semidefinite. In order to keep the QP problem always strictly convex, if the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12}, the quantity `10*sqrt(eps)` is added on each diagonal term. This may only occur when all input rates are not weighted ($W^{\Delta u}$=0) (see "Cost Function").

Alternative Cost Function

You can specify off-diagonal Q and R weight matrices in the cost function. To accomplish this, you must 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 abbreviate the field names as shown in Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7. You can also use diagonal weights (as defined in Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7) for one or more of these fields. If you omit a field, the object constructor uses the defaults shown in Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7.

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

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

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

Note You cannot specify off-diagonal time-varying weights.

Model

The property `Model` specifies plant, input disturbance, and output noise models, and nominal conditions, according to the model setup described in "State Estimation". It is specified through a structure containing the fields reported in Structure Model Describing the Models Used by MPC on page 3-9.

Structure Model Describing the Models Used by MPC

Field Name	Content	Default
`Plant`	LTI model or identified linear model of the plant	No default
`Disturbance`	LTI model describing color of input disturbances	An integrator on each `Unmeasured` input channel
`Noise`	LTI model describing color of plant output measurement noise	Unit white noise on each measured output = identity static gain
`Nominal`	Structure containing the state, input, and output values where `Model.Plant` is linearized	See Nominal Values at Operating Point on page 3-11.

Note Direct feedthrough from manipulated variables to any output in `Model.Plant` is not allowed. See "Prediction Model".

The type of input and output signals is assigned either through the `InputGroup` and `OutputGroup` properties of `Model.Plant`, or, more conveniently, through function `setmpcsignals`, according to the nomenclature described in Input Groups in Plant Model on page 3-10 and Output Groups in Plant Model on page 3-11.

Input Groups in Plant Model

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

Output Groups in Plant Model

Name	Value
`MeasuredOutputs` (or `MO` or `Measured`)	Indices of measured outputs
`UnmeasuredOutputs` (or `UO` or `Unmeasured`)	Indices of unmeasured outputs

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

Note With this current release, the `InputGroup` and `OutputGroup` properties of LTI objects are defined as structures, rather than cell arrays (see the Control System Toolbox™ documentation for more details).

The structure `Nominal` contains the nominal values for states, inputs, outputs and state derivatives/differences at the operating point where `Model.Plant` was linearized. The fields are reported in Nominal Values at Operating Point on page 3-11 (see "Offsets").

Nominal Values at Operating Point

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

Ts

Sampling 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 sampling time for the MPC controller.

Optimizer

Parameters for the QP optimization. Optimizer is a structure with the fields reported in the following table (Optimizer Properties on page 3-12).

Optimizer Properties

Field	Description	Default
`MaxIter`	Maximum number of iterations allowed in the QP solver	`200`
`Trace`	On/off	`'off'`
`Solver`	QP solver used (only `'ActiveSet'`)	`'ActiveSet'`
`MinOutputECR`	Minimum positive value allowed for `OutputMinECR` and `OutputMaxECR`	`1e-10`

`MinOutputECR` is a positive scalar used to specify the minimum allowed ECR for output constraints. If values smaller than `MinOutputECR` are provided in the `OutputVariables` property of the MPC objects a warning message is issued and the value is raised to `MinOutputECR`.

PredictionHorizon

`PredictionHorizon` is an integer value expressing the number *p* of sampling steps of prediction.

ControlHorizon

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

History

`History` stores the time the MPC controller was created.

Notes

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

UserData

Any additional data stored within the MPC controller object.

MPCData

`MPCData` is a private property of the MPC object used for storing intermediate operations, QP matrices, internal flags, etc.

Version

`Version` is a private property indicating the Model Predictive Control Toolbox version number.

Examples

Define an MPC controller based on the transfer function model s+1/(s^2+2s), with sampling time T_s=0.1 s, and satisfying the input constraint -1≤ u ≤1:

```
Ts=.1;    %Sampling time
MV=struct('Min',-1,'Max',1);
p=20;
```

```
m=3;

mpc1=mpc(tf([1 1],[1 2 0]),Ts,p,m,[],MV);
```

See Also set | get

mpchelp

Purpose MPC property and function help

Syntax

```
mpchelp
mpchelp name
out=mpchelp(`name')
mpchelp(obj)
mpchelp(obj,'name')
out=mpchelp(obj,'name')
```

Description `mpchelp` provides a complete listing of Model Predictive Control Toolbox help.

`mpchelp name` provides online help for the function or property `name`.

`out=mpchelp(`name')` returns the help text in string, `out`.

`mpchelp(obj)` displays a complete listing of functions and properties for the MPC object, `obj`, along with the online help for the object's constructor.

`mpchelp(obj,'name')` displays the help for function or property, `name`, for the MPC object, `obj`.

`out=mpchelp(obj,'name')` returns the help text in string, `out`.

Examples To get help on the MPC method `getoutdist`, you can type:

```
mpchelp getoutdist
```

See Also `mpcprops`

Purpose

Optimal control action

Syntax

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

Description

`u = mpcmove(MPCobj,x,ym,r,v)` computes the optimal manipulated variable moves, *u(k)*. *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 time *k*. Call `mpcmove` repeatedly to simulate closed-loop model predictive control.

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

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

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`.

Input Arguments

MPCobj

`mpc` object that defines the model predictive controller.

x

`mpcstate` object that contains the following:

- Estimated plant model states, $x(k|k\text{-}1)$
- Estimated input and output disturbance model states, $x_d(k|k\text{-}1)$
- Estimated measurement noise model states, $x_m(k|k\text{-}1)$
- Last controller moves, $u(k\text{-}1)$

To initialize `x`, use `x = mpcstate(MPCobj)` and modify default properties as needed.

You may need to change the value of `x` before using it at the next time step under certain circumstances. Suppose that the optimal value for *u* calculated in the previous call to `mpcmove` was not used in the plant for some reason (e.g., saturation). You must replace `LastMove` with the values actually used.

`mpcmove` updates all four properties of `x` to prepare for an `mpcmove` call at the next time step. For example, it copies the recommended *u* values into the `LastMove` field.

To retain the original contents of `x`, create a copy of it before calling `mpcmove`.

ym

1-by-n_{ym} vector of current measured output values at time *k*.

n_{ym} is the number of measured outputs.

r

p-by-n_y array of future reference values for the outputs, where *p* is the prediction horizon and n_y is the number of outputs. The i^{th} row of `r` defines the reference values at time `k+i`, for `i = 1,...,p`.

The first row must contain the reference signal at time `k+1`. Additional rows represent known future references and are optional (future values are unknown in most applications). If you supply fewer than *p* rows of data, `mpcmove` duplicates the last row to fill the array.

You cannot preview the reference signal (also referred to as look-ahead and anticipation) if `r` contains a single row. To support reference previewing, supply at least two rows of data for `r`.

v

p-by-n_{md} array of current and future measured disturbance, where p is the prediction horizon and n_{md} is the number of measured disturbances used in feed-forward control. The i^{th} row of `v` defines the measured disturbance values at time *k*`+i-1`, for `i = 1,...,p`.

The first row must contain the current measured disturbance values. Additional rows represent known future values and are optional (future values are unknown in most applications). If you supply fewer than p rows of data, `mpcmove` duplicates the last row to fill the array.

You cannot preview the measured disturbance signal (also referred to as look-ahead and anticipation) if `v` contains a single row. To support disturbance previewing, supply at least two rows of data for `v`.

Options

`mpcmoveopt` object that overrides constraints and weights in `MPCobj`. This approach is computationally efficient to simulate run-time changes in controller tuning and limit values. The same behavior can be found in a Simulink® MPC Controller block when supplied with controller tuning and limit input signals.

Output Arguments

u

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

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

Info

Information regarding the model predictive controller.

`Info` is a structure with the following fields:

- `Uopt` — *p*+1-by-n_u array containing the optimal manipulated variable adjustments (moves),where *p* is the prediction horizon and n_u is the number of manipulated variables.

  The first row is the same as `u`, which is to be applied at the current time *k*. `Uopt(i,:)` contains the predicted optimal values at time *k*+*i*-1, for `i = 1,...,p+1`.

  `mpcmove` does not calculate optimal control moves at time *k*+*p* and therefore it sets `Uopt(p+1,:)` to `NaN`.

- `Yopt` — *p*+1-by-n_y array containing the predicted output variable sequence, where *p* is the prediction horizon and n_y is the number of outputs.

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

- `Xopt` — *p*+1-by-n_x array containing the predicted state variable sequence, where *p* is the prediction horizon and n_x is the number of states.

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

- `Topt` — *p*+1-by-1 vector of time intervals where `Topt(1)=0` (representing the current time) and `Topt(i)=Ts*(i-1)`, where `Ts=MPCobj.Ts`, the controller sample time and for `i = 1,...,p+1`.

  Use when plotting `Uopt`, `Yopt` and/or `Xopt` sequences.

- `Slack` — Slack variable, ε, used in constraint softening.

  `Slack` is a scalar that may have the following values:

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

See “Optimization Problem” for details.

- `Iterations` — QP solution result.

  `Iterations` is a scalar integer that may have the following values:

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

- `QPCode` — QP solution status.

  `QPCode` is a String that may have the following values:

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

- `Cost` — Cost of the objective function.

  `Cost` quantifies the degree to which the controller has achieved its objectives and is a non-negative scalar value. `Cost` is only meaningful when `QPCode` is `'feasible'`.

  See “Optimization Problem” for details.

Examples

Analyze Closed-Loop Response

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

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

```
ts = 2;
Plant = ss(0.8,0.5,0.25,0,ts);
MPCobj = mpc(Plant);
MPCobj.MV(1).Min = -2;
MPCobj.MV(1).Max = 2;
```

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

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

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

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

Set the reference signal. There is no measured disturbance.

```
r = 1;
```

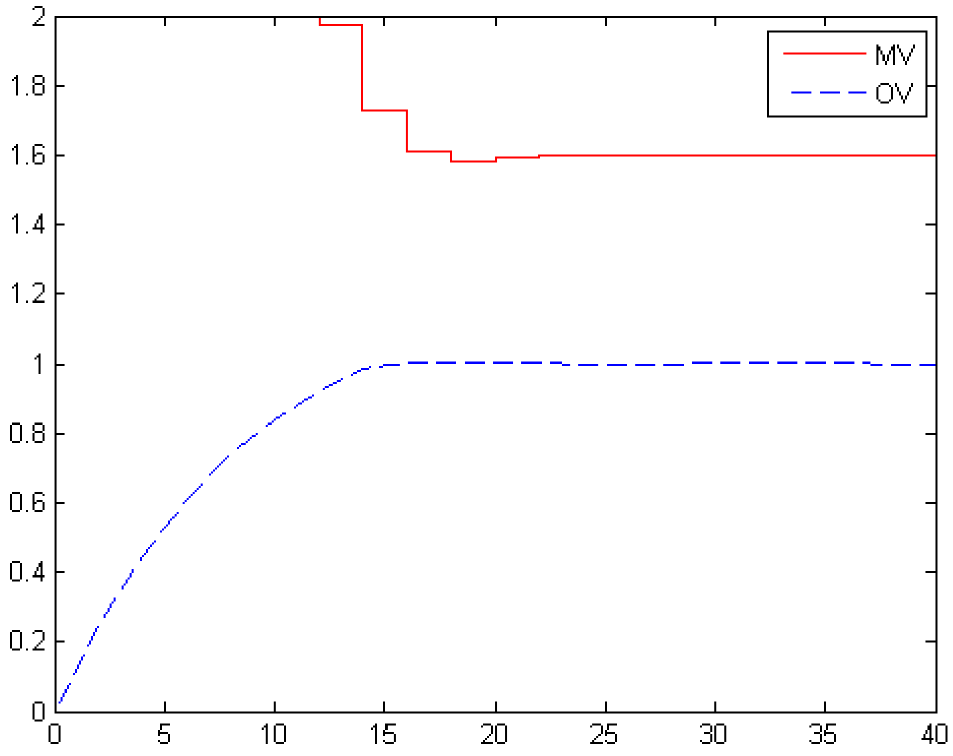
Simulate the closed-loop response by calling `mpcmove` iteratively.

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

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

Analyze the result.

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



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

```
MPCopt = mpcmoveopt;
```

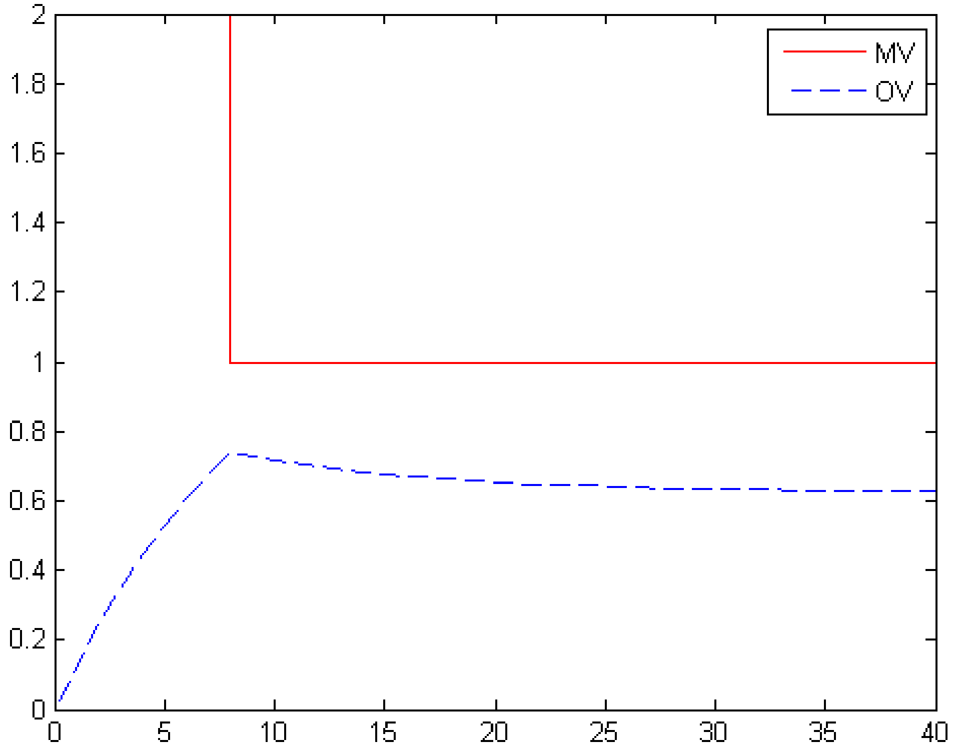
```
MPCopt.MVMin = -2;
MPCopt.MVMax = 2;
```

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

```
x = mpcstate(MPCobj);
y = zeros(N,1);
u = zeros(N,1);
for i=1:N
  % simulated plant and predictive model are identical
  y(i) = 0.25*x.Plant;
  if i == 5
   MPCopt.MVMax = 1;
  end
  u(i) = mpcmove(MPCobj,x,y(i),r,[],MPCopt);
end
```

Analyze the result.

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



Evaluate Scenario at Specific Time Instant

Define a plant model.

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

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

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

-->The "Weights.ManipulatedVariables" property of "mpc" object is empty.
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is emp
-->The "Weights.OutputVariables" property of "mpc" object is empty. Assum
```

Initialize an mpcstate object for simulation from a particular state.

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

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

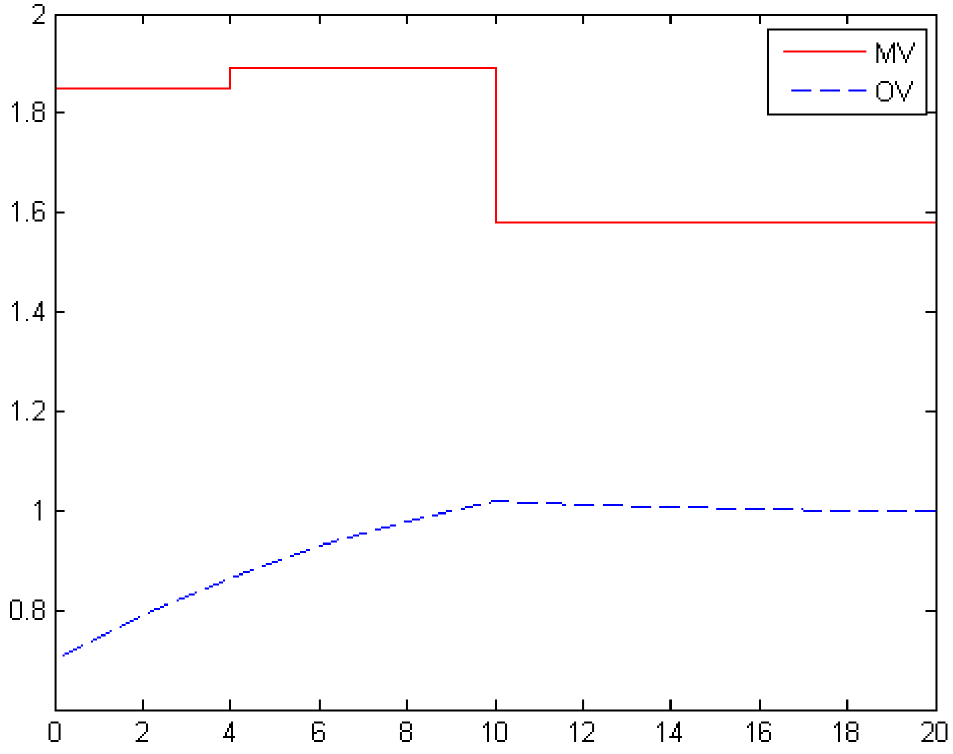
Compute the optimal control at current time.

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

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

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
```

Alternatives

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

See Also

`mpc` | `mpcmoveopt` | `mpcstate` | `review` | `sim` | `setestim` | `getestim`

Tutorials

- MPC Control with Anticipative Action (Look-Ahead)
- MPC Control with Input Quantization Based on Comparing the Optimal Costs
- Analysis of Control Sequences Optimized by MPC on a Double Integrator System

Purpose

Options set for `mpcmove`

Syntax

```
options = mpcmoveopt
```

Description

`options = mpcmoveopt` creates an `mpcmoveopt` object, with default values for its properties. Use this object with `mpcmove` to allow run-time adjustment of the weights and constraints of the Model Predictive Controller, an `mpc` object, that is operating.

Tips

- The number of manipulated variables (n_u) and output variables (n_y) for the `mpcmoveopt` object must match that of the corresponding `mpc` object.
- If any weight property in `options` is set to `[]` (default), `mpcmove` uses the corresponding weight specified in the `mpc` object.
- In general, constraint specifications of `options` must be consistent with those of the corresponding `mpc` object.

  If all of the four constraint properties of `options` are set to `[]` (default), `mpcmove` uses the existing constraints specified for the corresponding `mpc` object. The result will be the same as if there are no run-time constraint changes.

  Otherwise:

  - Constraints set to `[]` are treated as unbounded signals. The corresponding constraint settings for the `mpc` object must also be unbounded.
  - Constraints not set to `[]` are treated as bounded signals. The corresponding constraint settings for the `mpc` object must also be bounded.

    These requirements make the Model Predictive Controller object behave consistently with the corresponding MPC Controller block in Simulink

  A conflict may arise when you have a mixture of bounded and unbounded variables for one or more constraint properties of `options`. It may be resolved by defining compatible constraints for the `mpc` object and corresponding `mpcmoveopt` object.

For example, suppose n_y = 2, and your `mpc` object defines a maximum for the first output variable but the second has no upper bound. If,

- `options.OutputMax` = `[]`, then `mpcmove` interprets this as an attempt to remove the upper bound on the first output.
- `options.OutputMax` is not `[]`, then, because all entries must be finite, `mpcmove` interprets this as an attempt to impose a new upper bound on the second output.

To avoid this conflict, modify your `mpc` object to have a large but finite upper bound on the second output. It can then be paired with `options`, containing finite `OutputMax` values, without conflict.

Output Arguments

options

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

- OutputWeights — 1-by-n_y vector of output variable tuning weights, where n_y is the number of output variables.`mpcmove` replaces the `Weight.OutputVariables` field of the corresponding `mpc` object with this vector. The weights must be finite and real values.
- MVRateWeights — 1-by-n_u vector of manipulated variable rate tuning weights, where n_u is the number of manipulated variables. `mpcmove` replaces the `Weight.ManipulatedVariablesRate` field of the corresponding `mpc` object with this vector. The weights must be finite and real values.
- ECRWeight — Scalar weight on the slack variable used for constraint softening. `mpcmove` replaces the `Weight.ECR` field of the corresponding `mpc` object with this value. The weight must be a finite and real value.
- OutputMin — 1-by-n_y vector of lower bounds on the output variables, where n_y is the number of output variables. `mpcmove` replaces the `OutputVariables(i).Min` field of the corresponding `mpc` object with this vector, for `i = 1,...,ny`.

- OutputMax — 1-by-n_y vector of upper bounds on the output variables, where n_y is the number of output variables. `mpcmove` replaces the `OutputVariables(i).Max` field of the corresponding `mpc` object with this vector, for `i = 1,...,ny`.
- MVMin — 1-by-n_u vector of lower bounds on the manipulated variables, where n_u is the number of manipulated variables. `mpcmove` replaces the `ManipulatedVariables(i).Min` field of the corresponding `mpc` object with this vector, for `i = 1,...,nu`.
- MVMax — 1-by-n_u vector of upper bounds on the manipulated variables, where n_u is the number of manipulated variables. `mpcmove` replaces the `ManipulatedVariables(i).Max` field of the corresponding `mpc` object with this vector, for `i = 1,...,nu`.
- OnlyComputeCost — Logical value to control whether the optimal sequence is to be calculated and exported.
  - `0` (default) sets `mpcmove` to compute and return the optimal cost and the optimal sequence of the Model Predictive Controller objective function.
  - `1` sets `mpcmove` to compute and return only the optimal cost of the Model Predictive Controller objective function.

Examples

Use an `mpcmoveopt` object to vary an MV upper bound during a simulation employing `mpcmove`. The upper bound decreases step-wise from 2 to 1 when elapsed time exceeds 4 seconds. Also, setting `OnlyComputeCost` = `1` causes the `Info` object returned by `mpcmove` to contain the cost only.

Define an `mpcmoveopt` object and initialize some of its fields.

```
OPTobj = mpcmoveopt;
OPTobj.OnlyComputeCost = true;
OPTobj.MVMin = -2;
```

Define a plant model and a model predictive controller object.

```
ts = 2;
```

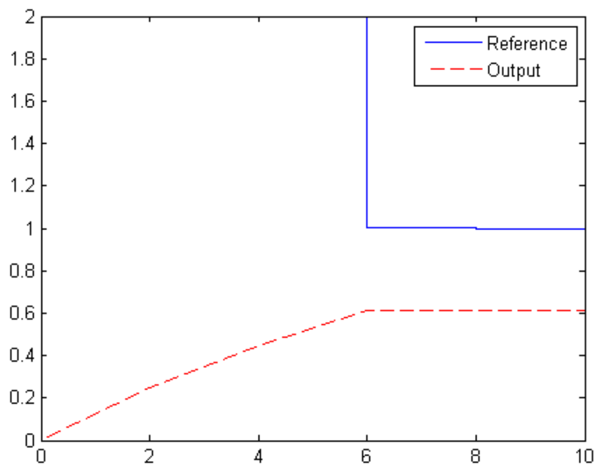
```
Plant = ss(0.8, 0.5, 0.25, 0, ts);
MPCobj = mpc(Plant, ts);
MPCobj.MV(1).Min = -2;
MPCobj.MV(1).Max =  2;
```

Perform `mpcmove` simulation.

```
x = mpcstate(MPCobj);
r = 1;
v = [];
t = [0:ts:10];
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
for i = 1:N
    y(i) = 0.25*x.Plant;
    % Use the MPCMOVEOPT object to modify the MV upper bound in
    % real time.
    if t(i) <= 4
        OPTobj.MVMax = 2;
    else
        OPTobj.MVMax = 1;
    end
    [u(i), Info] = mpcmove(MPCobj, x, y(i), r, v, OPTobj);
end
```

Analyze the results.

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



Alternatives

When you use `mpcmove`, an `mpcmoveopt` object is optional. As an alternative, you can modify the weight, constraint definitions, or both before calling `mpcmove`. This approach is usually less computationally efficient, but it avoids potential conflicts with constraint definitions when mixed bounded and unbounded variables are present.

See Also

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

Purpose Provide help on MPC controller's properties

Syntax

```
mpcprops
```

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

See Also `set` | `get` | `mpchelp`

Purpose

MPC simulation options

Syntax

```
SimOptions=mpcsimopt(mpcobj)
```

Description

`mpcsimopt` creates an `mpcsimopt` object for specifying additional parameters for simulation with `sim`.

`SimOptions=mpcsimopt(mpcobj)` creates an empty object `SimOptions` which is compatible with the MPC object `mpcobj`. You must use `set` / `get` to change simulation options.

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 columns as unmeasured disturbances.

MPC Simulation Options Properties (Continued)

Property	Description
`InputNoise`	Noise on manipulated variables. An array with as many columns as manipulated variables. The last sample of the array is extended constantly over the horizon to obtain the correct size.
`OutputNoise`	Noise on measured outputs. An array with as many columns as measured outputs. The last sample of the array is extended constantly over the horizon to obtain the correct size.
`RefLookAhead`	Preview on reference signal (`'on'` or `'off'`).
`MDLookAhead`	Preview on measured disturbance signal (`'on'` or `'off'`).
`Constraints`	Use MPC constraints (`'on'` or `'off'`).
`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`.

MPC Simulation Options Properties (Continued)

Property	Description
	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'`).
`MVSignal`	Sequence of manipulated variables (with offsets) for open-loop simulation (no MPC action). An array with as many columns as manipulated variables.
`OpenLoop`	Perform open-loop simulation.

Examples

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

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

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

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

% Model for unmeasured input disturbances
```

```
distModel = eye(2,2)*ss(-.5,1,1,0);

% Create MPC object
mpcobj = mpc(plant,1,40,2);
mpcobj.Model.Disturbance = distModel;

% Closed-loop MPC simulation with model mismatch
% and unforeseen unmeasured disturbance inputs

% Define plant model generating the data
p2 = tf(1.5,[0.1 1 2 1])*[1 1; 0 1];
psim = ss([p2 p2 tf(1,[1 1])*[0;1]]);
psim=setmpcsignals(psim,'MV',[1 2],'UD',[3 4 5]);

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

options=mpcsimopt(mpcobj);
options.unmeas=dist;
options.model=psim;

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

See Also

sim

Purpose

Define MPC controller state

Syntax

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

Description

`xmpc = mpcstate(MPCobj,xp,xd,xn,u)` defines an `mpcstate` object for state estimation and optimization in an MPC control algorithm based on the MPC object `MPCobj`. The state of an MPC controller contains the estimates of the states $x(k)$, $x_d(k)$, $x_m(k)$, where $x(k)$ is the state of the plant model, $x_d(k)$ is the overall state of the input and output disturbance model, $x_m(k)$ is the state of the measurement noise model, and the value of the last vector $u(k\text{-}1)$ of manipulated variables. The overall state is updated from the measured output $ym(k)$ by a linear state observer (see “State Observer”).

`xmpc = mpcstate(MPCobj)` returns a default extended initial state that is compatible with the MPC controller `MPCobj`. Such a default state has plant state and previous input initialized at nominal values, and the states of the disturbance and noise models at zero.

Note that `mpcstate` objects are updated by `mpcmove` through the internal state observer based on the extended prediction model.

`xmpc = mpcstate` returns an empty `mpcstate` object.

Properties

The `mpcstate` object type contains the state of an MPC controller. Its properties are listed in MPC State Object Properties on page 3-16.

MPC State Object Properties

Property	Description
`Plant`	Array of plant states. Values are absolute, i.e., they include possible state offsets (cf.`Model.Nominal.X`).
`Disturbance`	Array of states of unmeasured disturbance models. This contains the states of the input disturbance model and, appended below, the states of the unmeasured output disturbances model.
`Noise`	Array of states of measurement noise model.
`LastInput`	Array of previous manipulated variables $u(k\text{-}1)$. Values are absolute, i.e., they include possible input offsets (cf. `Model.Nominal.U`).

The command

```
mpcstate(mpcobj)
```

returns a zero extended initial state compatible with the MPC object `mpcobj`, and with `mpcobj.Plant` and `mpcobj.LastInput` initialized at the nominal values specified in `mpcobj.Model.Nominal`.

See Also

`getoutdist` | `setoutdist` | `setindist` | `getestim` | `setestim` | `ss` | `mpcmove`

Purpose

Start Model Predictive Controller GUI

Syntax

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

Description

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

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

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

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

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

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

See Also

`mpc`

mpcverbosity

Purpose

Change toolbox verbosity level

Syntax

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

Description

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

`mpcverbosity off` turns messages off.

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

`mpcverbosity` just shows the verbosity status.

By default, messages are turned on.

See also "Construction and Initialization" on page 3-13 .

See Also

`mpc`

Purpose

Reduce size of MPC object in memory

Note `pack` will be removed in a future version.

Syntax

```
pack(MPCobj)
```

Description

`pack(MPCobj)` cleans up information build at initialization and stored in the `MPCData` field of the MPC object `MPCobj`. This reduces the amount of bytes in memory required to store the MPC object. For MPC objects based on large prediction models, it is recommended to pack the object before saving the object to file, in order to minimize the size of the file.

See Also

`mpc` | `getmpcdata` | `setmpcdata` | `compare`

Purpose

Plot responses generated by MPC simulations

Syntax

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

Description

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

See Also

`sim` | `mpc`

Purpose

Solve convex quadratic program using Dantzig-Wolfe's algorithm

Note `qpdantz` will be removed in a future version. Use `quadprog` (requires Optimization Toolbox™) instead.

Syntax

```
[xopt,lambda,how]=qpdantz(H,f,A,b,xmin)
[xopt,lambda,how]=qpdantz(H,f,A,b,xmin,maxiter)
```

Description

`[xopt,lambda,how]=qpdantz(H,f,A,b,xmin)` solves the convex quadratic program

$$\min \frac{1}{2} x^T Hx + f^T x$$

subject to $Ax \leq b, x \geq x_{min}$

using Dantzig-Wolfe's active set method [2]. The Hessian matrix `H` should be positive definite. By default, `xmin=1e-3`. Vector `xopt` is the optimizer. Vector `lambda` contains the optimal dual variables (Lagrange multipliers).

The exit flag how is either `'feasible'`, `'infeasible'` or `'unreliable'`. The latter occurs when the solver terminates because the maximum number `maxiter` of allowed iterations was exceeded.

The solver is implemented in `qpsolver.mex`. Dantzig-Wolfe's algorithm uses the direction of the largest gradient, and the optimum is usually found after about $n+q$ iterations, where n=dim(x) is the number of optimization variables, and q=dim(b) is the number of constraints. More than 3($n+q$) iterations are rarely required (see Chapter 7.3 of [2]).

Examples

Solve a random QP problem using `quadprog` from the Optimization Toolbox software and `qpdantz`.

```
n=50;     % Number of vars
H=rand(n,n);H=H'*H;H=(H+H')/2;
```

```
f=rand(n,1);
A=[eye(n);-eye(n)];
b=[rand(n,1);rand(n,1)];
x1=quadprog(H,f,A,b,[],[],-100,[],[],...
optimset('LargeScale','off','Algorithm','active-set'));
[x2,how]=qpdantz(H,f,A,b,-100*ones(n,1));
```

References

[1] Fletcher, R. Practical Methods of Optimization, John Wiley & Sons, Chichester, UK, 1987.

[2] Dantzig, G.B. Linear Programming and Extensions, Princeton University Press, Princeton, 1963.

Purpose

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

Syntax

`review(mpcobj)`

Description

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

- Is the optimization problem to be solved online well defined?
- Is the controller internally stable?
- Is the closed loop system stable when no constraints are active and there is no model mismatch?
- Is the controller able to eliminate steady-state tracking error when no constraints are active?
- Is there a likelihood that constraint definitions will result in an ill-conditioned or infeasible optimization problem?

Tips

- Use `review` iteratively to check your initial MPC design or whenever you make substantial changes to `mpcobj`. Make the recommended changes to your controller to eliminate potential problems.
- If you design your controller using MPC Design Tool, export the controller to the MATLAB Workspace, and analyze it using `review`.
- `review` does not modify `mpcobj`.
- `review` cannot detect all possible performance factors. So, additionally test your design using techniques such as simulations.

Input Arguments

mpcobj

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

Examples

Create a Model Predictive Controller with hard upper and lower bounds on the manipulated variable and its rate-of-change.

Create a discrete Model Predictive Controller.

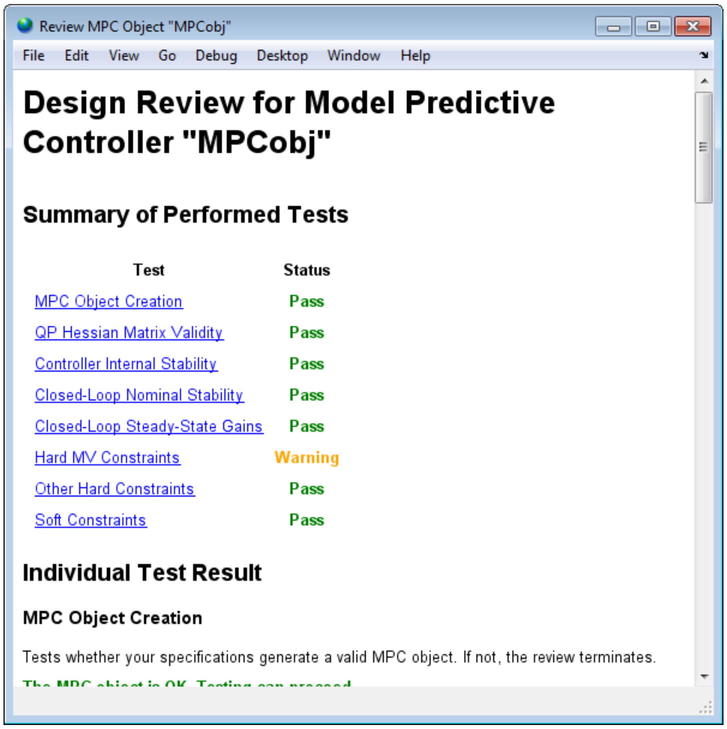
```
% Create a Model Predictive Controller
Plant = tf(1, [10 1]);
ts = 2;
MPCobj = mpc(Plant,ts);
```

Specify hard bounds on the MV 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 design.

```
review(MPCobj)
```


Review MPC Object "MPCobj"
File Edit View Go Debug Desktop Window Help

Design Review for Model Predictive Controller "MPCobj"

Summary of Performed Tests

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

Individual Test Result

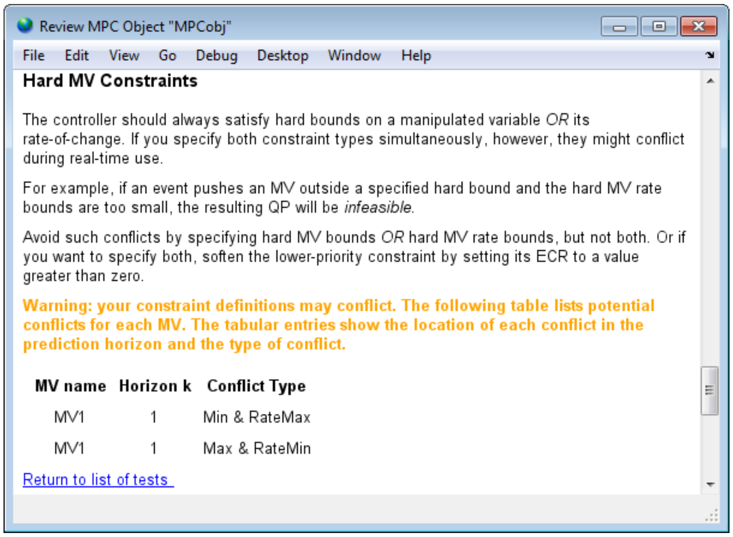
MPC Object Creation

Tests whether your specifications generate a valid MPC object. If not, the review terminates.

The MPC object is OK. Testing can proceed.

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

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


Review MPC Object "MPCobj"
File Edit View Go Debug Desktop Window Help

Hard MV Constraints

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

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

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

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

MV name	Horizon k	Conflict Type
MV1	1	Min & RateMax
MV1	1	Max & RateMin

Return to list of tests

Alternatives

`review` automates certain tests that you could perform yourself.

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

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

See Also

`cloffset` | `mpc` | `ss`

Tutorials

- Reviewing Model Predictive Controller Design for Potential Stability and Robustness Issues
- “Simulation and Code Generation Using Simulink Coder”

Purpose

Compute effect of controller tuning weights on performance

Syntax

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

Description

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

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

`PerfFunc` must be one of the following strings:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

See “Weights” on page 3-7 for details on the tuning weights contained in `MPCobj`.

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

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

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

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

Examples

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

```
PerfFunc = 'IAE';
PerfWts.OutputVariables = [1 0.5];
PerfWts.ManipulatedVariables = zeros(1,3);
PerfWts.ManipulatedVariablesRate = zeros(1,3);
Tstop = 20;
```

```
r = [1 0];
v = [];
simopt = mpcsimopt;
utarget = zeros(1,3);
[J, sens] = sensitivity(MPCobj, PerfFunc, PerfWts, Tstop, ...
 r, v, simopt, utarget)
```

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

See Also

`mpc` | `sim`

Purpose

Set or modify MPC object properties

Syntax

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

Description

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

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

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

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

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

See Also

`mpc` | `get` | `mpcprops`

setconstraint

Purpose

Custom constraints on plant inputs and outputs

Syntax

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

Description

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

$$Eu(k+j|k) + Fy(k+j|k) \leq G$$

where:

$j = 0, \ldots, p$
p is the prediction horizon length
y are the measured and unmeasured outputs
u are the manipulated variables
E, F, and G are constants. Each row of E, F, and G represents a linear constraint to be imposed at each prediction horizon step.

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

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

where,

`V` is a constant representing the Equal Concern for the Relaxation (ECR)
ε is the slack variable used for constraint softening (as in Equation 2-3 of "Standard Form")

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

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

where:

v are the measured disturbances
S is a constant

Tips

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

Input Arguments

MPCobj

MPC controller, specified as an MPC Controller object

E

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

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

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

F

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

n_y columns, where n_y is the number of controlled outputs (measured and unmeasured)
Same number of rows as `E`, `G`, `V`, and `S`

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

G

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

V

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

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

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 decreases the allowed constraint violation, i.e., the constraint becomes harder.

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

Default: vector of 1s

S

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

n_v columns, where n_v is the number of measured disturbances
Same number of rows as `E`, `F`, `G` and `V`

Examples

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

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

1 Formulate this constraint in the required form:

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

2 Specify the constraints on the plant inputs and outputs:

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

Alternatively, you can use:

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

See Also

`setterminal`

Tutorials

- MPC Control with Constraints on a Combination of Input and Output Signals
- MPC Control of a Nonlinear Blending Process

How To

- “Custom Constraints on Inputs and Outputs”

Purpose

Modify MPC object's linear state estimator

Syntax

```
setestim(MPCobj,M)
setestim(MPCobj,'default')
```

Description

The `setestim` function modifies the linear estimator gain of an MPC object. The state estimator is based on the linear model (see "State Estimation").

$x(k + 1) = Ax(k) + B_u u(k) + B_v v(k)$

$y_m(k) = C_m x(k) + D_{vm} v(k)$

where $v(k)$ are the measured disturbances, $u(k)$ are the manipulated plant inputs, $y_m(k)$ are the measured plant outputs, and $x(k)$ is the overall state vector collecting states of plant, unmeasured disturbance, and measurement noise models. The order of the states in x is the following: plant states; disturbance models states; noise model states.

`setestim(MPCobj,M)`, where `MPCobj` is an MPC object, changes the default Kalman estimator gain stored in `MPCobj` to that specified by matrix `M`.

`setestim(MPCobj,'default')` restores the default Kalman gain.

The estimator used in Model Predictive Control Toolbox software is described in "State Estimation". The estimator's equations are as follows.

Predicted Output Computation:

$$\hat{y}_m(k|k-1) = C_m \hat{x}(k|k-1) + D_{vm} v(k)$$

Measurement Update:

$$\hat{x}(k|k) = \hat{x}(k|k-1) + M\left(y_m(k) - \hat{y}_m(k|k-1)\right)$$

Time Update:

$$\hat{x}(k+1|k) = A\hat{x}(k|k) + B_u u(k) + B_v v(k)$$

By combining these three equations, the overall state observer is

$$\hat{x}(k+1|k) = (A - LC_m)\hat{x}(k|k-1) + Ly_m(k) + B_u u(k) + (B_v - LD_{vm})v(k)$$

where $L = AM$.

Note The estimator gain M has the same meaning as the gain L in the `kalman` function in Control System Toolbox software.

Matrices A, B_u, B_v, C_m, D_{vm} can be retrieved using `getestim` as follows:

```
[M,A,Cm,Bu,Bv,Dvm]=getestim(MPCobj)
```

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] = getestim(MPCobj);
```

Calculate the default observer poles.

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


ans =

    0.9402
    0.9402
    0.8816
    0.8816
    0.7430
    0.9020
```

Specify faster observer poles.

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

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

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

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

Obtain the estimator gain from the state-gain matrix.

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

Specify `M` as the estimator for `MPCobj`.

```
setestim(MPCobj,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);
```

See Also

`getestim` | `mpc` | `mpcstate` | `kalman`

Purpose

Modify unmeasured input disturbance model

Syntax

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

Description

`setindist(MPCobj,'integrators')` imposes the default disturbance model for unmeasured inputs, that is, for each unmeasured input disturbance channel, an integrator is added unless there is a violation of observability, otherwise the input is treated as white noise with unit variance (this is equivalent to `MPCobj.Model.Disturbance=[]`).

`setindist(MPCobj,'model',model)` sets the input disturbance model to `model` (this is equivalent to `MPCobj.Model.Disturbance=model`).

See Also

`mpc` | `getindist` | `setestim` | `getestim` | `setoutdist`

Purpose

Set private MPC data structure

Note `setmpcdata` will be removed in a future version. Use `set`, `setconstraint`, `setestim`, `setindist` and`setoutdist` instead.

Syntax

```
setmpcdata(MPCobj,mpcdata)
```

Description

`setmpcdata(MPCobj,mpcdata)` changes the private field `MPCData` of the MPC object `MPCobj`, where all internal QP matrices, models, estimator gains are stored at initalization of the object. You may only need this for very advanced use of Model Predictive Control Toolbox software.

Note Changes to the data structure may easily lead to unpredictable results.

See Also

`getmpcdata` | `set` | `get` | `pack`

setmpcsignals

Purpose

Set signal types in MPC plant model

Syntax

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

Description

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

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

Unambiguous abbreviations of signal types are also accepted.

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

Examples

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

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

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

See Also

`mpc` | `set`

Purpose

Set I/O signal names in MPC prediction model

Syntax

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

Description

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

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

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

See Also

`getname` | `mpc` | `set`

Purpose

Modify unmeasured output disturbance model

Syntax

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

Description

`setoutdist(MPCobj,'integrators')` specifies the default method output disturbance model, based on the specs stored in `MPCobj.OutputVariables.Integrator` and `MPCobj.Weights.OutputVariables`. Output integrators are added according to the following rules:

1. Outputs are ordered by decreasing output weight (in case of time-varying weights, the sum of the absolute values over time is considered for each output channel. In case of equal output weight, the order within the output vector is followed).

2. By following such order, an output integrator is added per measured outputs, unless one of the following is true:

   - There is a violation of observability
   - The corresponding value in `MPCobj.OutputVariables.Integrator` is zero
   - The corresponding value in `MPCobj.Weights.OutputVariables` is zero.

   A warning is issued when an integrator is added to an unmeasured output channel.

`setoutdist(MPCobj,'remove',channels)` removes integrators from the output channels specified in vector `channels`. This corresponds to setting `MPCobj.OutputVariables(channels).Integrator=0`. The default for `channels` is `(1:ny)`, where `ny` is the total number of outputs, that is, all output integrators are removed.

`setoutdist(MPCobj,'model',model)` replaces the array of output integrators designed by default according to

`MPCobj.OutputVariables.Integrator` with the LTI model `model`. The model must have `ny` outputs. If no `model` is specified, then the default model based on the specs stored in `MPCobj.OutputVariables.Integrator` and `MPCobj.Weights.OutputVariables` is used (same as `setoutdist(MPCobj, 'integrators')`).

See Also

`mpc` | `getestim` | `setestim` | `setoutdist` | `setindist`

Purpose

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.

Tips

- Advanced users can impose terminal polyhedral state constraints:

  $K_1 \leq Hx \leq K_2$.

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

Input Arguments

MPCobj

MPC controller, specified as an MPC controller object

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 in Equation 2-3 of "Standard Form"). To apply non-zero off-diagonal terminal weights, you must augment the plant model. See Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation.

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

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

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 in Equation 2-3 of "Standard Form"). To apply non-zero off-diagonal terminal weights, you must augment the plant model. See Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation.

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

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

Pt

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

Default: Prediction horizon *p*

Examples

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

1 Specify a prediction horizon of 8.

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

2 Define a penalty weight and constraints:

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

The constraints and weights include:

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

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

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

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

1 Specify a prediction horizon of 8.

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

2 Define a penalty weight and constraints:

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

The constraints and weights include:

- Diagonal penalty weights of 1 and 10 on the first two output variables
- Lower bounds of 0 and –1 on outputs 1 and 3, none on output 2
- Upper bound at 2 on output 2, none on outputs 1 and 3

- Lower bound at 1 on manipulated variable 1
- No other conditions (weights or bounds) on the manipulated variables

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

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

See Also

`| mpc | mpcprops | setconstraint`

Tutorials

- "Providing LQR Performance Using Terminal Penalty"
- Implementing Infinite-Horizon LQR by Setting Terminal Weights in a Finite-Horizon MPC Formulation

How To

- "Terminal Weights and Constraints"

Purpose

Simulate closed-loop/open-loop response to arbitrary reference and disturbance signals

Syntax

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

Description

The purpose of `sim` is to simulate the 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 between the prediction plant model and the model of the process generating the output data.

`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 object `MPCobj`, and plots the simulation results. `T` is the number of simulation steps. `r` is the reference signal array with as many columns as the number of output variables.

`sim(MPCobj,T,r,v)` also specifies the measured disturbance signal `v`, that has as many columns as the number of measured disturbances.

Note The last sample of `r/v` is extended constantly over the simulation horizon, to obtain the correct size.

`sim(MPCobj,T,r,SimOptions)` specifies the simulation options object `SimOptions`, such as initial states, input/output noise and unmeasured disturbances, plant mismatch, etc. See `mpcsimopt` for details.

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

Without output arguments, `sim` automatically plots input and output trajectories.

`[y,t,u,xp,xmpc,SimOptions] = sim(MPCobj,T,...)` instead of plotting closed-loop trajectories returns the sequence of plant outputs `y`, the time sequence `t` (equally spaced by `MPCobj.Ts`), the sequence `u` of manipulated variables 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 options object `SimOptions` used for the simulation.

The descriptions of the input arguments and their default values are shown in the table below.

Input Argument	**Description**	**Default**
`MPCobj`	MPC object specifying the parameters of the MPC control law	None
`T`	Number of simulation steps	Largest row-size of `r,v,d,n`
`r`	Reference signal	`MPCobj.Model.Nominal.Y`
`v`	Measured disturbance signal	Entries from `MPCobj.Model.Nominal.U`
`SimOptions`	Object of class `@mpcsimopt` containing the simulation parameters (See `mpcsimopt`)	`[]`

`r` is an array with as many columns as outputs, `v` is an array with as many columns as measured disturbances. The last sample of `r/v/d/n` is extended constantly over the horizon, to obtain the correct size.

The output arguments of `sim` are detailed below.

Output Argument	Description
`y`	Sequence of controlled plant outputs (without noise added on measured ones)
`t`	Time sequence (equally spaced by `MPCobj.Ts`)
`u`	Sequence of manipulated variables generated by MPC
`xp`	Sequence of states of plant model (from `Model` or `SimOptions.Model`)
`xmpc`	Sequence of states of MPC controller (estimates of the extended state). This is a structure with the same fields as the `mpcstate` 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 P
-->The "ControlHorizon" property of the "mpc" object is empty. Assumin
-->The "Weights.ManipulatedVariables" property of "mpc" object is empt
-->The "Weights.ManipulatedVariablesRate" property of "mpc" object is 
-->The "Weights.OutputVariables" property of "mpc" object is empty. As
```

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.

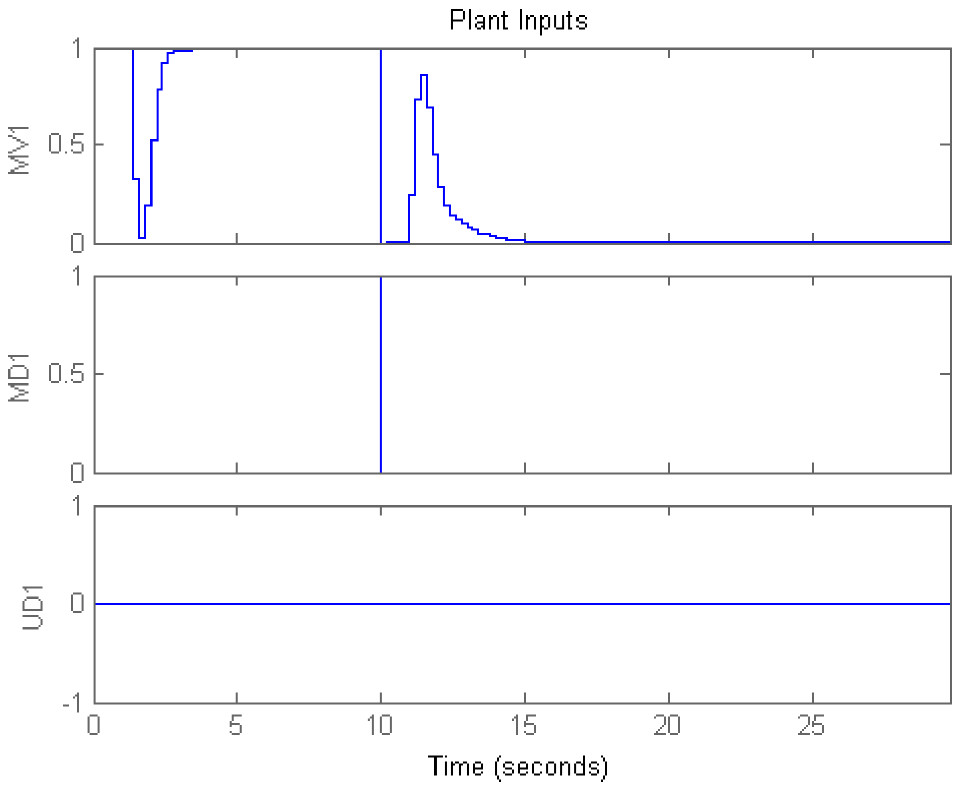
```
num_sim_steps = round(Tstop/Ts);
r = ones(num_sim_steps,1);
v = [zeros(num_sim_steps/3,1); ones(2*num_sim_steps/3,1)];
```

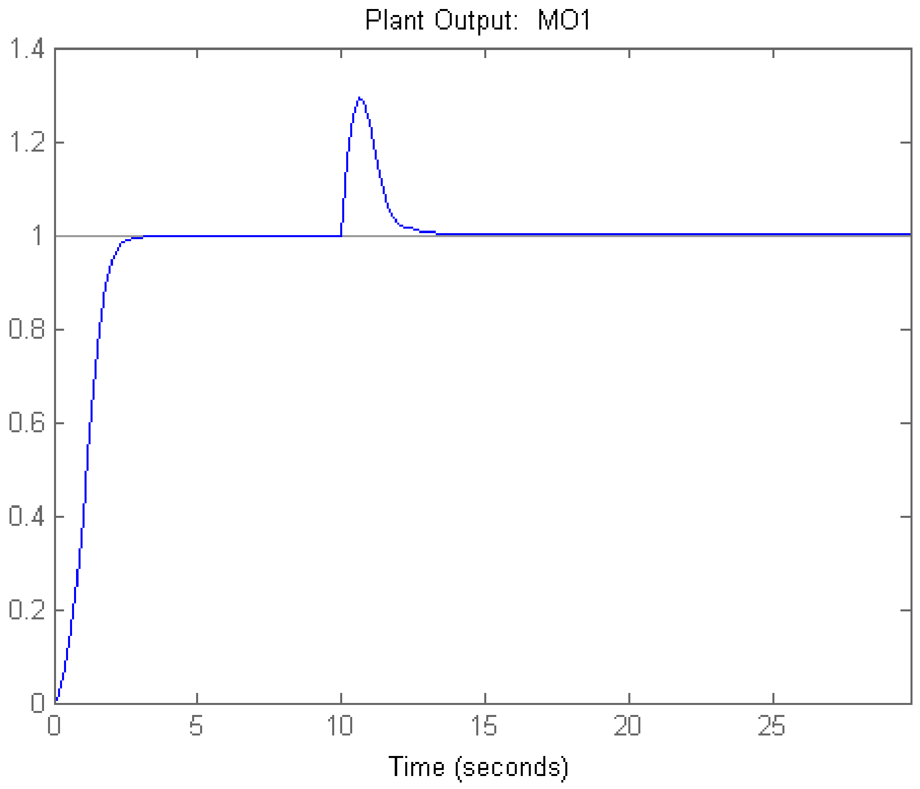
The reference signal, `r`, is a unit step. The measured disturbance signal, `v`, is a unit step, with a 10 unit delay.

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.
-->The "Model.Noise" property of the "mpc" object is empty. Assuming w
```


Plant Inputs
MV1
MD1
UD1
Time (seconds)
1
0.5
0
-1
5
10
15
20
25



See Also `mpcsimopt` | `mpc` | `mpcmove`

Purpose

Size and order of MPC Controller

Syntax

```
mpc_obj_size = size(MPCobj)
mpc_obj_size = size(MPCobj,signal_type)
size(MPCobj)
```

Description

`mpc_obj_size = size(MPCobj)` returns a row vector specifying the number of manipulated inputs and measured controlled outputs of an MPC controller. This row vector contains the elements [n_u n_{ym}], where n_u is the number of manipulated inputs and n_{ym} is the number of measured controlled outputs.

`mpc_obj_size = size(MPCobj,signal_type)` returns the number of signals of the specified type that are associated with the MPC controller.

You can specify `signal_type` as one of the following strings:

- `'uo'` — Unmeasured controlled outputs
- `'md'` — Measured disturbances
- `'ud'` — Unmeasured disturbances
- `'mv'` — Manipulated variables
- `'mo'` — Measured controlled outputs

`size(MPCobj)` displays the size information for all the signal types of the MPC controller.

See Also

`mpc` | `set`

Purpose

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 MPC controller `MPCobj` when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity analysis and other linear analysis.

`sys=ss(MPCobj)` returns the linear discrete-time dynamic controller `sys`

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

$u(k) = Cx(k) + Dy_m(k)$

where y_m is the vector of measured outputs of the plant, and u is the vector of manipulated variables. The sampling time of controller `sys` is `MPCobj.Ts`.

Note Vector x includes the states of the observer (plant+disturbance+noise model states) and the previous manipulated variable $u(k\text{-}1)$.

`sys = ss(MPCobj,signals)` returns the linearized MPC controller in its full form and allows you to specify the signals that you want to include as inputs for `sys`.

The full form of the MPC controller has the following structure:

$x(k+1) = Ax(k) + By_m(k) + B_r r(k) + B_v v(k) + B_{ut} u_{target}(k) + B_{off}$

$u(k) = Cx(k) + Dy_m(k) + D_r r(k) + D_v v(k) + D_{ut} u_{target}(k) + D_{off}$

Here, r is the vector of setpoints for both measured and unmeasured plant outputs, v is the vector of measured disturbances, u_{target} is the vector of preferred values for manipulated variables.

Specify `signals` as a single or multicharacter string constructed using any of the following:

- `'r'` — Output references
- `'v'` — Measured disturbances
- `'o'` — Offset terms
- `'t'` — Input targets

For example, to obtain a controller that maps $[y_m; r; v]$ to u, use:

```
sys = ss(MPCobj,'rv');
```

In the general case of nonzero offsets, y_m (as well as r, v, and u_{target}) must be interpreted as the difference between the vector and the corresponding offset. Offsets can be nonzero is `MPCobj.Model.Nominal.Y` or `MPCobj.Model.Nominal.U` are nonzero.

Vectors B_{off}, D_{off} are constant terms. They are nonzero if and only if `MPCobj.Model.Nominal.DX` is nonzero (continuous-time prediction models), or `MPCobj.Model.Nominal.Dx-MPCobj.Model.Nominal.X` is nonzero (discrete-time prediction models). In other words, when `Nominal.X` represents an equilibrium state, B_{off}, D_{off} are zero.

Only the following fields of `MPCobj` are used when computing the state-space model: `Model`, `PredictionHorizon`, `ControlHorizon`, `Ts`, `Weights`.

`sys = ss(MPCobj,signals,ref_preview,md_preview)` specifies if the MPC controller has preview actions on the reference and measured disturbance signals. If the flag `ref_preview='on'`, then matrices B_r and D_r multiply the whole reference sequence:

$x(k + 1) = Ax(k) + By_m(k) + B_r[r(k);r(k + 1);...;r(k + p - 1)] +...$

$u(k) = Cx(k) + Dy_m(k) + D_r[r(k);r(k + 1);...;r(k + p- 1)] +...$

Similarly if the flag `md_preview='on'`, then matrices B_v and D_v multiply the whole measured disturbance sequence:

$x(k + 1) = Ax(k) +\ldots+ B_v[v(k);v(k + 1);\ldots;v(k + p)] +\ldots$

$u(k) = Cx(k) +\ldots+ D_v[v(k);v(k + 1);\ldots;v(k + p)] +\ldots$

`[sys,ut] = ss(MPCobj)` additionally returns the input target values for the full form of the controller.

`ut` is returned as a vector of doubles, `[utarget(k); utarget(k+1); ... utarget(k+h)]`.

Here:

- *h* — Maximum length of previewed inputs, that is, `h = max(length(MPCobj.ManipulatedVariables(:).Target)`
- `utarget` — Difference between the input target and corresponding input offsets, that is, `MPCobj.ManipulatedVariables(:).Targets - MPCobj.Model.Nominal.U`

Examples

Convert Unconstrained MPC Controller to State-Space Model

To improve the clarity of the example, suppress messages about working with an MPC controller.

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

Create the plant model.

```
G = rss(5,2,3);
G.D = 0;
G = setmpcsignals(G,'mv',1,'md',2,'ud',3,'mo',1,'uo',2);
```

Configure the MPC controller with nonzero nominal values, weights, and input targets.

```
C = mpc(G,0.1);
C.Model.Nominal.U = [0.7 0.8 0];
C.Model.Nominal.Y = [0.5 0.6];
```

```
C.Model.Nominal.DX = rand(5,1);
C.Weights.MV = 2;
C.Weights.OV = [3 4];
C.MV.Target = [0.1 0.2 0.3];
```

`C` is an unconstrained MPC controller. Specifying `C.Model.Nominal.DX` as nonzero means that the nominal values are not at steady state. `C.MV.Target` specifies three preview steps.

Covert `C` to a state-space model.

```
sys = ss(C);
```

The output, `sys`, is a seventh-order SISO state-space model. The seven states include the five plant model states, one state from the default input disturbance model, and one state from the previous move, `u(k-1)`.

Restore `mpcverbosity`.

```
mpcverbosity(old_status);
```

See Also

`mpc` | `set` | `tf` | `zpk`

Purpose

Convert unconstrained MPC controller to linear transfer function

Syntax

```
sys=tf(MPCobj)
```

Description

The `tf` function computes the transfer function of the linear controller `ss(MPCobj)` as an LTI system in `tf` form corresponding to the MPC controller when the constraints are not active. The purpose is to use the linear equivalent control in Control System Toolbox software for sensitivity and other linear analysis.

See Also

`ss` | `zpk`

trim

Purpose

Compute steady-state value of MPC controller state for given inputs and outputs

Syntax

```
x = trim(MPCobj,y,u)
```

Description

The `trim` function finds a steady-state value for the plant state or the best approximation in a least squares sentence such that:

$$\begin{aligned} x - x_{off} &= A(x - x_{off}) + B(u - u_{off}) \\ y - y_{off} &= C(x - x_{off}) + D(u - u_{off}) \end{aligned}$$

Here, x_{off}, u_{off}, and y_{off} are the nominal values of the extended state x, input u, and output y.

`x` is returned as an `mpcstate` object. Specify `y` and `u` as doubles. `y` specifies the measured and unmeasured output values. `u` specifies the manipulated variable, measured disturbance, and unmeasured disturbance values. The values for unmeasured disturbances must be `0`.

`trim` assumes the disturbance model and measurement noise model to be zero when computing the steady-state value. The software uses the extended state vector to perform the calculation.

See Also

`mpc` | `mpcstate`

Purpose

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`

2

Block Reference

MPC Controller

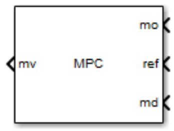
Purpose

Compute MPC control law

Library

MPC Simulink Library

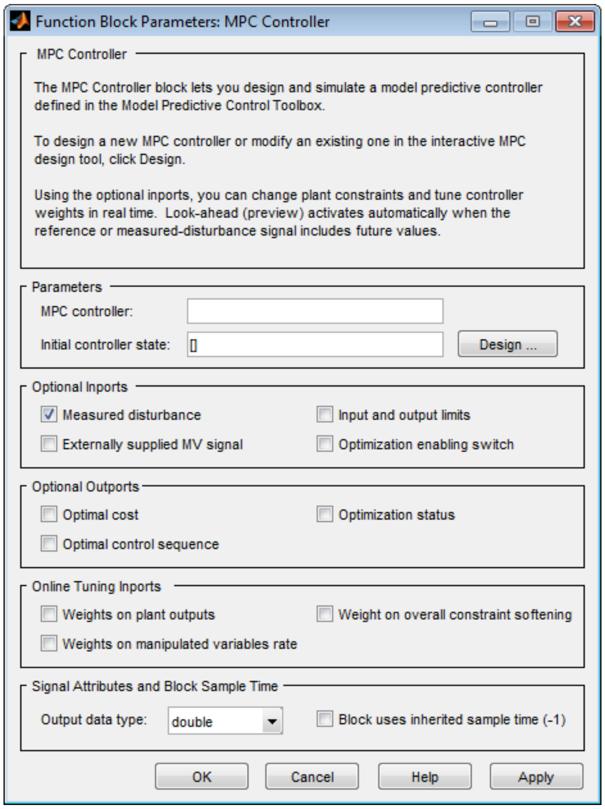
Description



The MPC Controller block receives the current measured output signal (`mo`), reference signal (`ref`), and optional measured disturbance signal (`md`). The block computes the optimal manipulated variables (`mv`) by solving a quadratic program (QP).

To use the block in simulation and code generation, you must specify an `mpc` object, which defines a model predictive controller. This controller must have already been designed for the plant that it will control.

Because the MPC Controller block uses MATLAB Function blocks to implement the QP solver, it requires compilation each time you change the MPC object and block. Also, because MATLAB does not allow compiled code to reside in any MATLAB product folder, you must use a non-MATLAB folder to work on your Simulink model when you use MPC blocks.


Function Block Parameters: MPC Controller

MPC Controller

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

To design a new MPC controller or modify an existing one in the interactive MPC design tool, click Design.

Using the optional inports, you can change plant constraints and tune controller weights in real time. Look-ahead (preview) activates automatically when the reference or measured-disturbance signal includes future values.

Parameters

MPC controller:

Initial controller state: []

Optional Inports

[✓] Measured disturbance
[ ] Input and output limits
[ ] Externally supplied MV signal
[ ] Optimization enabling switch

Optional Outports

[ ] Optimal cost
[ ] Optimization status
[ ] Optimal control sequence

Online Tuning Inports

[ ] Weights on plant outputs
[ ] Weight on overall constraint softening
[ ] Weights on manipulated variables rate

Signal Attributes and Block Sample Time

Output data type: double

[ ] Block uses inherited sample time (-1)

Dialog Box

The MPC Controller block has the following parameter groupings:

- “Parameters” on page 2-4
- “Required Inports” on page 2-5
- “Required Outports” on page 2-6
- “Optional Inports” on page 2-6
- “Optional Outports” on page 2-10
- “Online Tuning Inports” on page 2-12

- “Signal Attributes and Block Sample Time” on page 2-14

Parameters

MPC controller

You must provide an `mpc` object that defines your controller using one of the following methods:

- Enter the name of an `mpc` object in the **MPC Controller** edit box. This object must be present in the base workspace.

  Clicking **Design** opens the MPC design tool where you can modify the controller settings in a graphical environment. For example, you can:

  - Import a new prediction model.
  - Change horizons, constraints, and weights.
  - Evaluate MPC performance with a linear plant.
  - Export the updated controller to the base workspace.

  To see how well the controller works for the nonlinear plant, run a closed-loop Simulink simulation.

- If you do not have an existing `mpc` object in the base workspace, leave the **MPC controller** field empty and, with the MPC Controller block connected to the plant, click **Design**. This action constructs a default `mpc` controller by obtaining a linearized model from the Simulink diagram at the default operating point. Continue your controller design in the MPC design tool.

  To use this design approach, you must have Simulink Control Design™ software.

Initial controller state

Specifies the initial controller state. If this parameter is left blank, the block uses the nominal values that are defined in the `Model.Nominal` property of the `mpc` object. To override the default, create an `mpcstate` object in your workspace that represents the initial state, and enter its name in the field.

Required Inports

Measured output

At each control instant, the mo signal must contain the current output variable measurements. Let n_{ym} be the number of measured output variables (MO) defined in your predictive controller. If n_{ym}=1, connect a scalar signal to the mo inport. Otherwise, connect a row or column vector signal containing n_{ym} real, double-precision elements.

Reference

At each control instant, the ref signal must contain the current reference values (targets or setpoints) for the n_y output variables (ny = n_{ym}+ number of unmeasured outputs). You have the option to specify future reference values (previewing).

The ref signal must be size N by n_y, where $N(1 \le N \le p)$ is the number of time steps for which you are specifying reference values and p is the prediction horizon. Each element must be a real double-precision number. The ref dimension must not change from one control instant to the next.

When N=1, you cannot preview. To specify future reference values, choose N such that $1 < N \le p$ to enable previewing. Doing so usually improves performance via feedforward information. The first row specifies the n_y references for the first step in the prediction horizon (at the next control interval k=1), and so on for N steps. If N<p, the last row designates constant reference values to be used for the remaining p-N steps.

For example, suppose n_y=2 and p=6. At a given control instant, the signal connected to the controller's ref inport is

```
[2 5     k=1
 2 6     k=2
 2 7     k=3
 2 8]     k=4
```

The signal informs the controller that:

- Reference values for the first prediction horizon step (k=1) are 2 and 5.
- The first reference value remains at 2, but the second increases gradually.
- The second reference value becomes 8 at the beginning of the fourth step (k=4) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 5–6 of the prediction horizon.

mpcpreview shows how to use reference previewing in a specific case. For calculation details on the use of the reference signal, see "Optimization Problem".

Required Outports

Manipulated Variables

The mv outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its mv outport by solving a quadratic program at each control instant. The elements are real, double-precision values.

Optional Inports

Measured disturbance

Add an inport (md) to which you can connect a measured disturbance signal.

Your measured disturbance signal (MD) must be size Nxn_{md}, where $n_{md}(\geq 1)$ is the number of measured disturbances defined in your Model Predictive Controller and $N\ (1 \leq N \leq p+1)$ is the number of time steps for which the MD is known. Each element must be a real, double-precision number. The signal dimensions must not change from one control instant to the next.

If N=1 you cannot preview. At each control instant, the MD signal must contain the most recent measurements at the current time k=0 (as a row vector, length n_{md}). The controller assumes that the MDs remain constant at their current values for the entire prediction horizon.

If you are able to predict future MD values, choose N such that $1<N \leq p+1$ to enable previewing. Doing so usually improves performance via feedforward. In this case, the first row must contain the n_{md} current values at k=0, and the remaining rows designate variations over the next N-1 control instants. If N<p+1, the last row designates constant MD values to be used for the remaining p+1-N steps of the prediction horizon.

For example suppose n_{md}=2 and p=6. At a given control instant, the signal connected to the controller's md inport is:

```
[2 5      k=0
 2 6      k=1
 2 7      k=2
 2 8]     k=3
```

This signal informs the controller that:

- The current MDs are 2 and 5 at k=0.
- The first MD remains at 2, but the second increases gradually.
- The second MD becomes 8 at the beginning of the step 3 (k=3) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 4–6 of the prediction horizon.

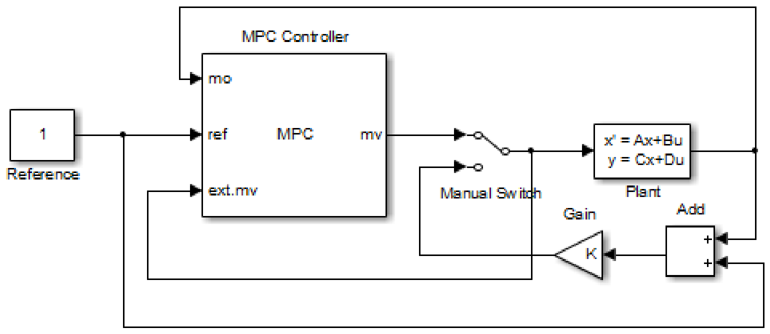
mpcpreview shows how to use MD previewing in a specific case.

For calculation details, see "Prediction Model" and "QP Matrices".

Externally supplied MV signals

Add an inport (ext.mv), which you can connect to the actual manipulated variables (MV) used in the plant. The block uses these to update its internal state estimates. For example, suppose the actual signals saturate at physical limits or the `MV` is under manual control. In both cases, feeding the actual value back to the MPC Controller block can improve performance significantly, because the prediction model's state estimates are updated more accurately.

The following example shows how a manual switch may override the controller's output. Also see Turning Controller Online and Offline with Bumpless Transfer.



Do not connect this option to leave the ext.mv inport unconnected. In either case, the model predictive controller assumes that the plant uses the MV signals sent by the MPC Controller block. In the preceding example, the external MV signal always provides the model predictive controller that the control signal actually used in the plant. Otherwise, the model predictive controller's internal state estimate would be inaccurate.

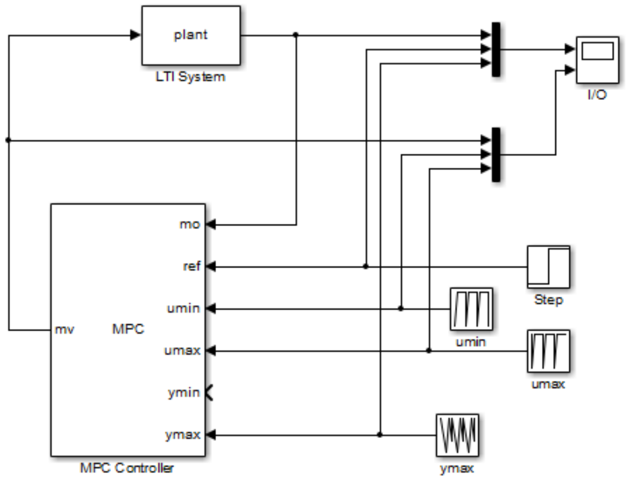
Note The MPC Controller block is a discrete-time block with sampling time inherited from the MPC object. The MPC block has direct feedthrough from measured outputs (`mo`), output references (`ref`), and measured disturbances (`md`) to MPC-manipulated variables (`mv`). There is no direct feedthrough from externally supplied manipulated variables (`ext.mv`) to MPC-manipulated variables (`mv`).

Input and output limits

Add inports (`umin`,`umax`,`ymin`,`ymax`), which you can connect to run-time constraint signals. If this check box is not selected, the block uses the constant constraint values stored within its `mpc` object. Example connections appear in the following model. See Varying Input and Output Constraints for an example of using this option.

Each unconnected limit inport, such as `ymin` in the following model, is treated as an unbounded signal. The corresponding constraint settings in the `mpc` object must also be unbounded. For connected limit inports, such as `ymax`, the signals must be finite and the corresponding variables in the `mpc` object must also be bounded.

All constraint signals connected to the block must be finite. Also, you cannot change the number or identity of constrained and unconstrained variables. For example, if your `mpc` object specifies that your first MV has a lower bound, you must supply a `umin` signal for it.



Optimization enabling switch

Add an inport (`QP Switch`) whose input specifies whether the controller performs optimization calculations. If the input signal is zero, the controller behaves normally. If the input signal becomes nonzero, the MPC Controller block turns off the controller's optimization calculations and sets the controller output to zero. These actions save computational effort when the controller output is not needed, such as when the system has been placed in manual operation or another controller has taken over. The controller, however, continues to update its internal state estimate in the usual way. Thus, it is ready to resume optimization calculations whenever the `QP Switch` signal returns to zero.

If you select this option, the mask automatically selects the **Externally supplied MV signal** option. Connect this option to the current MV value in the plant. Otherwise, there would be a "bump" each time the `QP Switch` signal reactivates optimization.

Optional Outports

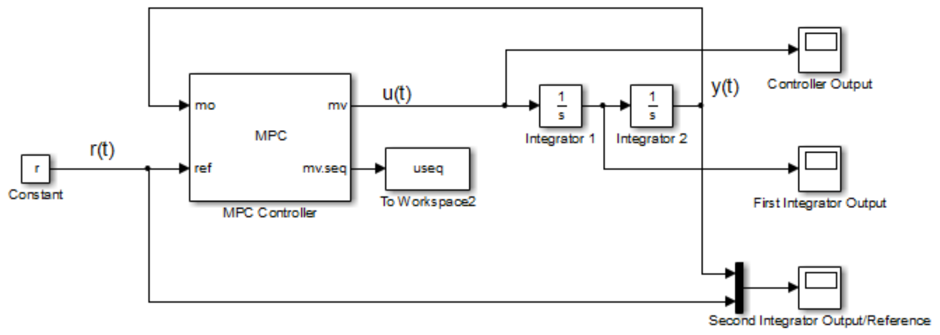
Optimal cost

Add an outport (`cost`) that provides the calculated optimal cost (scalar) of the quadratic program during operation. The computed value is an indication of controller performance. If the controller is performing well, the value is low. However, if the optimization problem is infeasible, this value is meaningless. (See `qp.status`.)

Optimal control sequence

Add an outport (`mv.seq`) that provides the controller's computed optimal `MV` sequence for the entire prediction horizon from `k=0` to `k=p-1`. If n_u is the number of MVs and p is the length of the prediction horizon, this signal is a p by n_u matrix. The first row represents `k=0` and duplicates the block's MV outport.

The following block diagram (from Analysis of Control Sequences Optimized by MPC on a Double Integrator System) illustrates how to use this option. The diagram shows how to collect diagnostic data and send it to the To Workspace2 block, which creates the variable, `useq`, in the workspace. Run the example to see how the optimal sequence evolves with time.



Optimization status

Add an outport (`qp.status`) that allows you to monitor the status of the QP solver.

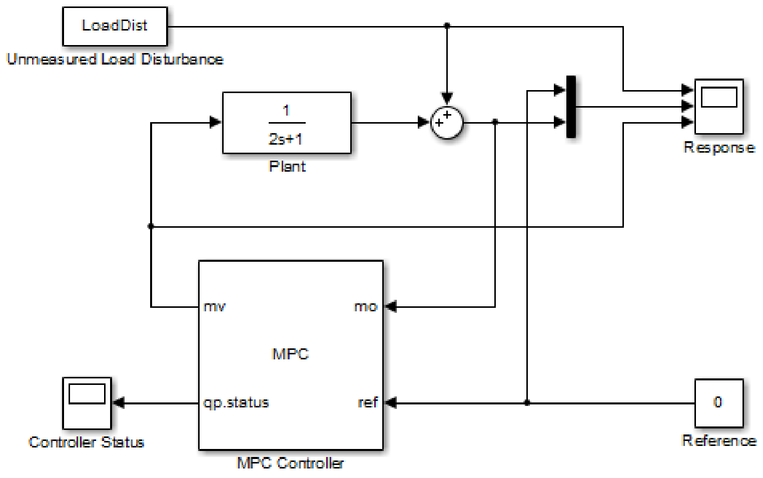
If a QP problem is solved successfully at a given control interval, the `qp.status` output returns the number of QP solver iterations used in computation. This value is a finite, positive integer and is proportional to the time required for the calculations. Thus, a large value means a relatively slow block execution at this time interval.

The QP solver may fail to find an optimal solution for the following reasons:

- `qp.status = 0` — The QP solver cannot find a solution within the maximum number of iterations specified in the `mpc` object.
- `qp.status = -1` — The QP solver detects an infeasible QP problem. See Monitoring Optimization Status to Detect Controller Failures for an example where a large, sustained disturbance drives the OV outside its specified bounds.
- `qp.status = -2` — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem.

For all the previous three failure modes, the MPC block holds its `mv` output at the most recent successful solution. In a real-time application, you can use status indicator to set an alarm or take other special action.

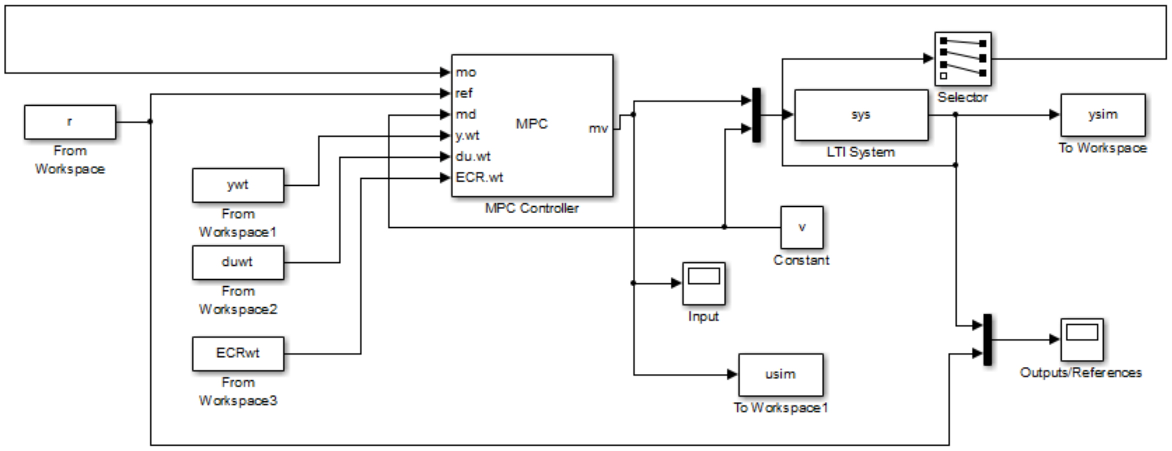
The next diagram shows how to use the status indicator to monitor the MPC Controller block in real time. See Monitoring Optimization Status to Detect Controller Failures for more details.



Online Tuning Inports

A controller intended for real-time applications should have “knobs” you can use to tune its performance when it operates with the real plant. This group of optional inports serves that purpose.

The diagram shown below displays the MPC Controller block’s three tuning knobs. In this simulation context, the knobs are being tuned by prestored signals (the ywt, duwt, and ECRwt variables in the From Workspace blocks). In practice, you would connect a knob or similar manual adjustment.



Weights on plant outputs

Add an inport (`y.wt`) whose input is a vector signal defining a nonnegative weight for each controlled output variable (OV). This signal overrides the `MPCobj.Weights.OV` property, which establishes the relative importance of OV reference tracking.

For example, if the preceding controller defined 3 OVs, the signal connected to the `y.wt` inport should be a vector with 3 elements. If the second element is relatively large, the controller would place a relatively high priority on making OV(2) track the r(2) reference signal. Setting a `y.wt` signal to zero turns off reference tracking for that OV.

If you do not connect a signal to the `y.wt` inport, the block uses the OV weights specified in your MPC object, and these values remain constant.

Weights on manipulated variables rate

Add an inport (`du.wt`), whose input is a vector signal defining *nu* nonnegative weights, where *nu* is the number of manipulated variables (MVs). The input overrides the `MPCobj.Weights.MVrate` property stored in the `mpc` object.

For example, if your controller defines four MVs and the second `du.wt` element is relatively large, the controller would use relatively small changes in the second MV. Such move suppression makes the controller less aggressive. However, too much suppression makes it sluggish.

If you do not connect a signal to the `du.wt` inport, the block uses the `MVrate` weights property specified in your `mpc` object, and these values remain constant.

Weight on overall constraint softening

Add an inport (`ECR.wt`), whose input is a scalar nonnegative signal that overrides the MPC Controller block's `MPCobj.Weights.ECR` property. This inport has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero.

If there are soft constraints, increasing the `ECR.wt` value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

You may not be able to avoid violations of an output variable constraint. Thus, increasing the `ECR.wt` value is often counterproductive. Such an increase causes the controller to pay less attention to its other objectives and does not help reduce constraint violations. You usually need to tune `ECR.wt` to achieve the proper balance in relation to the other control objectives.

Signal Attributes and Block Sample Time

Output data type

Specify the data type of the manipulated variables (MV) as one of the following:

- `double` — Double-precision floating point (default).
- `single` — Single-precision floating point.

  You specify the output data type as `single` if you are implementing the model predictive controller on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see "Simulation and Code Generation Using Simulink Coder".

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**. For more information, see "Display Port Data Types".

Block uses inherited sample time (-1)

Use the sample time inherited from the parent subsystem as the MPC Controller block's sample time.

Inheriting the sample time allows you to conditionally execute the MPC Controller block inside the Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note When you place an MPC controller inside a Function-Call Subsystem or Triggered Subsystem block, you must execute the subsystem at the controller's design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink Editor, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see "View Sample Time Information".

See Also

`mpc` | `mpcstate` | Multiple MPC Controllers

Related Examples

- MPC Control with Input Quantization Based on Comparing the Optimal Costs
- Analysis of Control Sequences Optimized by MPC on a Double Integrator System
- "Simulation and Code Generation Using Simulink Coder"
- "Simulation and Structured Text Generation Using PLC Coder"

Multiple MPC Controllers

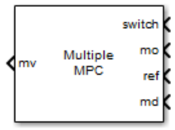
Purpose

Simulate switching between multiple MPC controllers

Library

MPC Simulink Library

Description

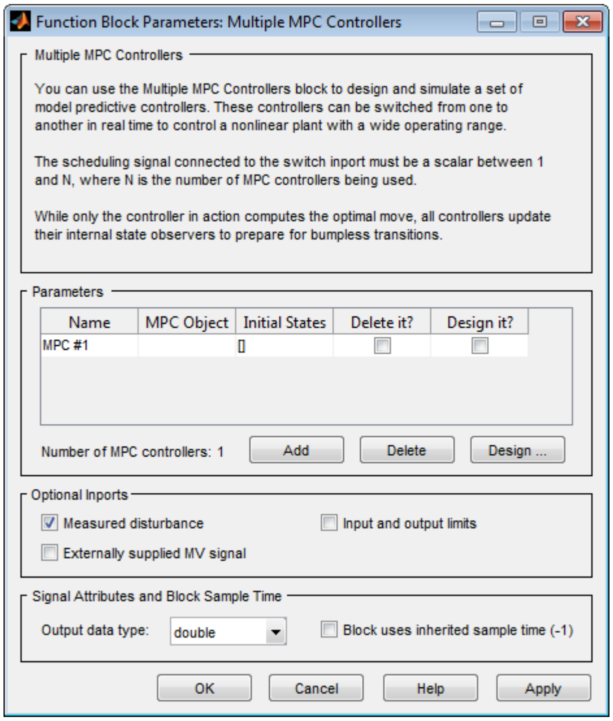


The Multiple MPC Controllers block receives the current measured output, reference signal, and measured disturbance signal. It then solves a quadratic program to calculate the optimal manipulated variables. It also receives a switching signal that designates which one of two or more controllers is to perform the calculation (i.e., the active controller). The block contains these controllers as MPC objects, each of which is designed for a particular operating region of a nonlinear plant.

The Multiple MPC Controllers block allows you to achieve better control of a nonlinear plant over a range of operating conditions. A controller that works well initially can degrade if the plant is nonlinear and its operating point changes. In conventional feedback control, you might compensate for this degradation by gain scheduling. In a similar manner, the Multiple MPC Controllers block allows you to transition between multiple MPC controllers in real time in a preordained manner. You design each controller to work well in a particular region of the operating space. When the plant moves away from this region, you instruct another MPC controller to take over.

The Multiple MPC Controllers block does not provide all the optional features found in the MPC Controller block. The following ports are currently not available:

- Optional outports such as optimal cost, optimal control sequence, and optimization status
- Optional inports for online tuning


Function Block Parameters: Multiple MPC Controllers

Multiple MPC Controllers

You can use the Multiple MPC Controllers block to design and simulate a set of model predictive controllers. These controllers can be switched from one to another in real time to control a nonlinear plant with a wide operating range.

The scheduling signal connected to the switch inport must be a scalar between 1 and N, where N is the number of MPC controllers being used.

While only the controller in action computes the optimal move, all controllers update their internal state observers to prepare for bumpless transitions.

Parameters

Name	MPC Object	Initial States	Delete it?	Design it?
MPC #1		[]	☐	☐

Number of MPC controllers: 1

Optional Inports

☑ Measured disturbance
☐ Input and output limits
☐ Externally supplied MV signal

Signal Attributes and Block Sample Time

Output data type: double
☐ Block uses inherited sample time (-1)

Dialog Box

The MPC Controller block has the following parameter groupings:

- "Parameters" on page 2-18
- "Required Inports" on page 2-19
- "Required Outports" on page 2-20
- "Optional Inports" on page 2-20
- "Signal Attributes and Block Sample Time" on page 2-24

Parameters

MPC Object List

The table is an ordered list of MPC objects. The first row designates the controller to be used when the `switch` input equals a certain number. The first designates which controller is used when the switch input equals 1, the second when the switch input equals 2, and so on. These controllers must refer to objects that already exist in your base workspace.

Note After entering each MPC object name, press **Enter**. Also press **Enter** after editing an object name.

Use **Add** and **Delete** to add and remove rows. When deleting, indicate any rows to delete using the **Delete It** check box.

When the edit box is empty, and the block is connected to the plant, clicking the **Design** button constructs a default MPC controller. This controller is constructed using a linearized plant model from the Simulink diagram. This action also opens the design tool so you can modify the default behavior.

You can also start the design tool by selecting one or more MPC objects using the **Design It** check box and then clicking **Design**. All selected MPC objects are loaded into the design tool where you can review and edit their properties.

Initial controller state

Initial state of each MPC object in the ordered list. Each must be a valid `mpcstate` object. If no value is supplied, the default is the nominal value defined in the `Model.Nominal` property of the `mpc` object.

Required Inports

Controller Selection

The `switch` input signal must be a scalar integer between 1 and n_c, where n_c is the number of controllers listed in your block mask. At each control instant, this signal designates the controller that will be used.

Measured output

At each control instant, the `mo` signal must contain the current output variable measurements. Let n_{ym} be the number of measured output variables (MO) defined in your predictive controller. If $n_{ym}=1$, connect a scalar signal to the `mo` inport. Otherwise, connect a row or column vector signal containing n_{ym} real, double-precision elements.

Reference

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables (`ny =` n_{ym}`+ number of unmeasured outputs`). You have the option to specify future reference values (previewing).

The `ref` signal must be size `N` by n_y, where $N(1 \le N \le p)$ is the number of time steps for which you are specifying reference values and `p` is the prediction horizon. Each element must be a real, double-precision number. The `ref` dimension must not change from one control instant to the next.

When `N=1`, you cannot preview. To specify future reference values, choose `N` such that $1 < N \le p$ to enable previewing. Doing so usually improves performance via `feedforward` information. The first row specifies the n_y references for the first step in the prediction horizon (at the next control interval `k=1`), and so on for `N` steps. If `N<p`, the last row designates constant reference values to be used for the remaining `p-N` steps.

For example, suppose n_y=2 and p=6. At a given control instant, the signal connected to the controller's `ref` inport is

```
[2 5     k=1
 2 6     k=2
 2 7     k=3
 2 8]     k=4
```

The signal informs the controller that:

- Reference values for the first prediction horizon step (`k=1`) are 2 and 5.
- The first reference value remains at 2, but the second increases gradually.
- The second reference value becomes 8 at the beginning of the fourth step (`k=4`) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 5–6 of the prediction horizon.

Required Outports

Manipulated Variables

The `mv` outport provides a signal defining the $n_u \geq 1$ manipulated variables, which are to be implemented in the plant. The controller updates its `mv` outport by solving a quadratic program at each control instant. The elements are real, double-precision values.

Optional Inports

Measured disturbance

Add an inport (`md`) to which you can connect a measured disturbance signal.

Your measured disturbance signal (MD) must be size $N x n_{md}$, where $n_{md} (\geq 1)$ is the number of measured disturbances defined in your Model Predictive Controller and N $(1 \leq N \leq p+1)$ is the number of time steps for which the MD is known. Each element must be a real, double-precision number. The signal dimensions must not change from one control instant to the next.

If N=1 you cannot preview. At each control instant, the MD signal must contain the most recent measurements at the current time k=0 (as a row vector, length n_{md}). The controller assumes that the MDs remain constant at their current values for the entire prediction horizon.

If you are able to predict future MD values, choose N such that $1 < N \leq p+1$ to enable previewing. Doing so usually improves performance via feedforward. In this case, the first row must contain the n_{md} current values at k=0, and the remaining rows designate variations over the next N-1 control instants. If N<p+1, the last row designates constant MD values to be used for the remaining p+1-N steps of the prediction horizon.

For example suppose n_{md}=2 and p=6. At a given control instant, the signal connected to the controller's md inport is:

```
[2 5     k=0
 2 6     k=1
 2 7     k=2
 2 8]    k=3
```

This signal informs the controller that:

- The current MDs are 2 and 5 at k=0.
- The first MD remains at 2, but the second increases gradually.
- The second MD becomes 8 at the beginning of the step 3 (k=3) in the prediction horizon.
- Both values remain constant at 2 and 8 respectively for steps 4–6 of the prediction horizon.

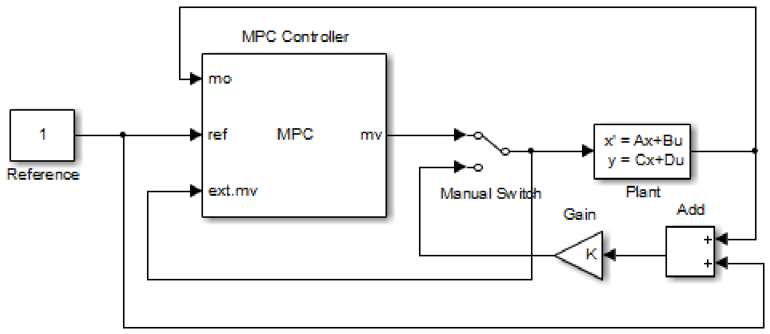
mpcpreview shows how to use MD previewing in a specific case.

For calculation details, see "Prediction Model" and "QP Matrices".

Externally supplied MV signals

Add an inport (`ext.mv`), which you can connect to the actual manipulated variables (MV) used in the plant. The block uses these to update its internal state estimates. For example, suppose the actual signals saturate at physical limits or the `MV` is under manual control. In both cases, feeding the actual value back to the MPC Controller block can improve performance significantly, because the prediction model's state estimates are updated more accurately.

The following example shows how a manual switch may override the controller's output. Also see Turning Controller Online and Offline with Bumpless Transfer.



Do not connect this option to leave the ext.mv inport unconnected. In either case, the model predictive controller assumes that the plant uses the MV signals sent by the MPC Controller block. In the preceding example, the external MV signal always provides the model predictive controller that the control signal actually used in the plant. Otherwise, the model predictive controller's internal state estimate would be inaccurate.

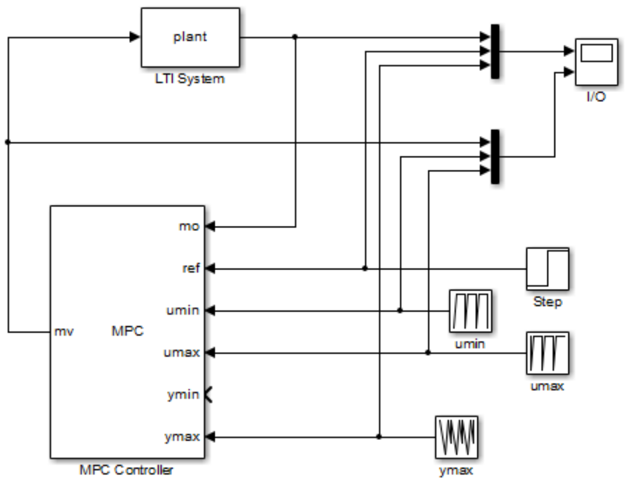
Note The MPC Controller block is a discrete-time block with sampling time inherited from the MPC object. The MPC block has direct feedthrough from measured outputs (`mo`), output references (`ref`), and measured disturbances (`md`) to MPC-manipulated variables (`mv`). There is no direct feedthrough from externally supplied manipulated variables (`ext.mv`) to MPC-manipulated variables (`mv`).

Input and output limits

Add inports (`umin`,`umax`,`ymin`,`ymax`), which you can connect to run-time constraint signals. If this check box is not selected, the block uses the constant constraint values stored within its `mpc` object. Example connections appear in the following model. See Varying Input and Output Constraints for an example of using this option.

Each unconnected limit inport, such as `ymin` in the following model, is treated as an unbounded signal. The corresponding constraint settings in the `mpc` object must also be unbounded. For connected limit inports, such as `ymax`, the signals must be finite and the corresponding variables in the `mpc` object must also be bounded.

All constraint signals connected to the block must be finite. Also, you cannot change the number or identity of constrained and unconstrained variables. For example, if your `mpc` object specifies that your first MV has a lower bound, you must supply a `umin` signal for it.



Signal Attributes and Block Sample Time

Output data type

Specify the data type of the manipulated variables (MV) as one of the following:

- `double` — Double-precision floating point (default).
- `single` — Single-precision floating point.

  You specify the output data type as `single` if you are implementing the model predictive controller on a single-precision target.

For an example of double- and single-precision simulation and code generation for an MPC controller, see “Simulation and Code Generation Using Simulink Coder”.

To view the port data types in a model, in the Simulink Editor, select **Display > Signals & PortsPort Data Types**. For more information, see “Display Port Data Types”.

Block uses inherited sample time (-1)

Use the sample time inherited from the parent subsystem as the Multiple MPC Controllers block's sample time.

Inheriting the sample time allows you to conditionally execute the Multiple MPC Controllers block inside the Function-Call Subsystem or Triggered Subsystem blocks. For an example, see Using MPC Controller Block Inside Function-Call and Triggered Subsystems.

Note When you place an MPC controller inside a Function-Call Subsystem or Triggered Subsystem block, you must execute the subsystem at the controller's design sample rate. You may see unexpected results if you use an alternate sample rate.

To view the sample time of a block, in the Simulink model window, select **Display > Sample Time**. Select **Colors**, **Annotations**, or **All**. For more information, see "View Sample Time Information".

See Also

`mpc` **|** `mpcmove` **|** `mpcstate` **|** MPC Controller

Related Examples

- Scheduling Controllers for a Plant with Multiple Operating Points
- Chemical Reactor with Multiple Operating Points
- "Simulation and Code Generation Using Simulink Coder"
- "Simulation and Structured Text Generation Using PLC Coder"

Object Reference

MPC Controller Object

All the parameters defining the MPC control law (prediction horizon, weights, constraints, etc.) are stored in an MPC object, whose properties are listed in the following table (MPC Controller Object on page 3-2).

MPC Controller Object

Property	Description
`ManipulatedVariables` (or `MV` or `Manipulated` or `Input`)	Input and input-rate upper and lower bounds, ECR values, names, units, and input target
`OutputVariables` (or `OV` or `Controlled` or `Output`)	Output upper and lower bounds, ECR values, names, units
`DisturbanceVariables` (or `DV` or `Disturbance`)	Disturbance names and units
`Weights`	Weights defining the performance function
`Model`	Plant, input disturbance, and output noise models, and nominal conditions.
`Ts`	Controller's sampling time
`Optimizer`	Parameters for 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

MPC Controller Object (Continued)

Property	Description
`MPCData` (private)	Matrices for the QP problem and other accessorial data
`Version` (private)	Model Predictive Control Toolbox version number

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 (Structure ManipulatedVariables on page 3-3), where p denotes the prediction horizon.

Structure ManipulatedVariables

Field Name	Content	Default
`Min`	1 to p dimensional vector of lower constraints on a manipulated variable u	`-Inf`
`Max`	1 to p dimensional vector of upper constraints on a manipulated variable u	`Inf`
`MinECR`	1 to p dimensional vector describing the equal concern for the relaxation of the lower constraints on u	`0`
`MaxECR`	1 to p dimensional vector describing the equal concern for the relaxation of the upper constraints on u	`0`
`Target`	1 to p dimensional vector of target values for the input variable u	`'nominal'`

Structure ManipulatedVariables (Continued)

Field Name	Content	Default
`RateMin`	1 to p dimensional vector of lower constraints on the rate of a manipulated variable u	`-Inf` if problem is unconstrained, otherwise `-10`
`RateMax`	1 to p dimensional vector of upper constraints on the rate of a manipulated variable u	`Inf`
`RateMinECR`	1 to p dimensional vector describing the equal concern for the relaxation of the lower constraints on the rate of u	`0`
`RateMaxECR`	1 to p dimensional vector describing the equal concern for the relaxation of the upper constraints on the rate of u	`0`
`Name`	Name of input signal. This is inherited from `InputName` of the LTI plant model.	`InputName` of LTI plant model
`Units`	String specifying the measurement units for the manipulated variable	`''`

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 (Structure OutputVariables on page 3-5), where p denotes the prediction horizon.

Structure OutputVariables

Field Name	Content	Default
`Min`	1 to p dimensional vector of lower constraints on an output y	`-Inf`
`Max`	1 to p dimensional vector of upper constraints on an output y	`Inf`
`MinECR`	1 to p dimensional vector describing the equal concern for the relaxation of the lower constraints on an output y	`1`
`MaxECR`	1 to p dimensional vector describing the equal concern for the relaxation of the upper constraints on an output y	`1`
`Name`	Name of output signal. This is inherited from `OutputName` of the LTI plant model.	`OutputName` of LTI plant model
`Units`	String specifying the measurement units for the measured output	`''`
`Integrator`	Magnitude of integrated white noise on the output channel (0=no integrator)	`[]`

In order to reject constant disturbances due for instance to gain nonlinearities, the default output disturbance model used in Model Predictive Control Toolbox software is a collection of integrators driven by white noise on measured outputs (see "Output Disturbance Model"in the Model Predictive Control Toolbox User's Guide). Output integrators are added according to the following rule:

1 Measured outputs are ordered by decreasing output weight (in case of time-varying weights, the sum of the absolute values over time is considered for each output channel, and in case of equal output weight, the order within the output vector is followed).

2 By following such order, an output integrator is added per measured outputs, unless there is a violation of observability, or you force it by zeroing the corresponding value in `OutputVariables.Integrators`).

By default, `OutputVariables.Integrators` is empty on all outputs. This enforces the default action of Model Predictive Control Toolbox software, namely add integrators on measured outputs, do not add integrators on unmeasured outputs. By setting the entry of `OutputVariables(i).Integrators` to zero, no attempt will be made to add integrated white noise on the `i`-th output . On the contrary, by setting the entry of `OutputVariables(i).Integrators` to one, an attempt will be made to add integrated white noise on the `i`-th output (see `getoutdist`).

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), one per input disturbance. Each structure has the fields described in the following table (Structure DisturbanceVariables on page 3-6).

Structure DisturbanceVariables

Field Name	Content	Default
`Name`	Name of input signal. This is inherited from `InputName` of the LTI plant model.	`InputName` of LTI plant model
`Units`	String specifying the measurement units for the manipulated variable	`''`

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

Note The Name properties of ManipulatedVariables, OutputVariables, and DisturbanceVariables are read only. You can set signal names in the Model.Plant.InputName and Model.Plant.OutputNameproperties of the MPC object, for instance by using the method setname.

Weights

Weights is the structure defining the QP weighting matrices. Unlike the InputSpecs and OutputSpecs, which are arrays of structures, weights is a single structure containing four fields. The values of these fields depend on whether you are using the standard quadratic cost function (see "Standard Form") or the alternative cost function (see "Alternative Cost Function").

Standard Cost Function

The table below, Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7, lists the content of the four fields where p denotes the prediction horizon, n_u the number of manipulated variables, n_y the number of output variables.

The fields ManipulatedVariables, ManipulatedVariablesRate, and OutputVariables are arrays with n_u, n_u, and n_y columns, respectively. If weights are time invariant, then ManipulatedVariables, ManipulatedVariablesRate, and OutputVariables are row vectors. However, for time-varying weights, each field is a matrix with up to p rows. If the number of rows is less than the prediction horizon, p, the object constructor duplicates the last row to create a matrix with p rows.

Weights for the Standard Cost Function (MATLAB Structure)

Field Name	Content	Default
ManipulatedVariables (or MV or Manipulated or Input)	(1 to p)-by-n_u dimensional array of input weights	zeros(1,nu)
ManipulatedVariablesRate (or MVRate or ManipulatedRate or InputRate)	(1 to p)-by-n_u dimensional array of input-rate weights	0.1*ones(1,nu)

Weights for the Standard Cost Function (MATLAB Structure) (Continued)

Field Name	Content	Default
`OutputVariables` (or `OV` or `Controlled` or `Output`)	(1 to p)-by-n_y dimensional array of output 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 are weighted with unit weight (with preference given to measured outputs), while the remaining outputs receive zero weight.)
`ECR`	Weight on the slack variable ε used for softening the constraints	1e5*(max weight)

The default ECR weight is 10^5 times the largest weight specified in `ManipulatedVariables`, `ManipulatedVariablesRate`, and `OutputVariables`.

Note All weights must be greater than or equal to zero. If all weights on manipulated variable increments are strictly positive, the resulting QP problem is always strictly convex. If some of those weights are zero, the Hessian matrix of the QP problem may become only positive semidefinite. In order to keep the QP problem always strictly convex, if the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12}, the quantity `10*sqrt(eps)` is added on each diagonal term. This may only occur when all input rates are not weighted ($W^{\Delta u}$=0) (see "Cost Function" in the *Model Predictive Control Toolbox User's Guide*).

Alternative Cost Function

You can specify off-diagonal Q and R weight matrices in the cost function. To accomplish this, you must 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 abbreviate the field names as shown in Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7. You can also use diagonal weights (as defined in Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7) for one or more of these fields. If you omit a field, the object constructor uses the defaults shown in Weights for the Standard Cost Function (MATLAB® Structure) on page 3-7.

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

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

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

Note You cannot specify off-diagonal time-varying weights.

Model

The property `Model` specifies plant, input disturbance, and output noise models, and nominal conditions, according to the model setup described in "State Estimation". It is specified through a structure containing the fields reported in Structure Model Describing the Models Used by MPC on page 3-9.

Structure Model Describing the Models Used by MPC

Field Name	Content	Default
`Plant`	LTI model or identified linear model of the plant	No default
`Disturbance`	LTI model describing color of input disturbances	An integrator on each `Unmeasured` input channel

Structure Model Describing the Models Used by MPC (Continued)

Field Name	Content	Default
`Noise`	LTI model describing color of plant output measurement noise	Unit white noise on each measured output = identity static gain
`Nominal`	Structure containing the state, input, and output values where `Model.Plant` is linearized	See Nominal Values at Operating Point on page 3-11.

Note Direct feedthrough from manipulated variables to any output in `Model.Plant` is not allowed. See “Prediction Model” in the Model Predictive Control Toolbox User’s Guide.

The type of input and output signals is assigned either through the `InputGroup` and `OutputGroup` properties of `Model.Plant`, or, more conveniently, through function `setmpcsignals`, according to the nomenclature described in Input Groups in Plant Model on page 3-10 and Output Groups in Plant Model on page 3-11.

Input Groups in Plant Model

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

Output Groups in Plant Model

Name	Value
`MeasuredOutputs` (or `MO` or `Measured`)	Indices of measured outputs
`UnmeasuredOutputs` (or `UO` or `Unmeasured`)	Indices of unmeasured outputs

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

Note With this current release, the `InputGroup` and `OutputGroup` properties of LTI objects are defined as structures, rather than cell arrays (see the Control System Toolbox documentation for more details).

The structure `Nominal` contains the nominal values for states, inputs, outputs and state derivatives/differences at the operating point where `Model.Plant` was linearized. The fields are reported in Nominal Values at Operating Point on page 3-11 (see "Offsets" in the Model Predictive Control Toolbox User's Guide).

Nominal Values at Operating Point

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

Ts

Sampling 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 sampling time for the MPC controller.

Optimizer

Parameters for the QP optimization. Optimizer is a structure with the fields reported in the following table (Optimizer Properties on page 3-12).

Optimizer Properties

Field	Description	Default
`MaxIter`	Maximum number of iterations allowed in the QP solver	`200`
`Trace`	On/off	`'off'`
`Solver`	QP solver used (only `'ActiveSet'`)	`'ActiveSet'`
`MinOutputECR`	Minimum positive value allowed for `OutputMinECR` and `OutputMaxECR`	`1e-10`

`MinOutputECR` is a positive scalar used to specify the minimum allowed ECR for output constraints. If values smaller than `MinOutputECR` are provided in the `OutputVariables` property of the MPC objects a warning message is issued and the value is raised to `MinOutputECR`.

PredictionHorizon

`PredictionHorizon` is an integer value expressing the number *p* of sampling steps of prediction.

ControlHorizon

`ControlHorizon` is either a number of free control moves, or a vector of blocking moves (see "Optimization Variables" in the Model Predictive Control Toolbox User's Guide).

History

`History` stores the time the MPC controller was created.

Notes

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

UserData

Any additional data stored within the MPC controller object.

MPCData

`MPCData` is a private property of the MPC object used for storing intermediate operations, QP matrices, internal flags, etc.

Version

`Version` is a private property indicating the Model Predictive Control Toolbox version number.

Construction and Initialization

An MPC object is built in two steps. The first step happens *at construction* of the object when the object constructor `mpc` is invoked, or properties are changed by a `set` command. At this first stage, only basic consistency checks are performed, such as dimensions of signals, weights, constraints, etc. The second step happens *at initialization* of the object, namely when the object is used for the first time by methods such as `mpcmove` and `sim`, that require the full computation of the QP matrices and the estimator gain. At this second stage, further checks are performed, such as a test of observability of the overall extended model.

Informative messages are displayed in the command window in both phases, you can turn them 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 on page 3-14).

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.
`UnmeasuredDisturbance`	Unmeasured disturbance signal entering the plant.
`InputNoise`	Noise on manipulated variables.
`OutputNoise`	Noise on measured outputs.
`RefLookAhead`	Preview on reference signal (`'on'` or `'off'`).
`MDLookAhead`	Preview on measured disturbance signal (`'on'` or `'off'`).
`Constraints`	Use MPC constraints (`'on'` or `'off'`).
`Model`	Model used in simulation for generating the data.
`StatusBar`	Display the wait bar (`'on'` or `'off'`).
`MVSignal`	Sequence of manipulated variables (with offsets) for open-loop simulation (no MPC action).
`OpenLoop`	Perform open-loop simulation.

The command

```
SimOptions=mpcsimopt(mpcobj)
```

returns an empty `@mpcsimopt` object. You must use `set` / `get` to change simulation options.

`UnmeasuredDisturbance` is an array with as many columns as unmeasured disturbances, `InputNoise` and `MVSignal` are arrays with as many columns as manipulated variables, `OutputNoise` is an array with as many columns as measured outputs. The last sample of the array is extended constantly over the horizon to obtain the correct size.

Note Nonzero values of `ControllerInitialState.LastMove` are only meaningful if there are constraints on the increments of the manipulated variables.

The property `Model` 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`, `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.

MPC State Object

The `mpcstate` object type contains the state of an MPC controller. Its properties are listed in MPC State Object Properties on page 3-16.

MPC State Object Properties

Property	Description
`Plant`	Array of plant states. Values are absolute, i.e., they include possible state offsets (cf.`Model.Nominal.X`).
`Disturbance`	Array of states of unmeasured disturbance models. This contains the states of the input disturbance model and, appended below, the states of the unmeasured output disturbances model.
`Noise`	Array of states of measurement noise model.
`LastInput`	Array of previous manipulated variables $u(k\text{-}1)$. Values are absolute, i.e., they include possible input offsets (cf. `Model.Nominal.U`).

The command

```
mpcstate(mpcobj)
```

returns a zero extended initial state compatible with the MPC object `mpcobj`, and with `mpcobj.Plant` and `mpcobj.LastInput` initialized at the nominal values specified in `mpcobj.Model.Nominal`.