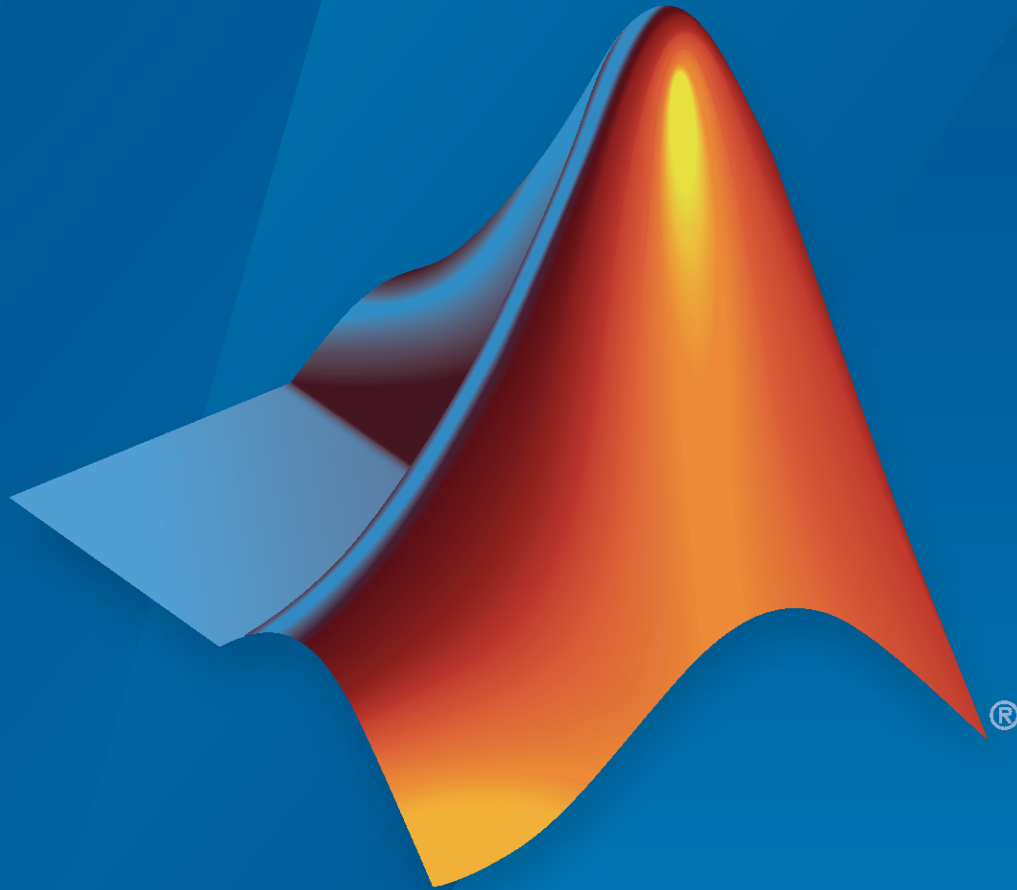# Model Predictive Control Toolbox™

Reference

Alberto Bemporad
N. Lawrence Ricker
Manfred Morari

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Contents

Apps

MPC Designer

Design and simulate model predictive controllers

Description

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

Using this app, you can:

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

Limitations

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

- Explicit MPC design
- Adaptive MPC design
- Nonlinear MPC design
- Mixed input/output constraints (`setconstraint`)
- Terminal weight specification (`setterminal`)
- Custom state estimation (`setEstimator`)
- Sensitivity analysis (`sensitivity`)
- Alternative cost functions with off-diagonal weights
- Specification of initial plant and controller states for simulation
- Specification of nominal state values using `mpcobj.Model.Nominal.X` and `mpcobj.Model.Nominal.DX`
- Updating weights, constraints, MV targets, and external MVs online during simulations

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

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

- Finding an operating point for linearizing a Simulink model using trimming or simulation snapshots. Instead, you must linearize your model at the model initial conditions.
- Generating Simulink models for your controller and plant.
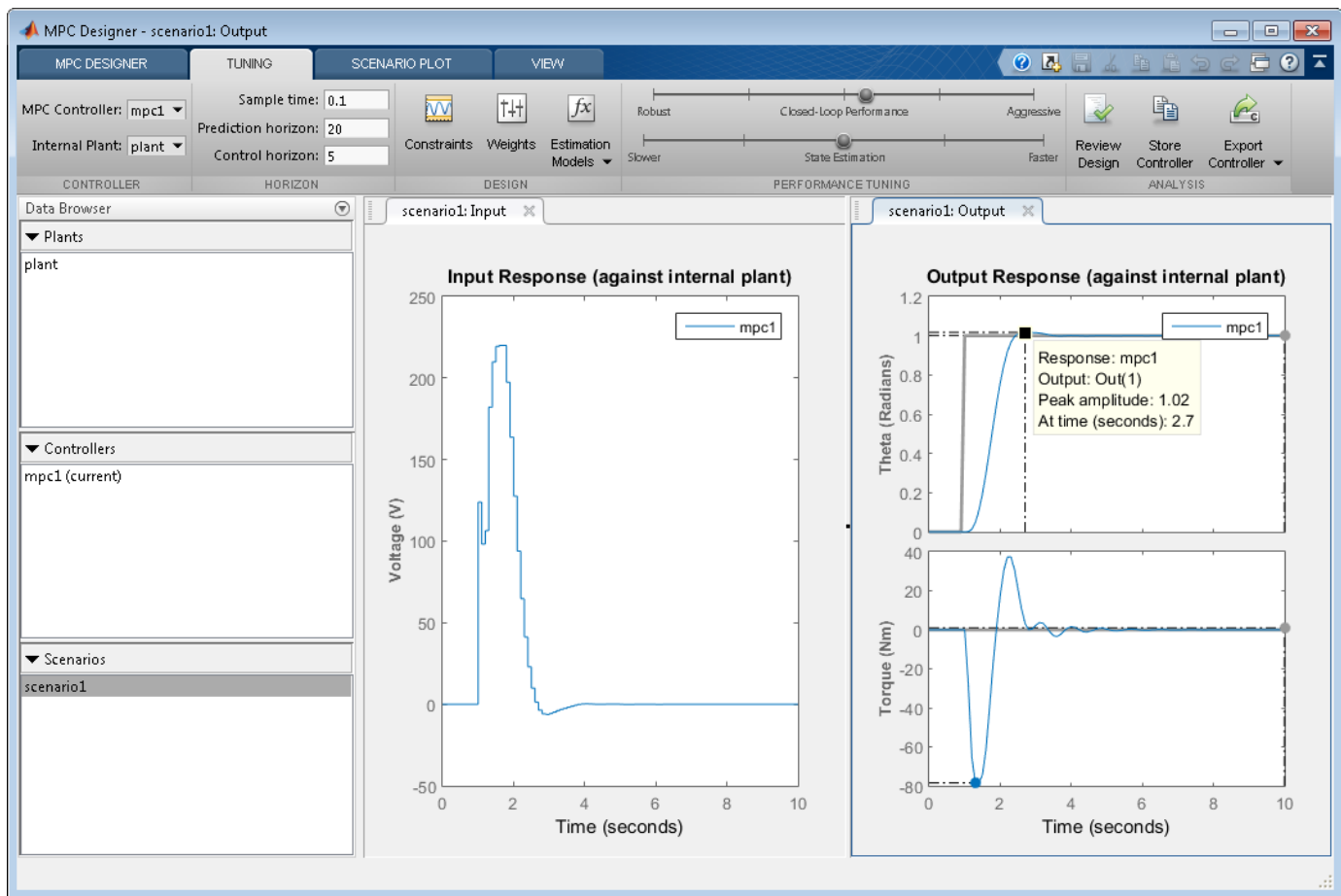
Open the MPC Designer App

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

Examples

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

Programmatic Use

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

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

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

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

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

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

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

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

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

* The name of a session data file in the current working directory or on the MATLAB path, specified as a character vector or string. To save session data to disk, in the **MPC Designer** app, on the

  MPC Designer tab, click ▣ **Save Session**. The saved session data includes all plants, controllers, and scenarios in the **Data Browser**, the current MPC structure, and the current plot configuration.

* A previously loaded `SessionData` object in the MATLAB workspace. To load a `SessionData` object from a session data file, at the command line, enter:

  ```
  load sessionFile
  ```

Version History
Introduced in R2015b

R2021b: Support for opening MPC Design Tool sessions saved before release R2015b has been removed
Errors starting in R2021b

Support for opening MPC Design Tool sessions saved before release R2015b has been removed in release R2021b.

If you have sessions saved before release R2015b, open and resave the session files using **MPC Designer** in any release from R2015b through R2021a.

See Also

Functions
sim

Objects
mpc

Topics
"Design Controller Using MPC Designer"
"Design MPC Controller in Simulink"
"Compare Multiple Controller Responses Using MPC Designer"
"Generate MATLAB Code from MPC Designer"
"Generate Simulink Model from MPC Designer"
"What is Model Predictive Control?"

Functions

buildMEX

Build MEX file that solves an MPC control problem

Syntax

```
mexLinFcn = buildMEX(mpcobj,mexName,coreData,stateData,onlineData)
```

```
mexNlnFcn = buildMEX(nlobj,mexName,coreData,onlineData)
mexFcn = buildMEX(nlobj,mexName,coreData,onlineData,mexConfig)
```

Description

Linear MPC

`mexLinFcn = buildMEX(mpcobj,mexName,coreData,stateData,onlineData)` builds a MEX file that solves a linear MPC control problem faster than `mpcmove`. The MEX file is created in the current working folder.

Nonlinear MPC

`mexNlnFcn = buildMEX(nlobj,mexName,coreData,onlineData)` builds a MEX file that solves a nonlinear MPC control problem faster than `nlmpcmove`. The MEX file is created in the current working folder.

`mexFcn = buildMEX(nlobj,mexName,coreData,onlineData,mexConfig)` generates a MEX function using the code generation configuration object `mexConfig`. Use this syntax to customize your MEX code generation.

Examples

Simulate Linear MPC Controller Using MEX File

Create a plant model and design an MPC controller for the plant, with sample time `0.1`.

```
plant = drss(1,1,1);plant.D = 0;
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Simulate the plant using `mpcmove` for 5 steps.

```
x = 0;
xc = mpcstate(mpcobj);
```

```
-->No sample time in plant model. Assuming controller's sample time of 0.1.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
for i=1:5
    % Update plant output.
    y = plant.C*x;
    % Compute control actions.
    u = mpcmove(mpcobj,xc,y,1);
    % Update plant state.
    x = plant.A*x + plant.B*u;
end
```

Display the final value of the plant output, input and next state.

```
[y u x]
```

ans = *1×3*

```
    0.9999    0.3057   -2.3063
```

Generate data structures and build mex file.

```
[coreData,stateData,onlineData] = getCodeGenerationData(mpcobj);
mexfun = buildMEX(mpcobj, 'myMPCMex', coreData, stateData, onlineData);
```

```
Generating MEX function "myMPCMex" from linear MPC to speed up simulation.
Code generation successful.
```

```
MEX function "myMPCMex" successfully generated.
```

Simulate the plant using the mex file myMPCmex, which you just generated, for 5 steps.

```
x=0;
for i = 1:5
    % Update plant output.
    y = plant.C*x;
    % Update measured output in online data.
    onlineData.signals.ym = y;
    % Update reference signal in online data.
    onlineData.signals.ref = 1;
    % Compute control actions.
    [u,stateData] = myMPCMex(stateData,onlineData);
    % Update plant state.
    x = plant.A*x + plant.B*u;
end
```

Display the final value of the plant output, input and next state.

```
[y u x]
```

ans = *1×3*

```
    0.9999    0.3057   -2.3063
```

Simulate Nonlinear MPC Controller Using MEX File

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

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

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

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

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

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

The prediction model uses an optional parameter `Ts` to represent the sample time. Specify the number of parameters and create a parameter vector.

```
nlobj.Model.NumberOfParameters = 1;
params = {Ts};
```

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

```
nlobj.Model.OutputFcn = "pendulumOutputFcn";
```

Define standard constraints for the controller.

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

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj,x0,u0,[],params);
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Kalman filter for estimating the four plant states. Its state transition function is defined in `pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero initial manipulated variable value.

```
x0 = [0;0;-pi;0];
y0 = [x0(1);x0(3)];
```

```
EKF.State = x0;
mv0 = 0;
```

Create code generation data structures for the controller, specifying the initial conditions and parameters.

```
[coreData,onlineData] = getCodeGenerationData(nlobj,x0,mv0,params);
```

Specify the output reference value in the online data structure.

```
onlineData.ref = [0 0];
```

Build a MEX function for solving the nonlinear MPC control problem. The MEX function is created in the current working directory.

```
mexFcn = buildMEX(nlobj,"myController",coreData,onlineData);
```

```
Generating MEX function "myController" from nonlinear MPC to speed up simulation.
Code generation successful.

MEX function "myController" successfully generated.
```

Run the simulation for 10 seconds. During each control interval:

1 Correct the previous prediction using the current measurement.
2 Compute optimal control moves using the MEX function. This function returns the computed optimal sequences in `onlineData`. Passing the updated data structure to the MEX function in the next control interval provides initial guesses for the optimal sequences.
3 Predict the model states.
4 Apply the first computed optimal control move to the plant, updating the plant states.
5 Generate sensor data with white noise.
6 Save the plant states.

```
mv = mv0;
y = y0;
x = x0;
Duration = 10;
xHistory = x0;
for ct = 1:(Duration/Ts)
    % Correct previous prediction
    xk = correct(EKF,y);
    % Compute optimal control move
    [mv,onlineData] = myController(xk,mv,onlineData);
    % Predict prediction model states for the next iteration
    predict(EKF,[mv; Ts]);
    % Implement first optimal control move
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data
    y = x([1 3]) + randn(2,1)*0.01;
    % Save plant states
    xHistory = [xHistory x];
end
```

Plot the resulting state trajectories.

```
figure
subplot(2,2,1)
```

```
plot(0:Ts:Duration,xHistory(1,:))
xlabel('time')
ylabel('z')
title('cart position')
subplot(2,2,2)
plot(0:Ts:Duration,xHistory(2,:))
xlabel('time')
ylabel('zdot')
title('cart velocity')
subplot(2,2,3)
plot(0:Ts:Duration,xHistory(3,:))
xlabel('time')
ylabel('theta')
title('pendulum angle')
subplot(2,2,4)
plot(0:Ts:Duration,xHistory(4,:))
xlabel('time')
ylabel('thetadot')
title('pendulum velocity')
```

Input Arguments

`mpcobj` — Model predictive controller
mpc object | explicitMPC object

Model predictive controller, specified as one of the following:

- `mpc` object — Implicit MPC controller
- `explicitMPC` object — Explicit MPC controller created using `generateExplicitMPC`.

nlobj — Nonlinear model predictive controller
`nlmpc` object | `nlmpcMultistage` object

Nonlinear model predictive controller, specified as an `nlmpc` or `nlmpcMultistage` object.

Your controller must use the default `fmincon` solver with the SQP algorithm. Also, your controller must not use anonymous functions for its prediction model, custom cost function, or custom constraint functions.

mexName — MEX function name
string | character vector

MEX function name, specified as a string or character vector.

coreData — Controller configuration parameters
structure

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

stateData — Linear controller state data structure
structure

Linear controller state data structure at run time, specified as a structure generated using `getCodeGenerationData`.

onlineData — Initial online controller data
structure

Initial online controller data, specified as a structure generated using `getCodeGenerationData`. For more information on setting the fields of `onlineData`, see `nlmpcmoveCodeGeneration`.

mexConfig — Code generation configuration object
`MexCodeConfig` object

Code generation configuration object, specified as a `MexCodeConfig` object.

To create the configuration object, use the following code.

```
mexConfig = coder.config('mex');
```

To customize your MEX code generation, modify the settings of this object. For example, to detect run-time memory access violations during debugging, set `IntegrityChecks` to `true`.

```
mexConfig.IntegrityChecks = true;
```

By default, to improve the performance of the generated code, checks such as `IntegrityChecks` and `ResponsivenessChecks` are disabled by `buildMEX`.

`buildMEX` overwrites the following configuration settings with the values indicated.

Configuration Setting	Value
cfg.DynamicMemoryAllocation	'AllVariableSizeArrays'
cfg.ConstantInputs	'Remove'

Output Arguments

mexLinFcn — Generated MEX function for linear MPC
function handle

Generated MEX function for linear MPC, returned as a function handle. This MEX function has the following signature.

```
[mv,newStateData,info] = mexLinFcn(stateData,onlineData)
```

The MEX function has the following input arguments, which are the same as the corresponding input arguments of mpcmoveCodeGeneration, except configData, which is embedded in mexLinFcn .

Input Argument	Description
stateData	Current state data. Structure containing the state of the linear MPC controller. For more information on setting the fields of onlineData, see mpcmoveCodeGeneration
onlineData	Online controller data that you must update at run time, specified as a structure. Generate the initial structure using getCodeGenerationData. For more information on setting the fields of onlineData, see mpcmoveCodeGeneration.

The MEX function has the following output arguments, which are the same as the output arguments of mpcmoveCodeGeneration.

Output Argument	Description
mv	Optimal manipulated variable control action, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.
newStateData	Updated state data. Structure containing the updated state of the linear MPC controller. For more information on setting the fields of onlineData, see mpcmoveCodeGeneration
info	Solution details, returned as a structure.

To simulate a controller using the generated MEX function, use the initial online data structure onlineData for the first control interval. For subsequent control intervals, modify the online data in newOnlineData and pass the updated structure to the MEX function as onlineData.

mexNlnFcn — Generated MEX function for nonlinear MPC
function handle

Generated MEX function for nonlinear MPC, returned as a function handle. This MEX function has the following signature.

```
[mv,newOnlineData,info] = mexNlnFcn(x,lastMV,onlineData)
```

The MEX function has the following input arguments, which are the same as the corresponding input arguments of `nlmpcmoveCodeGeneration`.

Input Argument	Description
x	Current prediction model states, specified as a vector of length N_x, where N_x is the number of prediction model states.
lastMV	Control signals used in plant at previous control interval, specified as a vector of length N_{mv}, where N_{mv} is the number of manipulated variables.
onlineData	Online controller data that you must update at run time, specified as a structure. Generate the initial structure using `getCodeGenerationData`. For more information on setting the fields of `onlineData`, see `nlmpcmoveCodeGeneration`.

The MEX function has the following output arguments, which are the same as the output arguments of `nlmpcmoveCodeGeneration`.

Output Argument	Description
mv	Optimal manipulated variable control action, returned as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.
newOnlineData	Updated online controller data, returned as a structure. This structure is the same as `onlineData`, except that the decision variable initial guesses are updated.
info	Solution details, returned as a structure.

To simulate a controller using the generated MEX function, use the initial online data structure `onlineData` for the first control interval. For subsequent control intervals, modify the online data in `newOnlineData` and pass the updated structure to the MEX function as `onlineData`.

Version History
Introduced in R2020a

See Also

Functions
mpcmove | nlmpcmove | mpcmoveCodeGeneration | nlmpcmoveCodeGeneration | getCodeGenerationData | codegen

Objects
mpc | nlmpc

Topics
"Parallel Parking Using Nonlinear Model Predictive Control"
"Generate Code and Deploy Controller to Real-Time Targets"

cloffset

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

Syntax

```
dcgain = cloffset(mpcobj)
```

Description

Use this function to calculate the steady state output sensitivity of the closed loop. A zero value means that the measured plant output can track the desired output reference setpoint.

`dcgain = cloffset(mpcobj)` returns the DC gain matrix `dcgain`. `mpcobj` is the MPC object specifying the controller for which the closed-loop gain is calculated.
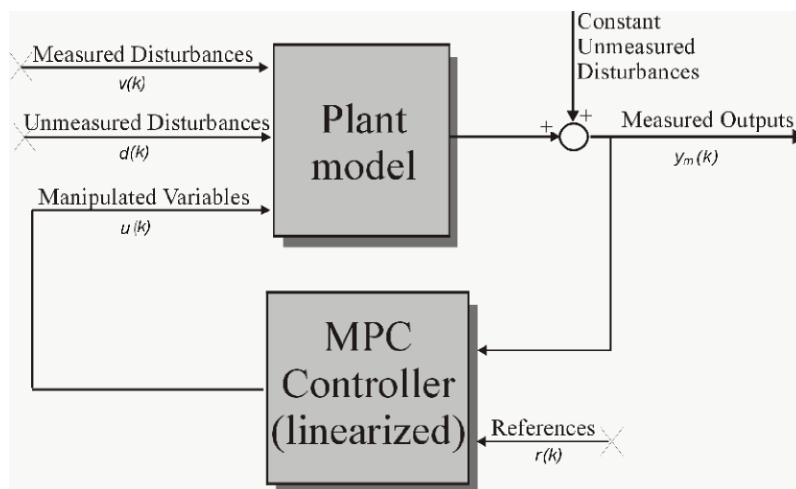
Computing the Effect of Output Disturbances

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

Examples

Calculate steady state output sensitivity of MPC in closed loop

Create a plant, a corresponding MPC object, and calculate the closed loop static gain (this is also referred to as steady state loop output sensitivity).

```
mpcverbosity off;           % turn off mpc messaging
plant=tf(1,[1 1],0.2);      % create plant (0.2 seconds sampling time)
mpcobj=mpc(plant,0.2);      % create mpc object (0.2 second sampling time)
```

```
cloffset(mpcobj)            % calculate steady state output sensitivity of closed loop

ans =

    0

A zero gain (which is typically the result of the controller having
% integrators as input or output disturbance models) means that the
% measured plant output will track the desired output reference setpoint.

zpk(mpcobj)                 % convert unconstrained MPC to zero/pole/gain form

ans =

  From input "MO1" to output "MV1":
      0.45205 z^2 (z-1.5)
  ----------------------------
  (z-1) (z-0.02575) (z+0.02485)

Sample time: 0.2 seconds
Discrete-time zero/pole/gain model.
```

Converting the unconstrained controller to zpk form shows that the pole in z=1, (resulting from the default noise model being an integrator), causes the controller static gain to approach infinity, in turn causing the closes loop output sensitivity to be zero at steady state (z=1). This allows the controller to successfully track the output reference signal.

Input Arguments

mpcobj — Model predictive controller
mpc object

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

Output Arguments

dcgain — Steady state closed loop output sensitivity
matrix

The steady state closed loop output sensitivity dcgain is an n_{ym}-by-n_{ym} matrix , where n_{ym} is the number of measured plant outputs.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, and that the controller can achieve perfect tracking of the ith component of an output reference setpoint (assuming constraints are inactive at steady state).

Version History
Introduced before R2006a

See Also

Functions
trim | review | sensitivity | mpcprops | ss | tf | zpk | d2d | sim

Objects
mpc | mpcsimopt

Topics
"Compute Steady-State Output Sensitivity Gain"
"Extract Controller"
"Simulating MPC Controller with Plant Model Mismatch"
"MPC Prediction Models"

compare

Compare two MPC objects

Syntax

```
yesno = compare(mpcobj1,mpcobj2)
```

Description

`yesno = compare(mpcobj1,mpcobj2)` compares the contents of the two MPC objects `mpcobj1` and, `mpcobj2` given as input arguments. If the design specifications (models, weights, horizons, etc.) are identical, then the returned value `yesno` is equal to 1.

Note `compare` may return `yesno = 1` even if the two objects are not identical. For instance, `mpcobj1` may have been initialized while `mpcobj2` 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.

Examples

Compare two MPC objects

Create two MPC controllers with different control horizons and compare them.

```
plant=zpk([],2,1);              % create plant
mpcverbosity off;               % turn off MPC messaging
mpcobj1=mpc(plant,0.1,10,2);    % create an mpc controller with a control horizon of 2 steps
mpcobj2=mpc(plant,0.1,10,3);    % create an mpc controller with a control horizon of 3 steps

compare(mpcobj1,mpcobj2)        % compare the controllers

ans =
  logical
   0
```

Input Arguments

mpcobj1 — MPC controller object
mpc object

First MPC object to compare

Example: `mpc(tf(1,[1 0]),1,12,3)`

mpcobj2 — MPC controller object
mpc object

Second MPC object to compare

Example: `mpc(tf(1,[1 0]),1,12,4)`

Output Arguments

yesno — Comparison result
`0 | 1`

The returned value is a logical `1` (that is `true`) if the design specifications (models, weights, horizons, etc.) are identical.

Version History

Introduced before R2006a

See Also

Functions
`review`

Objects
`mpc`

convertToMPC

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

Syntax

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

Description

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

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

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

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

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

Examples

Create Linear MPC Controllers from Nonlinear MPC Controller

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

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

Specify the controller sample time and horizons.

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

Specify the state function of the prediction model.

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

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

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

Specify the manipulated variable constraints and scale factor.

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

Specify the measured disturbance scale factor.

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

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

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

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

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

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

Create linear MPC controllers for each of these nominal conditions.

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

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

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

View the properties of the `mpcobjLow` controller.

```
mpcobjLow
```

```
MPC object (created on 03-Mar-2023 21:07:32):
---------------------------------------------
Sampling time:      1 (seconds)
Prediction Horizon: 10
Control Horizon:    3

Plant Model:
                                  -------------
      1  manipulated variable(s)   -->|  4 states   |
                                     |             |-->  1 measured output(s)
      1  measured disturbance(s)    -->|  2 inputs   |
```

```
                                 |                   |-->  0 unmeasured output(s)
        0  unmeasured disturbance(s) -->|  1 outputs |
                                        -------------
Indices:
   (input vector)     Manipulated variables: [1 ]
                     Measured disturbances: [2 ]
   (output vector)        Measured outputs: [1 ]

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

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

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

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

Use built-in "active-set" QP solver with MaxIterations of 120.
```

Input Arguments

`nlmpcobj` — Nonlinear MPC controller
`nlmpc` object

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

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

`states` — Nominal state values
array

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

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

`inputs` — Nominal input values
array

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

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

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

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

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

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

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

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

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

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

Output Arguments

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

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

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

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

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

If nlmpcobj:

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

Any state bounds in nlmpcobj are dropped during conversion.

Version History
Introduced in R2018b

See Also

Functions
review | ss | tf | compare

Objects
mpc | nlmpc

Topics
"Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant"
"Extract Controller"
"Nonlinear MPC"

createParameterBus

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

Syntax

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

Description

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

Examples

Create Parameter Bus for Nonlinear MPC Controller Block

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

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

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

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

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

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

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

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

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

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

Define standard constraints for the controller.

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

Open Simulink model.

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


Copyright 1990-2020 The MathWorks, Inc.
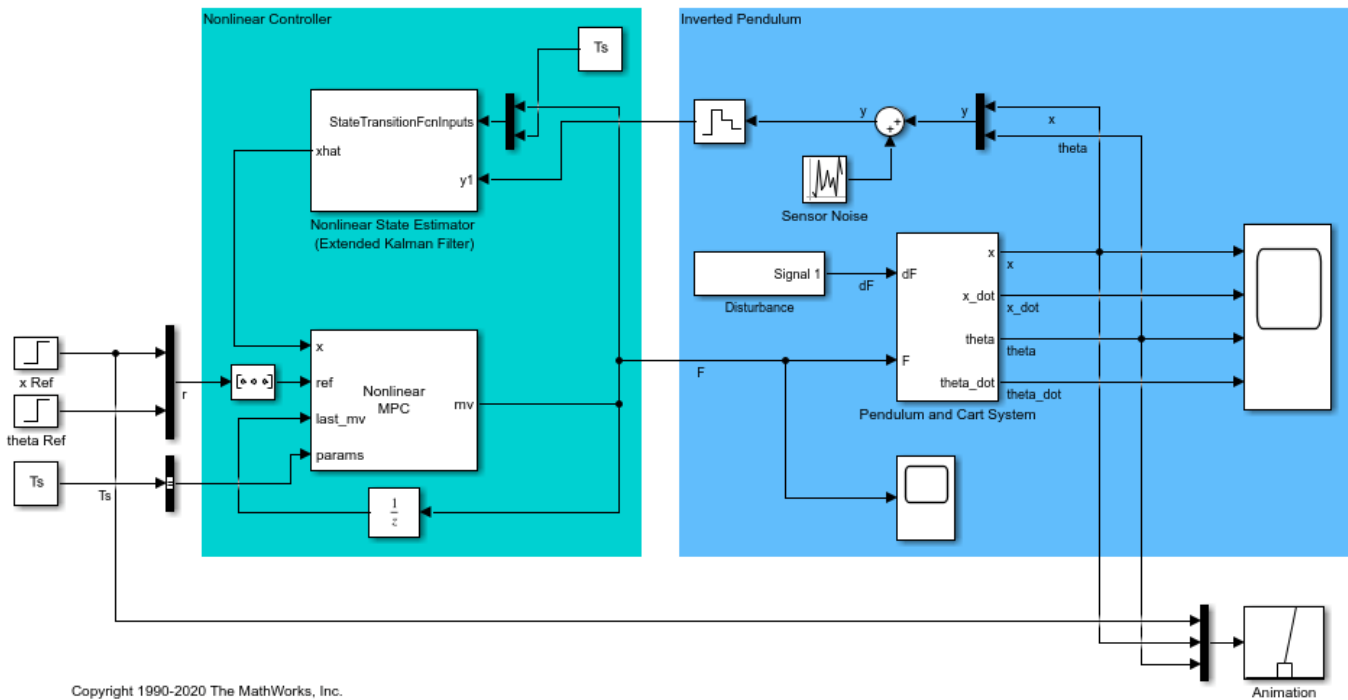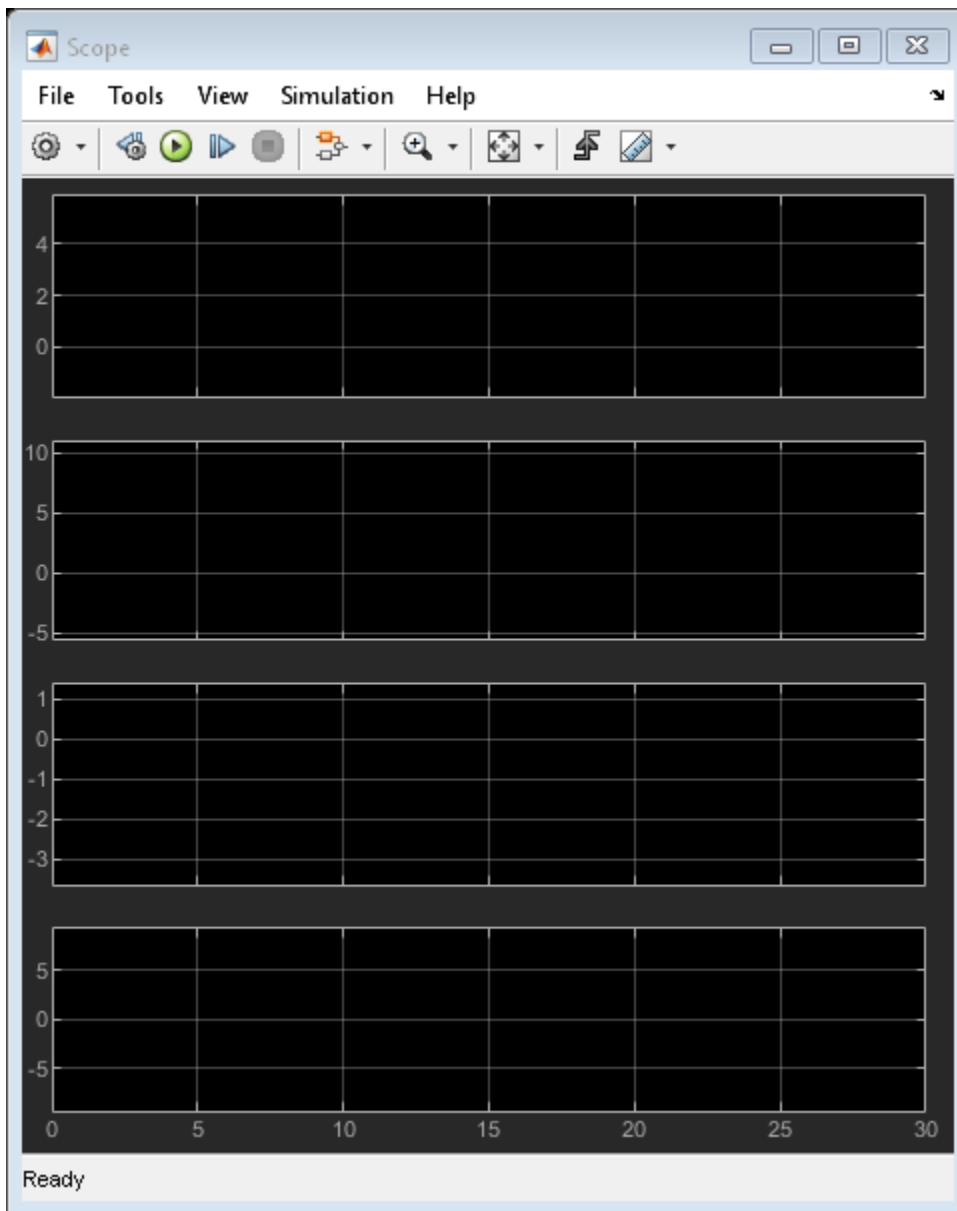
In this model, the Nonlinear MPC Controller block is configured to use the controller `nlobj`.

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

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

```
bdclose(mdl)
```

```
Simulink Bus object "myBusObject" created in the MATLAB Workspace.
Bus Creator block "mpc_pendcartNMPC/Nonlinear MPC Controller" is configured to use it.
```

Input Arguments

nlmpcobj — Nonlinear MPC controller
nlmpc object

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

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

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

busName — Name of Simulink bus object
string | character vector

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

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

parameters — Nominal prediction model parameter values
cell array

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

Version History
Introduced in R2018b

See Also

Functions
nlmpcmove

Objects
nlmpc | nlmpcmoveopt

Blocks
Nonlinear MPC Controller

Topics
"Specify Prediction Model for Nonlinear MPC"

d2d

Change sampling time of an MPC controller

Syntax

```
newmpc = d2d(mpcobj,newTs)
```

Description

Use the Model Predictive Control Toolbox `d2d` function to change the sampling time of an MPC controller (see `mpc` for background).

To resample a generic discrete-time LTI dynamical system instead, see `d2d`.

`newmpc = d2d(mpcobj,newTs)` returns the controller `newmpc`, which is identical to `mpcobj` except for the new sample time `newTs`. This is equivalent to coying `mpcobj` in a new object `newmpc` and assigning a new sample using either `newmpc.Ts=newTs` or `set(newmpc,'Ts',newTs)`. All models in `newmpc` are sampled or resampled when the QP matrices must be computed, for example when `sim` or `mpcmove` are called.

Examples

Change sampling time of MPC controller

Create a plant, a corresponding MPC object, and create a new controller with a different sampling time.

```
mpcverbosity off;                       % turn off mpc messaging
plant=tf(1,[1 1]);                      % create plant (note the steady state gain)
mpcobj=mpc(plant,1);                    % create mpc object (1 second sampling time)

newmpc=d2d(mpcobj,0.2);                 % change sampling time to 0.2 seconds
newmpc.Ts
ans =
    0.2000

newmpc.Ts=1;                            % change sampling time back to 1 second
compare(newmpc,mpcobj)                  % compare the two controllers
ans =
  logical
   1
```

Input Arguments

`mpcobj` — Model predictive controller
mpc object

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

newTs — Sampling Time
positive scalar

This is the new sampling time for the returned, resampled, MPC controller `mpcobjTs`.

Example: 0.2

Output Arguments

newmpc — MPC controller with the new sampling time
`mpc` object

This is the returned MPC controller, which is identical to `mpcobj` except for the fact that its sampling time is now `newTs`. The internal models of `newmpc` are sampled or resampled when the QP matrices must be computed for the MPC optimization problem to be solved.(for example when `sim` or `mpcmove` are called).

Version History
Introduced before R2006a

See Also

Functions
`set` | `sim` | `ss` | `tf` | `zpk` | `size` | `cloffset` | `review` | `sensitivity` | `mpcprops` | `get` | `set` | `trim`

Objects
`mpc`

Topics
"Extract Controller"

generateExplicitMPC

Convert implicit MPC controller to explicit MPC controller

Syntax

```
empcobj = generateExplicitMPC(mpcobj,range)
empcobj = generateExplicitMPC(mpcobj,range,opt)
```

Description

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

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

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

Examples

Generate Explicit MPC Controller

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

Define the double-integrator plant.

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

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

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

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

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

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

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

Generate the explicit MPC controller.

```
empcobj = generateExplicitMPC(mpcobj,range,opt)


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

Input Arguments

`mpcobj` — Traditional MPC controller
MPC controller object

Traditional MPC controller, specified as an `mpc` object

`range` — Parameter bounds
structure

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

`opt` — optimization options
structure

Optimization options for the conversion computation, specified as a structure that you create with the generateExplicitOptions function. For detailed descriptions of these options, see generateExplicitOptions.

Output Arguments

empcobj — Explicit MPC controller
explicitMPC object

Explicit MPC controller that is equivalent to the input traditional controller, returned as an explicitMPC object.

Property	Description
MPC	Traditional (implicit) controller object used to generate the explicit MPC controller. You create this MPC controller using is the mpc command. It is the first argument to generateExplicitMPC when you create the explicit MPC controller.
range	1-D structure containing the parameter bounds used to generate the explicit MPC controller. These determine the resulting controller's valid operating range. This property is automatically populated by the range input argument to generateExplicitMPC when you create the explicit MPC controller. See generateExplicitRange for details about this structure.
OptimizationOptions	1-D structure containing user-modifiable options used to generate the explicit MPC controller. This property is automatically populated by the opt argument to generateExplicitMPC when you create the explicit MPC controller. See generateExplicitOptions for details about this structure.
PiecewiseAffineSolution	n_r-dimensional structure, where n_r is the number of piecewise affine (PWA) regions required to represent the control law. The ith element contains the details needed to compute the optimal manipulated variables when the solution lies within the ith region. See "Implementation".
IsSimplified	Logical switch indicating whether the explicit control law has been modified using the simplify command such that the explicit control law approximates the base (implicit) MPC controller. If the control law has not been modified, the explicit controller should reproduce the base controller's behavior exactly, provided both operate within the bounds described by the range property.

Tips

- Using Explicit MPC, you will most likely achieve best performance in small control problems, which involve small numbers of plant inputs/outputs/states as well as the number of constraints.

- Test the implicit controller thoroughly before attempting a conversion. This helps to determine the range of controller states and other parameters needed to generate the explicit controller.

- Simulate the explicit controller's performance using the `sim` or `mpcmoveExplicit` commands, or the Explicit MPC Controller block in Simulink.

- `generateExplicitMPC` displays progress messages in the command window. Use `mpcverbosity` to turn off the display.

Version History
Introduced in R2014b

See Also

Functions
generateExplicitRange | generateExplicitOptions | simplify | generatePlotParameters | plotSection | mpcmoveExplicit

Objects
mpc | explicitMPC

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

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

generateExplicitOptions

Optimization options for explicit MPC generation

Syntax

```
opt = generateExplicitOptions(mpcobj)
```

Description

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

Examples

Generate Explicit MPC Controller

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

Define the double-integrator plant.

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

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

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

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

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

```
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

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

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

Generate the explicit MPC controller.

```
empcobj = generateExplicitMPC(mpcobj,range,opt)
```

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

Input Arguments

`mpcobj` — Traditional MPC controller
MPC controller object

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

Output Arguments

`opt` — Options for generating explicit MPC controller
structure

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

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

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

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

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

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

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

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

Constraint normalization tolerance, specified as a positive scalar value.

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

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

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

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

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

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

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

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

Version History
Introduced in R2014b

See Also

Functions
`generateExplicitMPC` | `generateExplicitRange` | `simplify` | `generatePlotParameters` | `plotSection` | `mpcmoveExplicit` | `sim`

Objects
`mpc` | `explicitMPC` | `mpcstate`

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"

"What is Model Predictive Control?"

generateExplicitRange

Bounds on explicit MPC control law parameters

Syntax

```
range = generateExplicitRange(mpcobj)
```

Description

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

Examples

Generate Explicit MPC Controller

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

Define the double-integrator plant.

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

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

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

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
```

```
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

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

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

Generate the explicit MPC controller.

```
empcobj = generateExplicitMPC(mpcobj,range,opt)


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

Input Arguments

mpcobj — Traditional MPC controller
MPC controller object

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

Output Arguments

range — Parameter bounds
structure

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

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

The fields of the range structure are as follows.

State — Bounds on controller state values
structure

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

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

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

Reference — Bounds on controller reference signal values
structure

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

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

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

MeasuredDisturbance — Bounds on measured disturbance values
structure

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

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

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

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

ManipulatedVariable — Bounds on manipulated variable values
structure

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

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

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

Version History

Introduced in R2014b

See Also

Functions
generateExplicitMPC | generateExplicitOptions | simplify | generatePlotParameters | plotSection | mpcmoveExplicit | sim

Objects
mpc | explicitMPC | mpcstate

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"
"What is Model Predictive Control?"

generateJacobianFunction

Generate MATLAB Jacobian functions for multistage nonlinear MPC using automatic differentiation

Syntax

```
newMSobj = generateJacobianFunction(oldMSobj,type)
generateJacobianFunction( ___ ,Name=Value)
```

Description

`newMSobj = generateJacobianFunction(oldMSobj,type)` uses automatic differentiation to generate a MATLAB function that calculates the derivatives, with respect to the plant states and inputs, of a function specified in the multistage nonlinear MPC object `oldMSobj` and indicated by `type`.

After successfully generating and saving the function file (along with two supporting files) in the same folder as the one in which the original function is located, `generateJacobianFunction` returns the new multistage nonlinear MPC object `newMSobj`. This returned object is the same as the original object `oldMSobj` but has its corresponding Jacobian function fields (in its `Model` and `Stages` properties) updated to use the generated Jacobian function.

The name of the generated Jacobian function is the same of the original function with the `Jacobian` appended at the end. If a stage of `oldMSobj` does not have any cost or constraint function defined, no cost or constraint Jacobian function is generated for that stage.

`generateJacobianFunction( ___ ,Name=Value)` also specifies the stage number and whether to save the generated function files in the current folder, using one or more name-value pair arguments.

Examples

Generate Jacobian Functions for Multistage Nonlinear MPC

Create a multistage nonlinear MPC object with a horizon of ten steps, for a plant with six states and two manipulated variables.

```
msobj = nlmpcMultistage(10,6,2);
```

Write a simple state function as a string.

```
sstr = "function xdot = mystatefcn(x,u)" + newline + ...
       "xdot = ones(6,2)*u-x.^3;" + newline + "end";
```

Write the string to the `mystatefcn.m` file.

```
sfid=fopen("mystatefcn.m","w");
fwrite(sfid,sstr,"char");
fclose(sfid);
```

Write a simple cost function as a string.

```
cstr = "function c = mycostfcn(k,x,u)" + newline + ...
       "c = k*u'*u + k*x'*x;" + newline + "end";
```

Write the string to the `mycostfcn.m` file.

```
cfid=fopen("mycostfcn.m","w");
fwrite(cfid,cstr,"char");
fclose(cfid);
```

Alternatively, you can write your state and cost MATLAB functions in the current folder (local functions are supported for the generation of Jacobians, but not for the generation of deployment code).

Specify the state transition function for the prediction model (`mystatefcn` is defined at the end of this example).

```
msobj.Model.StateFcn = "mystatefcn";
```

Specify the cost functions for all stages except the first two (`mycostfcn` is defined at the end of the file).

```
for i=3:10
    msobj.Stages(i).CostFcn = "mycostfcn";
end
```

Generate the state Jacobian function. This function calculates the derivatives of the state function with respect to the plant states and inputs.

```
msobj = generateJacobianFunction(msobj,"state");
```

Generate the cost Jacobian function. This function calculates the derivatives of each stage cost function with respect to the plant states and inputs.

```
msobj = generateJacobianFunction(msobj,"cost");
```

Display the generated cost Jacobian function.

```
type mycostfcnJacobian.m
```

```
function [Jx, Ju] = mycostfcnJacobian(stage, x, u)
% This function was generated by Model Predictive Control Toolbox (Version 8.0).
% 29-Nov-2022 19:38:51
%# codegen
persistent ADdata
if isempty(ADdata)
    ADdata = coder.load('mycostfcnJacobianADdata','constants');
end
params.stage = stage;
[~,J] = mycostfcnJacobianAD([x;u],ADdata.constants,params);
Jx = J(1:6,:);
Ju = J([7 8],:);
```

Display the Jacobian fields of the `Model` and `Stages` properties of `msobj`.

```
msobj.Model.StateJacFcn
```

```
ans =
'mystatefcnJacobian'
```

```
msobj.Stages(2).CostFcn

ans =

     []

msobj.Stages(3).CostFcn

ans =
'mycostfcn'
```

Validate all the functions specified in `msobj` for a random plant state and input.

```
validateFcns(msobj,rand(6,1),rand(2,1));

Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [3 4 5 6 7 8 9 10] are OK.
"CostJacFcn" of the following stages [3 4 5 6 7 8 9 10] are OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Input Arguments

`oldMSobj` — Original multistage MPC controller
`nlmpcMultistage` object

Original multistage nonlinear MPC controller, specified as an `nlmpcMultistage` object.

`type` — Type of function
`"state"` | `"cost"` | `"ineqcon"` | `"eqcon"`

Type of function for which Jacobians are generated, specified as one of the following strings:

- `"state"` — Generate Jacobian functions for the state function.
- `"cost"` — Generate Jacobian functions for the cost function.
- `"ineqcon"` — Generate Jacobian functions for the inequality constraint function.
- `"eqcon"` — Generate Jacobian functions for the equality constraint function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FunctionName="computeAction"`

`SaveInSameFolder` — Option to save generated function files in same folder
`true` or `1` (default) | `false` or `0`

Option to save generated function files in the same folder as the one in which the original function is located, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you set this option to `false`, the generated function files are saved in the current MATLAB folder (the folder returned by the `pwd` command).

Note If the original state, cost or constraint function is specified in `oldMSobj` as a handle to a local function (that is a function defined in the same script file that you execute), then its Jacobian function is always saved in the current folder, even if `SaveInSameFolder` is specified as `true`.

Example: `SaveInSameFolder=false`

StageNumber — Stage number
0 (default) | nonnegative integer

Stage number for which the Jacobian cost or constraint function must be generated, specified as a nonnegative integer not grater than the number of stages. When you specify a positive number, Jacobian functions are generated only for the specified stage number. If you specify `0` (default), Jacobians are generated for all the stages.

Example: `StageNumber=3`

Output Arguments

newMSobj — New nonlinear multistage MPC controller
`nlmpcMultistage` object

New nonlinear multistage MPC controller, returned as an `nlmpcMultistage` object. It is the same as the original object `oldMSobj` but has its corresponding Jacobian function fields (in its `Model` and `Stages` properties) updated to use the generated MATLAB function.

Limitations

- Automatic differentiation currently supports only a limited set of mathematical operations, which are described in "Supported Operations for Optimization Variables and Expressions" (Optimization Toolbox). If your original function uses operations or functions that are not in the list, or has if-else statements or loops, `generateJacobianFunction` terminates with an error.

- To generate Jacobian functions, do not preallocate any optimization variable. For example, suppose you try to generate Jacobians from a function containing the following code.

```
dxdt = zeros(2,1);
dxdt(1) = x(1)*x(2);
dxdt(2) = x(1)/x(2);
```

This code results in the following error.

```
Unable to perform assignment because value of type
'optim.problemdef.OptimizationExpression'
is not convertible to 'double'.
```

Instead, use the following code.

```
dxdt = [x(1)*x(2); x(1)/x(2)];
```

Note that you can use conditional preallocation using `optimexpr` to preallocate variables; however, doing so results in a Jacobian function that does not support C/C++ code generation for deployment. Therefore, the best practice is to avoid preallocation entirely. For more information, see "Initialize Optimization Expressions" (Optimization Toolbox).

- Specifying the state, cost, and constraint functions in `oldMSobj` as files in the current folder or in a folder on the MATLAB path is recommended. While handles to local functions are supported for

Jacobian function generation, they are not supported for generation of C/C++ deployment code. For more information on local functions, see "Local Functions".

Version History

Introduced in R2023a

See Also

Functions
validateFcns | nlmpcmove | nlmpcmoveCodeGeneration | getSimulationData | getCodeGenerationData

Objects
nlmpcMultistage | nlmpcmoveopt

Blocks
Multistage Nonlinear MPC Controller

Topics
"Landing a Vehicle Using Multistage Nonlinear MPC"
"Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC"
"Nonlinear MPC"
"Initialize Optimization Expressions" (Optimization Toolbox)
"Supported Operations for Optimization Variables and Expressions" (Optimization Toolbox)

generatePlotParameters

Parameters for `plotSection`

Syntax

```
plotParams = generatePlotParameters(empcobj)
```

Description

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

Examples

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

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

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

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

Define the parameter bounds for generating an explicit MPC controller.

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
range.State.Min(:) = [-4;-4];
range.State.Max(:) = [ 4; 4];
range.Reference.Min(:) = -2;
range.Reference.Max(:) = 2;
range.ManipulatedVariable.Min(:) = -1.1;
range.ManipulatedVariable.Max(:) = 1.1;
```

Create an explicit MPC controller.

```
empcobj = generateExplicitMPC(mpcobj,range);
```

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

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

```
plotParams = generatePlotParameters(empcobj);
```

Allow the controller states to vary when creating a plot.

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

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

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

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

```
plotSection(empcobj,plotParams)
```



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

      Number: 1
        Name: 'PiecewiseAffineSectionPlot'
       Color: [1 1 1]
    Position: [360 402 560 420]
```

```
    Units: 'pixels'

  Show all properties
```

Input Arguments

empcobj — Explicit MPC controller
explicit MPC controller object

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

Output Arguments

plotParams — Parameters for sectional plot
structure

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

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

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

State — Fixed controller states
structure

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

Modify the default value of `plotParams.State` to generate the desired plot. See "Specify Fixed Parameters for 2-D Plot of Explicit Control Law" on page 2-43.

Reference — Fixed reference signal values
structure

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

Modify the default value of `plotParams.Reference` to generate the desired plot. See "Specify Fixed Parameters for 2-D Plot of Explicit Control Law" on page 2-43.

MeasuredDisturbance — Fixed measured disturbance values
structure

Fixed measured disturbance values, specified as a structure having an `Index` field and a `Value` field. The field `plotParams.MeasuredDisturbance.Index` is a vector that contains the indices of the

measured disturbances to fix for the plot, and `plotParams.MeasuredDisturbance.Value` contains the corresponding constant measured disturbance values.

Modify the default value of `plotParams.MeasuredDisturbance` to generate the desired plot. See "Specify Fixed Parameters for 2-D Plot of Explicit Control Law" on page 2-43.

`ManipulatedVariable` — Fixed manipulated variable values
structure

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

Modify the default value of `plotParams.ManipulatedVariable` to generate the desired plot. See "Specify Fixed Parameters for 2-D Plot of Explicit Control Law" on page 2-43.

Version History
Introduced in R2014b

See Also

Functions
generateExplicitMPC | generateExplicitRange | generateExplicitOptions | simplify | plotSection | mpcmoveExplicit | sim

Objects
mpc | explicitMPC | mpcstate

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"
"What is Model Predictive Control?"

get

Get property values from MPC object

Syntax

```
PropertyValue = get(mpcobj,PropertyName)
Struct = get(mpcobj)
get(mpcobj)
```

Description

Use the Model Predictive Control Toolbox `get` function to read the property values of an MPC controller (see `mpc` for background).

To implement Get/Set interface of standard MATLAB object, see "Implement Set/Get Interface for Properties".

`PropertyValue = get(mpcobj,PropertyName)` returns the current value of the property `PropertyName` of the MPC controller `mpcobj`.

`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.

Examples

Get property values from an MPC object

Create plant model and related MPC object

```
mpcverbosity off;                          % turn off mpc messages

% create plant model
plant = rss(4,4,4);                        % random state space
plant.D = 0;                               % set D matrix to zero

mpcobj=mpc(plant,1);
```

Get values of some properties

```
>> get(mpcobj,'Ts')
ans =
     1
>> get(mpcobj,"Ts")
ans =
     1
>> mpcobj.Ts
ans =
     1
```

```
>> get(mpcobj,'ControlHorizon')
ans =
      2
>> get(mpcobj,'Model')
ans =
  struct with fields:

          Plant: [4×4 ss]
    Disturbance: []
          Noise: []
        Nominal: [1×1 struct]

% display all properties
get(mpcobj)
                            Ts: 1
       PredictionHorizon (P): 10
          ControlHorizon (C): 2
                        Model: [1x1 struct]
   ManipulatedVariables (MV): [1x4 struct]
        OutputVariables (OV): [1x4 struct]
   DisturbanceVariables (DV): []
                  Weights (W): [1x1 struct]
                    Optimizer: [1x1 struct]
                        Notes: {}
                     UserData: []
                      History: 11-Sep-2020 16:50:19


% get whole MPC structure
WholeMPCStruct=get(mpcobj);

% display History field
WholeMPCStruct.History
ans =
   1.0e+03 *
   2.0200    0.0090    0.0110    0.0160    0.0500    0.0193
```

Input Arguments

mpcobj — Model predictive controller
mpc object

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

PropertyName — Name of MPC object property
character array | string

Specify PropertyName as a character array or string that contains the full property name (for example, 'UserData') or any unambiguous case-insensitive abbreviation (for example, 'user' instead of 'UserData'). You can specify any generic MPC property.

Example: 'Model'

Output Arguments

PropertyValue — Value of MPC object property
double | matrix | structure | other

The value returned in `PropertyValue` depends on the specific property of the MPC object. See `mpcprops` for more information on MPC object properties.

Struct — Structure containing all property values
double | matrix | structure | other

This is a standard MATLAB structure containing all the property names of the MPC object as field names and the property values as field values. See `mpcprops` for more information on MPC object properties.

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`.

Version History
Introduced before R2006a

See Also

Functions
set | mpcprops | getname | setname | getindist | getoutdist | setindist | setoutdist |
getconstraint | setconstraint | getEstimator | setEstimator | setterminal | review |
sim

Objects
mpc | mpcsimopt

Blocks
MPC Controller

getCodeGenerationData

Create data structures for `mpcmoveCodeGeneration`

Syntax

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

Description

Use this function to create data structures for the `mpcmoveCodeGeneration` function, which computes optimal control moves for implicit and explicit linear MPC controllers.

For information on generating data structures for `nlmpcmoveCodeGeneration`, see `getCodeGenerationData`.

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

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

Examples

Create MPC Code Generation Data Structures

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

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

Create an MPC controller.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

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

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

Create code generation data structures.

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

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

Specify Options for Creating MPC Code Generation Structures

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

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

Create an MPC controller.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Create code generation data structures. Configure options to:

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

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

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Input Arguments

mpcobj — Model predictive controller
mpc object | explicitMPC object

Model predictive controller, specified as one of the following:

- `mpc` object — MPC controller
- `explicitMPC` object — Explicit MPC controller created using `generateExplicitMPC`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

InitialState — Initial controller state
`mpcstate` object

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

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

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

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

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

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

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

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

For more information on adaptive MPC, see "Adaptive MPC".

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

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

Time-varying MPC indicator when using `mpcmoveCodeGeneration`, specified as the comma-separated pair consisting of `'IsLTV'` and either `true` or `false`. Set `IsLTV` to `true` if your controller is running in time-varying mode.

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

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

UseVariableHorizon — Variable horizon indicator
`false` (default) | `true`

Variable horizon indicator when using `mpcmoveCodeGeneration`, specified as the comma-separated pair consisting of `'UseVariableHorizon'` and either `true` or `false`. To vary your prediction and control horizons at run time, set `UseVariableHorizons` to `true`.

When you use variable horizons, `mpcmoveCodeGeneration` ignores the horizons specified in `configData` and instead uses the prediction and control horizon specified in `onlineData.horizons`.

For more information, see "Adjust Horizons at Run Time".

Output Arguments

configData — MPC configuration parameters
structure

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

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

stateData — Initial controller states
structure

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

For more information on the `stateData` fields, see `mpcmoveCodeGeneration`.

`stateData` has the following fields.

Field	Description
`Plant`	Plant model state estimates
`Disturbance`	Unmeasured disturbance model state estimates
`Noise`	Output measurement noise model state estimates
`LastMove`	Manipulated variable control moves from previous control interval
`Covariance`	Covariance matrix for controller state estimates

Field	Description
iA	Active inequality constraints

onlineData — Online MPC controller data
structure

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

Field	Description
signals	Input and output signals, returned as a structure with the following fields. <table><tr><td>**Field**</td><td>**Description**</td></tr><tr><td>ym</td><td>Measured outputs</td></tr><tr><td>ref</td><td>Output references</td></tr><tr><td>md</td><td>Measured disturbances</td></tr><tr><td>mvTarget</td><td>Targets for manipulated variables</td></tr><tr><td>externalMV</td><td>Manipulated variables externally applied to the plant</td></tr></table>
limits	Input and output constraints, returned as a structure with the following fields: <table><tr><td>**Field**</td><td>**Description**</td></tr><tr><td>ymin</td><td>Lower bounds on output signals</td></tr><tr><td>ymax</td><td>Upper bounds on output signals</td></tr><tr><td>umin</td><td>Lower bounds on input signals</td></tr><tr><td>umax</td><td>Upper bounds on input signals</td></tr></table> When mpcobj is an explicit MPC controller, mpcmoveCodeGeneration ignores the limits field.
weights	Updated QP optimization weights, returned as a structure with the following fields: <table><tr><td>**Field**</td><td>**Description**</td></tr><tr><td>ywt</td><td>Output weights</td></tr><tr><td>uwt</td><td>Manipulated variable weights</td></tr><tr><td>duwt</td><td>Manipulated variable rate weights</td></tr><tr><td>ecr</td><td>Weight on slack variable used for constraint softening</td></tr></table> When mpcobj is an explicit MPC controller, mpcmoveCodeGeneration ignores the weights field.

Field	Description
customconstraints	Updated custom mixed input/output constraints, returned as a structure with the following fields: | Field	Description
E	Manipulated variable constraint constant
F	Controlled output constraint constant
G	Mixed input/output constraint constant
S	Measured disturbance constraint constant | When mpcobj is an explicit MPC controller, mpcmoveCodeGeneration ignores the customconstraints field.
horizons	Updated controller horizon values, returned as a structure with the following fields: | Field	Description
p	Prediction horizon
m	Control horizon | The horizons field is returned only when the UseVariableHorizon name-value pair is true. When mpcobj is an explicit MPC controller, mpcmoveCodeGeneration ignores the horizons field.
model	Updated plant and nominal values for adaptive MPC and time-varying MPC, returned as a structure with the following fields: | Field	Description
A, B, C, D	State-space matrices of discrete-time state-space model.
X	Nominal plant states
U	Nominal plant inputs
Y	Nominal plant outputs
DX	Nominal plant state derivatives | The model field is returned only when either the IsAdaptive or IsLTV name-value pair is true.

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

For more information on configuring onlineData fields, see mpcmoveCodeGeneration.

Version History
Introduced in R2016a

See Also

Functions
mpcmoveCodeGeneration | sim | buildMEX | codegen

Objects
mpcstate | mpcsimopt | mpcmoveopt | explicitMPC

Topics
"Generate Code to Compute Optimal MPC Moves in MATLAB"
"Generate Code and Deploy Controller to Real-Time Targets"

getCodeGenerationData

Create data structures for nlmpcmoveCodeGeneration

Syntax

```
[coreData,onlineData] = getCodeGenerationData(nlobj,x,lastMV)
[coreData,onlineData] = getCodeGenerationData(nlobj,x,lastMV,params)
[ ___ ] = getCodeGenerationData( ___ ,field)
[ ___ ] = getCodeGenerationData( ___ ,field1,...,fieldn)
```

Description

Use this function to create data structures for the nlmpcmoveCodeGeneration function, which computes optimal control moves for nonlinear MPC controllers.

For information on generating data structures for mpcmoveCodeGeneration, see getCodeGenerationData.

[coreData,onlineData] = getCodeGenerationData(nlobj,x,lastMV) creates data structures for use with nlmpcmoveCodeGeneration.

[coreData,onlineData] = getCodeGenerationData(nlobj,x,lastMV,params) copies initial parameter values in the onlineData structure if nlobj is an nlmpc object. If nlobj is an nlmpcMultistage object then passing the params argument is not allowed and you have to manually specify the initial guesses in the InitialGuess field of onlineData instead.

[___] = getCodeGenerationData(___ ,field) enables the specified online weight or constraint field by adding it to the onlineData structure.

[___] = getCodeGenerationData(___ ,field1,...,fieldn) enables multiple online weight or constraint fields by adding them to the onlineData structure.

Examples

Create Nonlinear MPC Code Generation Structures

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

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

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

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

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

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

The prediction model uses an optional parameter `Ts` to represent the sample time. Specify the number of parameters and create a parameter vector.

```
nlobj.Model.NumberOfParameters = 1;
params = {Ts};
```

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

```
nlobj.Model.OutputFcn = "pendulumOutputFcn";
```

Define standard constraints for the controller.

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

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj,x0,u0,[],params);
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Kalman filter for estimating the four plant states. Its state transition function is defined in `pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero initial manipulated variable value.

```
x0 = [0;0;-pi;0];
y0 = [x0(1);x0(3)];
EKF.State = x0;
mv0 = 0;
```

Create code generation data structures for the controller, specifying the initial conditions and parameters.

```
[coreData,onlineData] = getCodeGenerationData(nlobj,x0,mv0,params);
```

View the online data structure.

```
onlineData
```

```
onlineData = struct with fields:
          ref: [0 0]
     MVTarget: 0
   Parameters: {[0.1000]}
           X0: [10x4 double]
          MV0: [10x1 double]
        Slack0: 0
```

If your application uses online weights or constraints, you must add corresponding fields to the code generation data structures. For example, the following syntax creates data structures that include fields for output variable tuning weights, manipulated variable tuning weights, and manipulated variable bounds.

```
[coreData2,onlineData2] = getCodeGenerationData(nlobj,x0,mv0,params,...
    'OutputWeights','MVWeights','MVMin','MVMax');
```

View the online data structure. At run time, specify the online weights and constraints in the added structure fields.

```
onlineData2
```

```
onlineData2 = struct with fields:
             ref: [0 0]
        MVTarget: 0
      Parameters: {[0.1000]}
              X0: [10x4 double]
             MV0: [10x1 double]
          Slack0: 0
   OutputWeights: [3 3]
       MVWeights: 0
           MVMin: [10x1 double]
           MVMax: [10x1 double]
```

Input Arguments

nlobj — Nonlinear model predictive controller
nlmpc object | nlmpcMultistage object

Nonlinear model predictive controller, specified as an nlmpc or nlmpcMultistage object.

x — Initial states of nonlinear prediction model
column vector of length N_x

Initial states of the nonlinear prediction model, specified as a column vector of length N_x, where N_x is the number of prediction model states.

lastMV — Initial manipulated variable control signals
column vector of length N_{mv}

Initial manipulated variable control signals, specified as a column vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

params — Initial parameter values for non multistage MPC
cell vector

Initial parameter values for non multistage MPC, specified as a cell vector with length equal to `nlobj.Model.NumberOfParameters`, which is the number of optional parameters in the controller prediction model. If the controller has no optional parameters, specify `params` as `{}`.

If `nlobj` is an `nlmpc` object then the initial values specified in `params` are copied into the `onlineData` structure. If `nlobj` is an `nlmpcMultistage` object then the `params` argument is not allowed and you have to manually specify the initial guesses in the `InitialGuess` field of `onlineData` instead.

For more information on optional prediction model parameters, see "Specify Prediction Model for Nonlinear MPC".

`field` — Online weight or constraint field name
string | character vector

Online weight or constraint field name, specified as a string or character vector. When creating data structures for `nlmpcmoveCodeGeneration`, you can add any of the following fields to the `onlineData` output structure. Add a given field to the online data structure only if you expect the corresponding weight or constraint to vary at run time.

Online Constraints

- `"StateMin"` — State lower bounds
- `"StateMax"` — State upper bounds
- `"MVMin"` — Manipulated variable lower bounds
- `"MVMax"` — Manipulated variable upper bounds
- `"MVRateMin"` — Manipulated variable rate of change lower bound
- `"MVRateMax"` — Manipulated variable rate of change upper bound

Online constraints and Tuning Weights for non-Multistage MPC

- `"OutputWeights"` — Output variable weights
- `"MVWeights"` — Manipulated variable weights
- `"MVRateWeights"` — Manipulated variable rate weights
- `"ECRWeight"` — Slack variable weight
- `"OutputMin"` — Output variable lower bounds
- `"OutputMax"` — Output variable upper bounds

Disturbances, Parameters, and Initial Guesses for Multistage MPC

- `"MeasuredDistrubance"` — Measured disturbances
- `"StateParameter"` — Parameter vector for state function and Jacobians
- `"StageParameter"` — Parameter vector for stage cost, constraints, and Jacobians
- `"TerminalState"` — Terminal state constraint
- `"InitialGuess"` — Initial guesses for decision variables

Output Arguments

`coreData` — Nonlinear MPC configuration parameters
structure

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

onlineData — Online nonlinear MPC controller data
structure

Run-time simulation data, returned as a structure. The fields in the structure depend on whether `nlobj` is an `nlmpc` object or an `nlmpcMultistage` object. During a simulation, you must supply this structure as an input to `nlmpcmoveCodeGeneration` at every control interval. `nlmpcmoveCodeGeneration` then returns as output the updated structure that you will need to supply as input in the following control interval.

`Non-Multistage MPC` — Structure for generic MPC controllers
structure

For `nlmpc` objects, the structure always contains the following fields.

Field	Description
`ref`	Output reference values, returned as a column vector of zeros with length N_y, where N_y is the number of prediction model outputs.
`mvTarget`	Manipulated variable reference values, returned as a column vector of zeros with length N_{mv}, where N_{mv} is the number of manipulated variables.
`X0`	Initial guess for the state trajectory, returned as a column vector equal to `x`.
`MV0`	Initial guess for the manipulated variable trajectory, returned as a column vector equal to `lastMV`.
`Slack0`	Initial guess for the slack variable, returned as zero.

For `nlmpc` objects, `onlineData` can also contain the following fields, depending on the controller configuration and argument values.

Field	Description
`md`	Measured disturbance values — This field is returned only when the controller has measured disturbance inputs, that is, when `nlobj.Dimensions.MDIndex` is nonzero. `md` is returned as a column vector of zeros with length N_{md}, where N_{md} is the number of measured disturbances.
`Parameters`	Parameter values — This field is returned only when the controller uses optional model parameters. `Parameters` is returned as a cell vector equal to `params`.

Field	Description
• `OutputWeights` • `MVWeights` • `MVRateWeights` • `ECRWeight` • `OutputMin` • `OutputMax` • `StateMin` • `StateMax` • `MVMin` • `MVMax` • `MVRateMin` • `MVRateMax`	Weight and constraint values — Each field is returned only when the corresponding field name is specified using the `field` argument. The value of each field is equal to the corresponding default value defined in the controller, as returned in `coreData`.

For more information on configuring `onlineData` fields, see `nlmpcmoveCodeGeneration`.

Multistage MPC — Structure for multistage MPC controllers
structure

For `nlmpcMultistage` objects, the returned `onlineData` structure always contains the `InitialGuess` field.

Field	Description
`InitialGuess`	Initial guess for the decision variables, returned as a column vector of length equal to the sum of the lengths of all the decision variable vectors for each stage. For more information, see `nlmpcmove`.

For `nlmpcMultistage` objects, `onlineData` can also contain the following fields, depending on the controller configuration and argument values.

Field	Description
`MeasuredDisturbances`	Measured disturbance values — This field is returned only when the controller has measured disturbance inputs, that is, when `nlobj.Dimensions.MDIndex` is nonzero. md is returned as a column vector of zeros with length N_{md}, where N_{md} is the number of measured disturbances.
`StateFcnParameters`	Parameter values for state functions and Jacobians — This field is returned only when the controller state prediction function or its Jacobian use model parameters, that is when `Model.ParameterLength` is greater than zero. `StateFcnParameter` is returned as a vector.
`StageFcnParameters`	Parameter values for stage cost and constraints functions and their Jacobians— This field is returned only when any stage cost or constraint function, or its Jacobian, uses parameters, that is when there is at least one stage `i` for which `Stages(i).ParameterLength` is greater than zero. `StageFcnParameter` is returned as a vector.

Field	Description
• `StateMin` • `StateMax` • `MVMin` • `MVMax` • `MVRateMin` • `MVRateMax`	Constraint values — Each field is returned only when the corresponding field name is specified using the `field` argument. The value of each field is equal to the corresponding default value defined in the controller, as returned in `coreData`.
`TerminalState`	Terminal state, returned as a column vector with as many elements as the number of states. The terminal state is the desired state at the last prediction step. To specify desired terminal states at run-time via this field, you must specify finite values in the `TerminalState` field of the `Model` property of `nlobj`. Specify `inf` for the states that do not need to be constrained to a terminal value. At run time, `nlmpcmoveCodeGeneration` ignores any values in the `TerminalState` field of `simdata` that correspond to `inf` values in `nlobj`. If you do not specify any terminal value condition in `nlobj`, this field is not created in `onlinedata`.

For more information on configuring `onlineData` fields, see `nlmpcmove` and `nlmpcmoveCodeGeneration`.

Version History
Introduced in R2020a

See Also

Functions
validateFcns | nlmpcmove | getSimulationData | nlmpcmoveCodeGeneration | buildMEX | codegen

Objects
nlmpc | nlmpcMultistage

Topics
"Generate Code to Compute Optimal MPC Moves in MATLAB"
"Nonlinear MPC"
"Generate Code and Deploy Controller to Real-Time Targets"

getconstraint

Obtain mixed input/output constraints from model predictive controller

Syntax

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

Description

`[E,F,G,V,S] = getconstraint(mpcobj)` returns the mixed-input/output constraints previously defined for the MPC controller, `mpcobj`. For more information, see "Mixed Input/Output Constraints" on page 2-66.

Examples

Retrieve Custom Constraints from MPC Controller

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

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

Create an MPC controller for this plant.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Assume that you have two soft constraints.

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

Set the constraints for the MPC controller.

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

Retrieve the constraints from the controller.

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

```
E = 2×2

     1     1
     0     0


F = 2×2

     0     0
     0     1


G = 2×1

     5
    10


V = 2×1

     1
     1


S = 2×1

     0
     1
```

Input Arguments

mpcobj — Model predictive controller
mpc object

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

Output Arguments

E — Manipulated variable constraint constant
N_c-by-N_{mv} array | [ ]

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

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

F — Controlled output constraint constant
N_c-by-N_y array | [ ]

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

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

`G` — **Mixed input/output constraint constant**
column vector of length N_c | [ ]

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

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

`V` — **Constraint softening constant**
column vector of length N_c | [ ]

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

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

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

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

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

`S` — **Measured disturbance constraint constant**
N_c-by-N_v array | [ ]

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

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

Algorithms

Mixed Input/Output Constraints

The general form of the mixed input/output constraints is:
$$Eu(k + j) + Fy(k + j) + Sv(k + j) \le G + \varepsilon V$$

Here, $j = 0,...,p$, and:

- p is the prediction horizon.
- k is the current time index.
- u is a column vector manipulated variables.
- y is a column vector of all plant output variables.
- v is a column vector of measured disturbance variables.
- ε is a scalar slack variable used for constraint softening (as in "Standard Cost Function").

- *E*, *F*, *G*, *V*, and *S* are constant matrices.

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

Version History
Introduced in R2011a

See Also

Functions
setconstraint | get | set | mpcprops | setterminal

Objects
mpc

Topics
"Design MPC Controller at the Command Line"
"Constraints on Linear Combinations of Inputs and Outputs"
"Update Constraints at Run Time"
"What is Model Predictive Control?"

getEstimator

Obtain Kalman gains and model for estimator design

Syntax

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

Description

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

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

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

Examples

Extract Parameters for State Estimation

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

Fix the number generator seed to ensure repeatable results.

```
rng(0)
```

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

```
plant = drss(4,3,3);
plant.Ts = 0.25;
plant = setmpcsignals(plant,MV=[1,2],UD=3,MO=[1 3],UO=2);
plant.d(:,[1,2]) = 0;
```

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

Calculate the default model predictive controller for this plant.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
```

```
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 y3 and zero weight for output(s) y2
```

Obtain the parameters to be used in state estimation.

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

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

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

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

```
Poles = 6×1

   -0.5409
   -0.0026
    0.2358
    0.7628
    0.8595
    0.9718
```

Confirm that the default estimator is asymptotically stable.

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

```
ans = 0.9718
```

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

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

```
L - A*M
```

```
ans = 6×2
10^{-15} ×

    0.4163   -0.0971
   -0.1665    0.1665
   -0.4025    0.3053
    0.2220    0.0139
    0.0971         0
    0.2359   -0.0590
```

Input Arguments

mpcobj — MPC controller
MPC controller object

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

Output Arguments

L — Kalman gain matrix for time update
matrix

Kalman gain matrix for the time update, returned as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs.

M — Kalman gain matrix for measurement update
matrix

Kalman gain matrix for the measurement update, returned as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs.

A,Cm,Bu,Bv,Dvm — System matrices
matrices

System matrices used to calculate the estimator gains, returned as matrices of various dimensions. For definitions of these system matrices, see "State Estimator Equations" on page 2-71.

model — System used for state-estimator design
state-space model

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

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

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

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

index — Locations of variables within model
structure

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

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

Algorithms

State Estimator Equations

In general, the controller states are unmeasured and must be estimated. By default, the controller uses a steady-state Kalman filter that derives from the state observer. For more information, see "Controller State Estimation".

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

1 Obtain the following data:

- $x_c(k|k-1)$ — Controller state estimate from previous control interval, k–1
- $u^{act}(k-1)$ — Manipulated variable (MV) actually used in the plant from k–1 to k (assumed constant)
- $u^{opt}(k-1)$ — Optimal MV recommended by MPC and assumed to be used in the plant from k–1 to k
- $v(k)$ — Current measured disturbances
- $y_m(k)$ — Current measured plant outputs
- B_u, B_v — Columns of observer parameter B corresponding to $u(k)$ and $v(k)$ inputs
- C_m — Rows of observer parameter C corresponding to measured plant outputs
- D_{mv} — Rows and columns of observer parameter D corresponding to measured plant outputs and measured disturbance inputs
- L, M — Constant Kalman gain matrices

Plant input and output signals are scaled to be dimensionless prior to use in calculations.

2 Revise $x_c(k|k-1)$ when $u^{act}(k-1)$ and $u^{opt}(k-1)$ are different.

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

3 Compute the innovation.

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

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

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

Then, the software uses the current state estimate $x_c(k|k)$ to solve the quadratic program at interval k. The solution is $u^{opt}(k)$, the MPC-recommended manipulated-variable value to be used between control intervals k and $k+1$.

Finally, the software prepares for the next control interval assuming that the unknown inputs, $w_{id}(k)$, $w_{od}(k)$, and $w_n(k)$ assume their mean value (zero) between times k and $k+1$. The software predicts the impact of the known inputs and the innovation as follows:

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

Version History
Introduced in R2014b

See Also

Functions
setEstimator | kalman | getindist | getoutdist | get | set | sim

Objects
mpc | mpcstate | mpcsimopt | mpcmoveopt

Blocks
MPC Controller | Adaptive MPC Controller | Multiple MPC Controllers

Topics
"Custom State Estimation"
"Implement Custom State Estimator Equivalent to Built-In Kalman Filter"
"Controller State Estimation"
"MPC Prediction Models"

getindist

Retrieve unmeasured input disturbance model

Syntax

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

Description

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

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

Examples

Retrieve Input Disturbance Model

Define a plant model with no direct feedthrough.

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

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

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

Create an MPC controller for the defined plant.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Extract the input disturbance model.

```
indist = getindist(mpcobj);
```

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

Retrieve Input Disturbance Model Channels with Default Integrated White Noise

Define a plant model with no direct feedthrough.

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

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

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

Create an MPC controller for the defined plant.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Extract the default output disturbance model.

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

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property is empty:
   Assuming unmeasured input disturbance #2 is integrated white noise.
   Assuming unmeasured input disturbance #3 is white noise.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Check which input disturbance channels have integrated white noise added by default.

```
channels
```

```
channels = 1
```

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

Input Arguments

mpcobj — Model predictive controller
mpc object

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

Output Arguments

indist — Input disturbance model
discrete-time, delay-free, state-space model

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

The input disturbance model has:

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

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

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

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

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

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

Tips

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

Version History
Introduced in R2006a

See Also

Functions
getoutdist | setindist | setoutdist | getEstimator | setEstimator | get | set | review | sim

Objects
mpc | mpcsimopt

getname

Retrieve I/O signal names from MPC plant 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 of the plant model in `mpcobj`. This is equivalent to `name = mpcobj.Model.Plant.InputName{i}`.

`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}`.

Examples

Get names of input and output signals from MPC object

Create a plant and an MPC object, and then retrieve the names of some input and output signals.

```
mpcverbosity off;                               % turn off mpc messages

% create plant model
plant = rss(4,4,4);                             % random state space
plant.D = 0;                                    % set D matrix to zero

% set signals type in plant model
plant = setmpcsignals(plant,'MV',1,'MD',3,'UD',4,'MO',1,'UO',[3 4]);

% create MPC object
mpcobj=mpc(plant,1);                            % sampling time = 1 second
```

Get names of input signals

```
% get input signal names
getname(mpcobj,'input',1)                       % get name of first input signal
ans =
    'MV1'

getname(mpcobj,'input',2)                       % get name of second input signal
ans =
    'MV2'

getname(mpcobj,'input',3)                       % get name of third input signal
ans =
    'MD1'

getname(mpcobj,'input',4)                       % get name of fourth input signal
```

```
ans =

    'UD1'
```

Get names of output signals

```
% get output signal names
getname(mpcobj,'output',1)                      % get name of first output signal
ans =

    'MO1'

getname(mpcobj,'output',2)                      % get name of second output signal
ans =

    'MO2'

getname(mpcobj,'output',3)                      % get name of third output signal
ans =

    'UO1'

getname(mpcobj,'output',4)                      % get name of fourth output signal
ans =

    'UO2'

% alternative ways to retrieve names
mpcobj.Model.Plant.InputName{2}                 % second plant input
ans =

    'MV2'

mpcobj.ManipulatedVariables(2).Name             % second manipulated variable
ans =

    'MV2'

mpcobj.Model.Plant.InputName{4}                 % fourth plant input
ans =

    'UD1'

mpcobj.DisturbanceVariables(2).Name             % second disturbance variable
ans =

    'UD1'

mpcobj.Model.Plant.OutputName{4}                % fourth plant output
ans =

    'UO2'

mpcobj.OutputVariables(4).Name                  % fourth plant variable name
ans =

    'UO2'
```

Note that signals not specified with `setmpcsignals` are assumed to be measured inputs (for non-specified inputs) or measured outputs (for non-specified outputs).

Input Arguments

mpcobj — Model predictive controller
mpc object

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

i — Signal number selection
`'integer'` greater than zero

This integer specify that the name of the `ith` signal needs to be retrieved.

Signal number to be retrieved.

Example: 2

Output Arguments

name — Signal name
character array

This character array is the name of the `ith` input or output signal (and it does not affect whether the signal is categorized as a manipulated variable, measured or unmeasured disturbance, measured or unmeasured output).

For input signals, this is the content of `mpcobj.Model.Plant.InputName{i}`, while for output signals, this is the content of `mpcobj.Model.Plant.OutputName{i}`.

If the specified signal is a manipulated variable, this field is typically `'MV1'`, `'MV2'`, and so on, up to the number of manipulated variables, unless specifically set otherwise. This is also identical to the content of the `Name` field of the corresponding structure in `mpcobj.ManipulatedVariables`.

If the specified signal is a disturbance input, this field is typically `'MD1'`, `'MD2'`, and so on, up to the number of measured disturbance variables, or `'UD1'`, `'UD2'`, and so on, up to the number of unmeasured disturbance variables, unless specifically set otherwise. This is also the content of the corresponding `Name` field of `mpcobj.DisturbanceVariables`.

If the specified signal is a output signal, this field is typically `'MO1'`, `'MO2'`, and so on, up to the number of measured output variables, or `'UO1'`, `'UO2'`, and so on, up to the number of unmeasured output variables, unless specifically set otherwise. This is also the content of the corresponding `Name` field of `mpcobj.OutputVariables`.

Version History
Introduced before R2006a

See Also

Functions
setname | setmpcsignals | get | set | mpcverbosity | review

Objects
mpc

Blocks
MPC Controller

getoutdist

Retrieve unmeasured output disturbance model

Syntax

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

Description

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

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

Examples

Retrieve Output Disturbance Model

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

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Extract the output disturbance model.

```
outdist = getoutdist(mpcobj);
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Retrieve Output Disturbance Model Channels with Default Integrated White Noise

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

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Extract the default output disturbance model.

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

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->Assuming output disturbance added to measured output #3 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Check which channels have default integrated white noise disturbances.

```
channels
```

```
channels = 1×3

     1     2     3
```

Integrators have been added to all three output channels.

Input Arguments

mpcobj — Model predictive controller
mpc object

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

Output Arguments

outdist — Output disturbance model
discrete-time, delay-free, state-space model

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

The output disturbance model has:

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

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

channels — Output channels with integrated white noise
vector of output indices

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

Tips

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

Version History
Introduced before R2006a

See Also

Functions
setoutdist | getindist | setindist | getEstimator | setEstimator | get | set | review | sim

Objects
mpc | mpcsimopt

getSimulationData

Create data structure to simulate multistage MPC controller with `nlmpcmove`

Syntax

```
simdata = getSimulationData(nlmpcMSobj)
```

Description

Use this function to create a default data structure to simulate a multistage MPC controller with the `nlmpcmove` function.

For information on generating data structures for `mpcmoveCodeGeneration`, see `getCodeGenerationData`.

`simdata = getSimulationData(nlmpcMSobj)` creates an initial simulation data structure for use with `nlmpcmove`.

Examples

Simulate Multistage Nonlinear MPC Controller Using Initial Guesses

This example shows how to create and simulate a simple multistage MPC controller in closed loop using initial guesses, with the MATLAB® function `nlmpcmove`.

Create Multistage MPC Controller

Create a multistage MPC object with a seven-steps horizon, one state, and one manipulated variable.

```
msobj = nlmpcMultistage(7,1,1);
```

Defining your state and cost functions as separate files in the current folder on in a folder on the MATLAB path is recommended, as local functions are not supported for the generation of C/C++ deployment code. However, for this example, the state, cost, and state Jacobian functions are defined as local functions at the end of the example.

Specify the state transition function for the prediction model.

```
msobj.Model.StateFcn = @mystatefcn;
```

As a best practice, use Jacobians whenever they are available, otherwise the solver must compute it numerically. You can also automatically generate a Jacobian function using `generateJacobianFunction`.

Specify the Jacobian of the state transition function.

```
msobj.Model.StateJacFcn = @mystatejac;
```

Specify the cost functions for all stages except the first

```
for i=2:8
    msobj.Stages(i).CostFcn = @mycostfcn;
end
```

Define Initial Conditions, Create Data Structure, and Validate Functions

Initialize the plant state and input.

```
x=3;
mv=0;
```

Create the initial simulation data structure.

```
simdata = getSimulationData(msobj)
```

```
simdata = struct with fields:
    InitialGuess: []
```

Validate functions and the data structure.

```
validateFcns(msobj,x,mv,simdata);
```

```
Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [2 3 4 5 6 7 8] are OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Simulate Controller in Closed Loop

Simulate the control loop for 5 steps.

```
for k=1:5
    [mv,simdata] = nlmpcmove(msobj, x, mv, simdata);                % calculate move and update
    [~,xhist] = ode45(@(t,xode) mystatefcn(xode,mv),[0 msobj.Ts],x);  % simulate plant for one sa
    x = xhist(end);                                                  % update plant state
end
```

Since updated initial guesses are supplied as an input argument within the `simdata` structure, `nlmpcmove` does not need to recalculate them at each time step, which saves computation time and improves performance. Updating initial guesses at every time step is a best practice.

Display the last values of the state and manipulated variables.

```
disp(['Final value of x =' num2str(x)])
```

```
Final value of x =-0.039545
```

```
disp(['Final value of mv =' num2str(mv)])
```

```
Final value of mv =-0.066672
```

Support Functions

State transition function.

```
function xdot = mystatefcn(x,u)
    xdot = u-sin(x);
end
```

Jacobian of the state transition function.

```
function [A,B] = mystatejac(x,~)
    A = -cos(x);
    B = 1;
end
```

Stage cost functions.

```
function j = mycostfcn(s,x,u)
    j = abs(u)/s+s*x^2;
end
```

Input Arguments

`nlmpcMSobj` — Nonlinear Multistage MPC controller
`nlmpcMultistage` object

Multistage nonlinear MPC controller, specified as an `nlmpcMultistage` object.

Output Arguments

`simdata` — Run-time simulation data structure
structure

Run-time simulation data, specified as a structure with the following fields.

`MeasuredDisturbance` — Measured disturbance values
`[]` (default) | row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your multistage MPC object has any measured disturbance channel defined, you must specify `MeasuredDisturbance`. If your controller has no measured disturbances, you can omit this field in the structure or specify it as `[]`.

To use the same disturbance values across the prediction horizon, specify a row vector.

To vary the disturbance values over the prediction horizon from time k to time $k+p$, specify an array with up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbance values for one prediction horizon step. If you specify fewer than p rows, `nlmpcmove` uses the values in the final row for the remaining steps of the prediction horizon.

If you define measured disturbances in the input object, you must provide them via `simdata` at run-time.

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

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

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$–1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

If `simdata` does not contain a `MVMin` field, then the manipulated variable lower bound (if present in the input object) does not change at run time.

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

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

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

If `simdata` does not contain a `MVMax` field, then the manipulated variable upper bound (if present in the input object) does not change at run time.

MVRateMin — Manipulated variable rate lower bounds
[ ] (default) | row vector | matrix

Manipulated variable rate lower bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMin(:,i)` replaces the `ManipulatedVariables(i).RateMin` property of the controller at run time. `MVRateMin` bounds must be nonpositive.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

If `simdata` does not contain a `MVRateMin` field, then the manipulated variable rate lower bound (if present in the input object) does not change at run time.

MVRateMax — Manipulated variable rate upper bounds
[ ] (default) | row vector | matrix

Manipulated variable rate upper bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMax(:,i)` replaces the `ManipulatedVariables(i).RateMax` property of the controller at run time. `MVRateMax` bounds must be nonnegative.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for

one prediction horizon step. If you specify fewer than *p* rows, the final bounds are used for the remaining steps of the prediction horizon.

If `simdata` does not contain a `MVRateMax` field, then the manipulated variable rate upper bound (if present in the input object) does not change at run time.

StateMin — State lower bounds
[ ] (default) | row vector | matrix

State lower bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMin(:,i)` replaces the `States(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to *p* rows. Here, *k* is the current time and *p* is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than *p* rows, the final bounds are used for the remaining steps of the prediction horizon.

If `simdata` does not contain a `StateMin` field, then the state lower bound (if present in the input object) does not change at run time.

StateMax — State upper bounds
[ ] (default) | row vector | matrix

State upper bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMax(:,i)` replaces the `States(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to *p* rows. Here, *k* is the current time and *p* is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than *p* rows, the final bounds are used for the remaining steps of the prediction horizon.

If `simdata` does not contain a `StateMax` field, then the state upper bound (if present in the input object) does not change at run time.

StateFcnParameters — State function parameter values
[ ] (default) | vector

State function parameter values, specified as a vector with length equal to the value of the `Model.ParameterLength` property of the multistage controller object. If `Model.StateFcn` needs a parameter vector, you must provide its value at runtime using this field, otherwise you can omit this field or set it to [ ].

StageFcnParameters — Stage function parameter values
[ ] (default) | vector

Stage functions parameter values, specified as a vector with length equal to the sum of all the values in the `Stages(i).ParameterLength` properties of the multistage controller object. If any cost or constraint function defined in the `Stages` property needs a parameter vector, you must provide all

the parameter vectors at runtime (stacked in a single column) using this field, otherwise you can omit this field or set it to `[]`.

You must stack the parameter vectors for all stages in the column vector `StageFcnParameters` as follows.

```
[parameter vector for stage 1;
 parameter vector for stage 2;
 ...
 parameter vector for stage p+1;
]
```

`TerminalState` — Terminal state
`[]` (default) | vector

Terminal state, specified as a column vector with as many elements as the number of states. The terminal state is the desired state at the last prediction step. To specify desired terminal states at run-time via this field, you must specify finite values in the `TerminalState` field of the `Model` property of `nlmpcMSobj`. Specify `inf` for the states that you do not need to constrain to a terminal value. At run time, `nlmpcmove` ignores any values in the `TerminalState` field of `simdata` that correspond to `inf` values in `nlmpcMSobj`. If you do not specify any terminal value condition in `nlmpcMSobj`, this field is not created in `simdata`.

If `simdata` does not contain a `TerminalState` field, then the terminal state constraint (if present in the input object) does not change at run time.

`InitialGuess` — Initial guesses for the decision variables
`[]` (default) | vector

Initial guesses for the decision variables, specified as a row vector of length equal to the sum of the lengths of all the decision variable vectors for each stages.

You must be stack the initial guesses for all stages in the column vector `InitialGuess` as follows.

```
[state vector guess for stage 1;
 manipulated variable vector guess for stage 1;
 manipulated variable vector rate guess for stage 1; % if used
 slack variable vector guess for stage 1; % if used
 state vector guess for stage 2;
 manipulated variable vector guess for stage 2;
 manipulated variable vector rate guess for stage 2; % if used
 slack variable vector guess for stage 2; % if used
 ...
 state vector guess for stage p+1;
 manipulated variable vector guess for stage p+1;
 manipulated variable vector rate guess for stage p+1; % if used
 slack variable vector guess for stage p+1; % if used
]
```

If `InitialGuess` is `[]`, then `nlmpcmove` calculates the initial guesses from its `x` and `lastmv` arguments.

In general, during closed-loop simulation, you do not specify `InitialGuess` yourself. Instead, when calling `nlmpcmove`, return the `simdata` output argument, which contains the calculated initial guesses for the next control interval. You can then pass `simdata` as an input argument to `nlmpcmove` for the next control interval. These steps are a best practice, even if you do not specify any other run-time options.

Version History
Introduced in R2021a

See Also

Functions
validateFcns | nlmpcmove | getCodeGenerationData | nlmpcmoveCodeGeneration

Objects
nlmpcMultistage | nlmpcmoveopt

Topics
"Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC"
"Nonlinear MPC"

gpc2mpc

Generate MPC controller using generalized predictive controller (GPC) settings

Syntax

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

Description

`mpcobj = gpc2mpc(plant)` generates a single-input single-output MPC controller with default GPC settings and sample time of the specified plant, `plant`. The GPC is a nonminimal state-space representation described in "References" on page 2-91. `plant` is a discrete-time LTI model with sample time greater than 0.

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

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

Examples

Design an MPC controller using GPC settings

```
% Specify the plant described in Example 1.8  of
% "References".
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
% "References".
% 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);
```

Input Arguments

`plant` — plant model
single-output discrete-time `ss`, `tf` or `zpk` object

Single-output LTI model with sampling time greater than 0, and only one manipulated variable input.

Example: `zpk([],-1,1)`

`gpcOptions` — GPC settings
structure

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

Output Arguments

`mpcobj` — Model predictive controller
`mpc` object

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

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.

Version History
Introduced in R2010a

References

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

See Also

Functions
setmpcsignals | mpcprops | review | cloffset | sim

Objects
mpc | mpcsimopt

Blocks
MPC Controller

Topics
"Design Controller Using MPC Designer"
"Design MPC Controller at the Command Line"

mpcActiveSetOptions

Create default option set for `mpcActiveSetSolver`

Syntax

```
options = mpcActiveSetOptions
options = mpcActiveSetOptions(type)
```

Description

`options = mpcActiveSetOptions` creates a structure of default options for `mpcActiveSetSolver`, which solves a quadratic programming (QP) problem using an active-set algorithm.

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

Examples

Create Default Option Set for Active-Set QP Solver

Create a default option set.

```
opt = mpcActiveSetOptions;
```

Create and Modify Default Active-Set QP Solver Option Set

Create a default option set.

```
opt = mpcActiveSetOptions;
```

Specify the maximum number of iterations allowed during computation.

```
opt.MaxIterations = 100;
```

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

```
opt.ConstraintTolerance = 1.0e-4;
```

Create Active-Set Option Set Specifying Input Argument Type

Create a default option set, specifying the input argument type.

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

Input Arguments

type — Solver input argument data type
'double' (default) | 'single'

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

Output Arguments

options — Option set for `mpcActiveSetSolver`
structure

Option set for `mpcActiveSetSolver`, returned as a structure with the following fields.

Field	Description	Default
DataType	Input argument data type, specified as either `'double'` or `'single'`. This data type is used for both simulation and code generation, and all real input arguments to the solver function must match this type.	'double'
MaxIterations	Maximum number of iterations allowed when computing the QP solution, specified as a positive integer.	200
ConstraintTolerance	Tolerance used to verify that inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger `ConstraintTolerance` value allows for larger constraint violations.	1e-6
UseHessianAsInput	Indicator of whether the first input argument to mpcActiveSetSolver is the Hessian matrix, specified as a logical value. If `UseHessianAsInput` is `true`, pass the Hessian matrix to `mpcActiveSetSolver`. Otherwise, use the inverse of the lower-triangular Cholesky decomposition (`Linv`) of the Hessian matrix. If your application requires repetitive calls of `mpcActiveSetSolver` using a constant Hessian matrix, you can improve computational efficiency by passing `Linv` to `mpcActiveSetSolver` instead of the Hessian matrix.	true
IntegrityChecks	Indicator of whether integrity checks are performed on the solver function input data, specified as a logical value. If `IntegrityChecks` is `true`, then integrity checks are performed and diagnostic messages are displayed. Use `false` for code generation only.	true

Version History
Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use `mpcActiveSetSolver` as a general-purpose QP solver that supports code generation. To specify solver options, use `mpcActiveSetOptions`. Create the function `myCode`, which uses `mpcActiveSetSolver` and `mpcActiveSetOptions`.

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

  Generate C code with MATLAB Coder™.

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

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

See Also

Functions
mpcActiveSetSolver | mpcqpsolver | mpcqpsolverOptions | mpcInteriorPointSolver | mpcInteriorPointOptions | setCustomSolver | quadprog

Objects
mpc

Topics
"Solve Custom MPC Quadratic Programming Problem and Generate Code"
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

mpcActiveSetSolver

Solve quadratic programming problem using active-set algorithm

Syntax

```
[x,exitflag] = mpcActiveSetSolver(H,f,A,b,Aeq,beq,iA0,options)
[x,exitflag,iA,lambda] = mpcActiveSetSolver(H,f,A,b,Aeq,beq,iA0,options)
```

Description

Using `mpcActiveSetSolver`, you can solve a quadratic programming (QP) problem using an active-set algorithm. This function provides access to the built-in Model Predictive Control Toolbox active-set QP solver.

Using an active-set solver can provide fast and robust performance for small-scale and medium-scale optimization problems in both double and single precision.

This solver is useful for:

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

Alternatively, you can also access the built-in interior-point QP solver using `mpcInteriorPointSolver`.

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

$$J = \frac{1}{2}x^\mathsf{T}Hx + f^\mathsf{T}x$$

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

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

Examples

Solve Quadratic Programming Problem Using Active-Set Solver

Find the values of x that minimize

$$f(x) = 0.5x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2,$$

subject to the constraints

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

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

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

Specify the inequality constraint parameters.

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

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

```
n = length(f);
Aeq = zeros(0,n);
beq = zeros(0,1);
```

Create a default option set for mpcActiveSetSolver.

```
opt = mpcActiveSetOptions;
```

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

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

Solve the QP problem.

```
[x,exitflag] = mpcActiveSetSolver(H,f,A,b,Aeq,beq,iA0,opt);
```

Examine the solution x.

```
x
```

x = 2×1

```
    0.6667
    1.3333
```

When solving the QP problem, you can also determine which inequality constraints are active for the solution.

```
[x,exitflag,iA,lambda] = mpcActiveSetSolver(H,f,A,b,Aeq,beq,iA0,opt);
```

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

```
iA
```

iA = 5x1 logical array

```
   0
```

```
0
1
1
0
```

There is a single active inequality constraint. View the Lagrange multiplier for this constraint.

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

```
ans = 0
```

Input Arguments

H — Hessian matrix
n-by-*n* matrix

Hessian matrix, specified as a symmetric *n*-by-*n* matrix, where $n > 0$ is the number of optimization variables.

The active-set QP algorithm requires that the Hessian matrix be positive definite. To determine whether H is positive definite, use the `chol` function.

```
[~,p] = chol(H);
```

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

The active-set QP algorithm computes the lower-triangular Cholesky decomposition (`Linv`) of the Hessian matrix. If your application requires repetitive calls of `mpcActiveSetSolver` using a constant Hessian matrix, you can improve computational efficiency by computing `Linv` once and passing it to `mpcActiveSetSolver` instead of the Hessian matrix. To do so, you must set the `UseHessianAsInput` field of `options` to `false`.

```
options = mpcActiveSetOptions;
options.UseHessianAsInput = false;
```

To compute `Linv`, use the following code.

```
[L,p] = chol(H,'lower');
Linv = linsolve(L,eye(size(L)),struct('LT',true));
```

f — Multiplier of the objective function linear term
column vector of length *n*

Multiplier of the objective function linear term, specified as a column vector of length *n*, where *n* is the number of optimization variables.

A — Linear inequality constraint coefficients
m-by-*n* matrix

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

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

b — Right-hand side of inequality constraints
column vector of length *m*

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

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

Aeq — Linear equality constraint coefficients
q-by-*n* matrix

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

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

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

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

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

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

Initial active inequalities, where the equal portion of the inequality is true, specified as a logical vector of length *m*, where *m* is the number of inequality constraints. Specify iA0 as follows:

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

options — Option set for `mpcActiveSetSolver`
structure

Option set for `mpcActiveSetSolver`, specified as a structure created using `mpcActiveSetOptions`.

Output Arguments

x — Optimal solution to the QP problem
column vector of length *n*

Optimal solution to the QP problem, returned as a column vector of length *n*, where *n* is the number of optimization variables. `mpcActiveSetSolver` always returns a value for x. To determine whether the solution is optimal or feasible, check `exitflag`.

exitflag — Solution validity indicator
positive integer | 0 | -1 | -2

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

Value	Description
> 0	x is optimal. In this case, `exitflag` represents the number of iterations performed during optimization.
0	The maximum number of iterations was reached. Solution x might be suboptimal or infeasible. To determine if x is infeasible, check whether the solution violates the constraint tolerance specified in `options`. `feasible = (A*x-b) <= options.ConstraintTolerance;` If any element of `feasible` is `false`, then x is infeasible.
-1	The problem appears to be infeasible, that is, the constraint $Ax \leq b$ cannot be satisfied.
-2	An unrecoverable numerical error occurred.

iA — Active inequalities
logical vector of length m

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

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

lambda — Lagrange multipliers
structure

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

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

Tips

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

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

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

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

Algorithms

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

Version History

Introduced in R2020a

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use mpcActiveSetSolver as a general-purpose QP solver that supports code generation. To specify solver options, use mpcActiveSetOptions. Create the function myCode, which uses mpcActiveSetSolver and mpcActiveSetOptions.

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

  Generate C code with MATLAB Coder.

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

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

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
mpcActiveSetOptions | mpcqpsolver | mpcqpsolverOptions | mpcInteriorPointSolver | mpcInteriorPointOptions | setCustomSolver | quadprog

Objects
mpc

Topics
"Solve Custom MPC Quadratic Programming Problem and Generate Code"
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

mpcInteriorPointOptions

Create default option set for `mpcInteriorPointSolver`

Syntax

```
options = mpcInteriorPointOptions
options = mpcInteriorPointOptions(type)
```

Description

`options = mpcInteriorPointOptions` creates a structure of default options for `mpcInteriorPointSolver`, which solves a quadratic programming (QP) problem using an interior-point algorithm.

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

Examples

Create Default Option Set for Interior-Point QP Solver

Create a default option set.

```
opt = mpcInteriorPointOptions;
```

Create and Modify Default Interior-Point QP Solver Option Set

Create a default option set.

```
opt = mpcInteriorPointOptions;
```

Specify the maximum number of iterations allowed during computation.

```
opt.MaxIterations = 100;
```

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

```
opt.ConstraintTolerance = 1.0e-4;
```

Create Interior-Point Option Set Specifying Input Argument Type

Create a default option set, specifying the input argument type.

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

Input Arguments

type — Solver input argument data type
'double' (default) | 'single'

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

Output Arguments

options — Option set for `mpcInteriorPointSolver`
structure

Option set for `mpcInteriorPointSolver`, returned as a structure with the following fields.

Field	Description	Default
DataType	Input argument data type, specified as either `'double'` or `'single'`. This data type is used for both simulation and code generation, and all real input arguments to the solver function must match this type.	'double'
MaxIterations	Maximum number of iterations allowed when computing the QP solution, specified as a positive integer.	50
ConstraintTolerance	Tolerance used to verify that equality and inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger `ConstraintTolerance` value allows for larger constraint violations.	1e-6
OptimalityTolerance	Termination tolerance for first-order optimality (KKT dual residual), specified as a positive scalar. Increasing this value relaxes the condition for the optimality check.	1e-6
ComplementarityTolerance	Termination tolerance for first-order optimality (KKT average complementarity residual), specified as a positive scalar. Increasing this value improves robustness, while decreasing this value increases accuracy.	1e-8
StepTolerance	Termination tolerance for decision variables, specified as a positive scalar.	1e-8
IntegrityChecks	Indicator of whether integrity checks are performed on the solver function input data, specified as a logical value. If `IntegrityChecks` is `true`, then integrity checks are performed and diagnostic messages are displayed. Use `false` for code generation only.	true

Version History
Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use `mpcInteriorPointSolver` as a general-purpose QP solver that supports code generation. To specify solver options, use `mpcInteriorPointOptions`. Create the function `myCode`, which uses `mpcInteriorPointSolver` and `mpcInteriorPointOptions`.

```
function [out1,out2] = myCode(in1,in2)
%#codegen
...
options = mpcInteriorPointOptions;
[x,exitflag] = mpcInteriorPointSolver(H,f,A,b,Aeq,Beq,x0,options);
...
```

  Generate C code with MATLAB Coder.

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

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

See Also

Functions
mpcInteriorPointSolver | mpcqpsolver | mpcqpsolverOptions | mpcActiveSetSolver | mpcActiveSetOptions | setCustomSolver | quadprog

Objects
mpc

Topics
"Solve Custom MPC Quadratic Programming Problem and Generate Code"
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

mpcInteriorPointSolver

Solve a quadratic programming problem using an interior-point algorithm

Syntax

```
[x,exitflag] = mpcInteriorPointSolver(H,f,A,b,Aeq,beq,x0,options)
[x,exitflag,feasible,lambda] = mpcInteriorPointSolver(H,f,A,b,Aeq,beq,x0,
options)
```

Description

Using `mpcInteriorPointSolver`, you can solve a quadratic programming (QP) problem using a primal-dual interior-point algorithm with a Mehrotra predictor-corrector. This function provides access to the built-in Model Predictive Control Toolbox interior-point QP solver.

Using an interior-point solver can provide superior performance for large-scale optimization problems, such as MPC applications that enforce constraints over large prediction and control horizons.

This solver is useful for:

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

Alternatively, you can also access the built-in active-set QP solver using `mpcActiveSetSolver`.

`[x,exitflag] = mpcInteriorPointSolver(H,f,A,b,Aeq,beq,x0,options)` finds an optimal solution x to a quadratic programming problem by minimizing the objective function

$$J = \frac{1}{2}x^{\mathsf{T}}Hx + f^{\mathsf{T}}x$$

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

`[x,exitflag,feasible,lambda] = mpcInteriorPointSolver(H,f,A,b,Aeq,beq,x0,options)` also returns a logical flag `feasible` that indicates the feasibility of the solution and the Lagrange multipliers `lambda` for the solution.

Examples

Solve Quadratic Programming Problem Using Interior-Point Solver

Find the values of x that minimize

$$f(x) = 0.5x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2,$$

subject to the constraints

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

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

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

Specify the inequality constraint parameters.

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

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

```
n = length(f);
Aeq = zeros(0,n);
beq = zeros(0,1);
```

As a best practice, verify that `H` is positive definite using the `chol` function.

```
[~,p] = chol(H);
```

If `p = 0`, then `H` is positive definite.

```
p
```

```
p = 0
```

Create a default option set for `mpcInteriorPointSolver`.

```
opt = mpcInteriorPointOptions;
```

To cold start the solver, specify an initial guess of zeros for the elements of `x`.

```
x0 = zeros(n,1);
```

Solve the QP problem.

```
[x,exitflag] = mpcInteriorPointSolver(H,f,A,b,Aeq,beq,x0,opt);
```

Examine the solution `x`.

```
x
```

```
x = 2×1

    0.6667
    1.3333
```

Input Arguments

H — Hessian matrix
n-by-*n* matrix

Hessian matrix, specified as an *n*-by-*n* matrix, where $n > 0$ is the number of optimization variables.

The interior-point QP algorithm requires that the Hessian matrix be positive definite. To determine whether H is positive definite, use the `chol` function.

```
[~,p] = chol(H);
```

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

f — Multiplier of the objective function linear term
column vector of length *n*

Multiplier of the objective function linear term, specified as a column vector of length *n*, where *n* is the number of optimization variables.

A — Linear inequality constraint coefficients
m-by-*n* matrix | [ ]

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

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

b — Right-hand side of inequality constraints
column vector of length *m*

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

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

Aeq — Linear equality constraint coefficients
q-by-*n* matrix | [ ]

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

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

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

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

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

x0 — Initial guess
column vector of length *n*

Initial guess for the solution, where the equal portion of the inequality is true, specified as a column vector of length *n*, where *n* is the number of optimization variables. For a *cold start*, specify the initial guess as `zeros(n,1)`.

options — Option set for `mpcInteriorPointSolver`
structure

Option set for `mpcInteriorPointSolver`, specified as a structure created using `mpcInteriorPointOptions`.

Output Arguments

x — Optimal solution to the QP problem
column vector of length *n*

Optimal solution to the QP problem, returned as a column vector of length *n*, where *n* is the number of optimization variables. `mpcInteriorPointSolver` always returns a value for x. To determine whether the solution is optimal or feasible, check `exitflag` and `feasible`.

`exitflag` — Solution validity indicator
positive integer | 0 | -1

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

Value	Description
> 0	x is optimal. `exitflag` represents the number of iterations performed during optimization.
0	The maximum number of iterations was reached before the solver could find an optimal solution. Solution x is feasible only if `feasible` is `true`.
-1	The problem appears to be infeasible; that is, the constraint $Ax \leq b$ cannot be satisfied.

`feasible` — Solution feasibility
logical scalar

Solution feasibility, returned as a logical scalar. When `exitflag` is 0, the solver reached the maximum number of iterations without finding an optimal solution. This suboptimal solution, returned in x, is feasible only if `feasible` is true.

`lambda` — Lagrange multipliers
structure

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

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

Tips

- To determine whether *H* is positive definite, use the `chol` function.

  ```
  [~,p] = chol(H);
  ```

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

- `mpcInteriorPointSolver` provides access to the interior-point QP solver used by Model Predictive Control Toolbox software. Use this command to solve QP problems in your own custom MPC applications. For an example of a custom MPC application, see "Solve Custom MPC Quadratic Programming Problem and Generate Code". This example uses `mpcActiveSetSolver`, however, the workflow applies to `mpcInteriorPointSolver` as well.

Algorithms

`mpcInteriorPointSolver` solves the QP problem using an interior-point method. For more information, see "QP Solvers".

Version History
Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use `mpcInteriorPointSolver` as a general-purpose QP solver that supports code generation. To specify solver options, use `mpcInteriorPointOptions`. Create the function `myCode`, which uses `mpcInteriorPointSolver` and `mpcInteriorPointOptions`.

  ```
  function [out1,out2] = myCode(in1,in2)
  %#codegen
  ...
  options = mpcInteriorPointOptions;
  [x,exitflag] = mpcInteriorPointSolver(H,f,A,b,Aeq,Beq,x0,options);
  ...
  ```

  Generate C code with MATLAB Coder.

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

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

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
mpcInteriorPointOptions | mpcqpsolver | mpcqpsolverOptions | mpcActiveSetSolver | mpcActiveSetOptions | setCustomSolver | quadprog

Objects
mpc

Topics
"Solve Custom MPC Quadratic Programming Problem and Generate Code"
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

mpcmove

Compute optimal control action and update controller states

Syntax

```
mv = mpcmove(mpcobj,xc,ym,r,v)
[mv,info] = mpcmove(mpcobj,xc,ym,r,v)
[ ___ ] = mpcmove( ___ ,options)
```

Description

Use this command to simulate an MPC controller in closed-loop with a plant model. Call `mpcmove` repeatedly in a for loop to calculate the manipulated variable and update the controller states at each time step.

`mv = mpcmove(mpcobj,xc,ym,r,v)` returns the optimal move `mv` and updates the states *xc* of the controller `mpcobj`.

The manipulated variable `mv` at the current time is calculated given:

- the controller object, `mpcobj`,
- a pointer to the current estimated extended state, `xc`,
- the measured plant outputs, `ym`,
- the output references, `r`,
- and the measured disturbance input, `v`.

If `ym`, `r` or `v` is specified as `[]`, or if it is missing as a last input argument, `mpcmove` uses the appropriate `mpcobj.Model.Nominal` value instead.

When using default state estimation, `mpcmove` also updates the controller state referenced by the handle object `xc`. Therefore, when using default state estimation, `xc` always points to the updated controller state. When using custom state estimation, you should update `xc` prior to each `mpcmove` call.

`[mv,info] = mpcmove(mpcobj,xc,ym,r,v)` returns additional information about the optimization problem solved to calculate `mv`.

`[ ___ ] = mpcmove( ___ ,options)` overrides default constraints and weights in `mpcobj` with the values specified in `Options`, an `mpcmoveopt` object. Use `Options` to provide run-time adjustment of constraints and weights during the closed-loop simulation.

Examples

Simulate Closed-Loop Response Using `mpcMove`

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);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

Obtain an handle object pointing to the controller state.

```
xc = mpcstate(mpcobj)
```

```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
MPCSTATE object with fields
          Plant: 0
    Disturbance: 0
          Noise: [1x0 double]
       LastMove: 0
     Covariance: [2x2 double]
```

The controller has one state for the internal plant model, one for the disturbance model, and one to hold the last value of the manipulated variable. All these three states are initialized to zero.

Set the reference signal. There is no measured disturbance.

```
r = 1;
```

Simulate the closed-loop response by calling `mpcmove` iteratively. In the simulation, assume that the simulated plant is identical to the predictive model. Therefore the plant state x in this case is identical to `xc.Plant` and the plant output is $y = C*x + D*u = 0.25*x = 0.25*xc.Plant$. Here, `mpcmove` updates the controller state referenced by `xc` (therefore including `xc.Plant`), and returns the manipulated variable in `u(i)`, which is used just for plotting.

```
t = 0:ts:40;
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
for i = 1:N
    y(i) = 0.25*xc.Plant;
    u(i) = mpcmove(mpcobj,xc,y(i),r);
end
```

Analyze the result.

```
[ts,us] = stairs(t,u);
plot(ts,us,'r-',t,y,'b--')
legend('MV','OV')
```
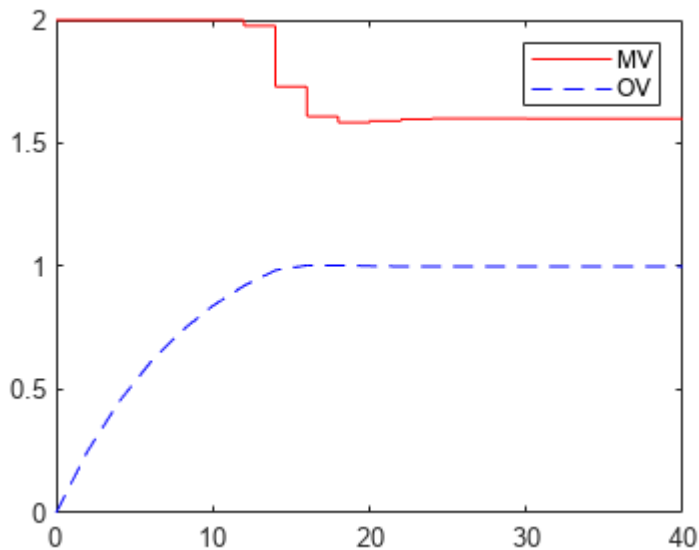
Modify the MV upper bound as the simulation proceeds using an `mpcmoveopt` object. Since the options argument overrides selected `mpcobj` properties, specify MV constraints again.

```
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).

```
xc = mpcstate(mpcobj);
y = zeros(N,1);
u = zeros(N,1);
for i = 1:N
    y(i) = 0.25*xc.Plant;
    if i == 5
        MPCopt.MVMax = 1;
    end
    u(i) = mpcmove(mpcobj,xc,y(i),r,[],MPCopt);
end
```

Analyze the results.

```
[ts,us] = stairs(t,u);
plot(ts,us,'r-',t,y,'b--')
legend('MV','OV')
```
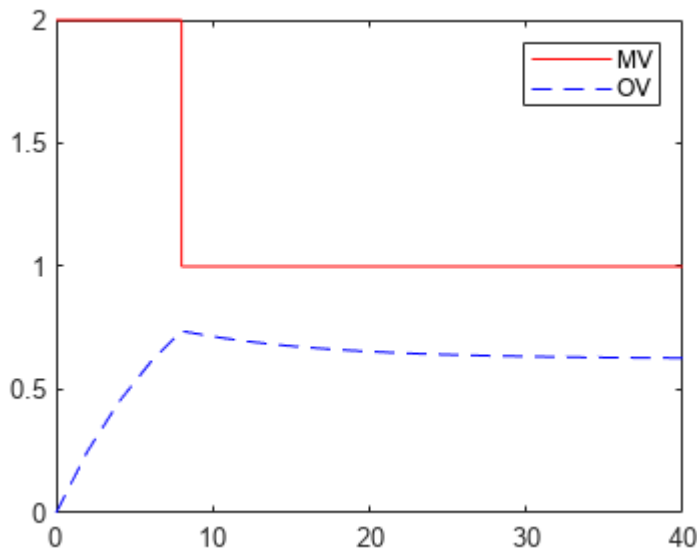
Evaluate Scenario at Specific Time Instant

Define a plant model.

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

Create a model predictive controller with constraints on both the manipulated variable and the rate of change of the manipulated variable. The prediction horizon is 10 intervals, and the control horizon is blocked.

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

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

Initialize (and return an handle to) the controller internal state for simulation.

```
xc = mpcstate(mpcobj);
```

```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

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

Compute the optimal control move at the current time.

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

Analyze the predicted optimal sequences.

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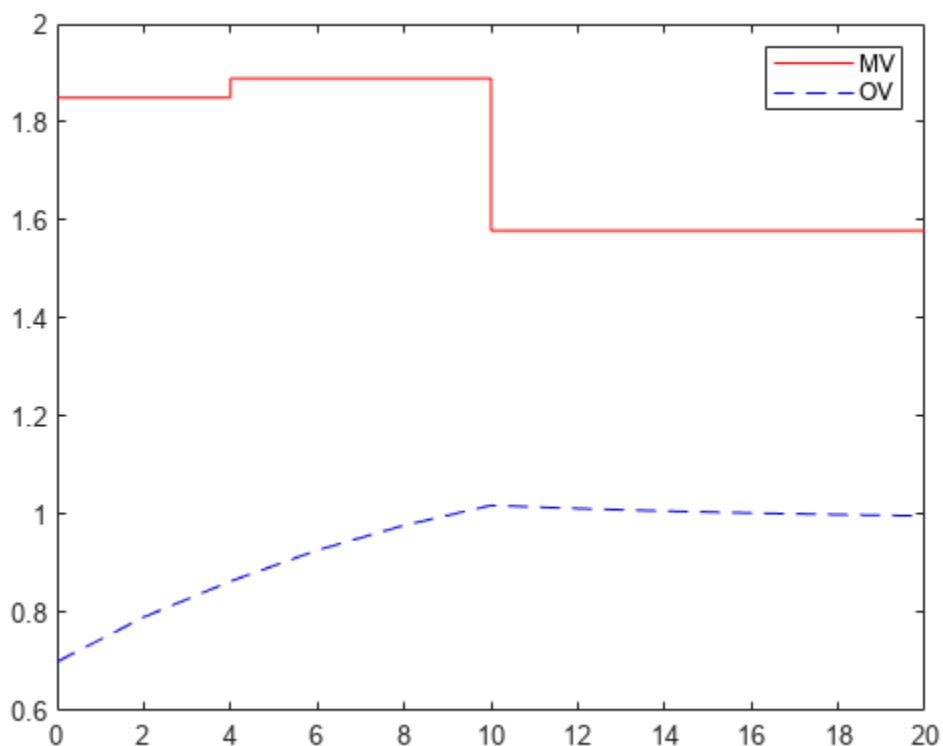

`plot` ignores `Info.Uopt(end)` as it is `NaN`.

Examine the optimal cost.

`Info.Cost`

`ans = 0.0793`

Input Arguments

`mpcobj` — Model predictive controller
`mpc object`

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

xc — Handle object pointing to current controller state
mpcstate handle object

Handle object pointing to current controller state, specified as an mpcstate object.

Note Since xc is an handle object, if you copy it to a new variable, the new variable still points to the current state of the same mpc object. If this state changes, this is reflected in both xc and its copy. To preserve the original state for later use, you can save xc to a MAT-file. During simulation, mpcmove also updates xc when a new control move is calculated. For more information about handle objects, see "Handle Object Behavior".

Before you begin a simulation with mpcmove, initialize the controller, and return an handle to its state using xc = mpcstate(mpcobj). Then, modify the default properties of xc as appropriate. mpcmove modifies the controller state. The handle object xc always reflect the current (updated) state of the controller.

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

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

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

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

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

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

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

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

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

v — Current and anticipated measured disturbances
$(p+1)$-by-N_{md} array

Current and anticipated measured disturbances, specified as a $(p+1)$-by-N_{md} array, where p is the prediction horizon of mpcobj and N_{md} is the number of measured disturbances. The first row of v

specifies the current measured disturbance values. Row $v(\texttt{i+1},:)$ defines the anticipated disturbance values at step i of the prediction horizon.

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

If your model includes measured disturbances, v must contain at least one row. If v contains fewer than p+1 rows, mpcmove duplicates the last row to fill the (p+1)-by-N_{md} array. If you supply exactly one row, a constant measured disturbance applies for the entire prediction horizon.

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

options — Run-time options
mpcmoveopt object

Run-time options, specified as an mpcmoveopt object. Use options to override selected properties of mpcobj during simulation. These options apply to the current mpcmove time instant only. Using options yields the same result as redefining or modifying mpcobj before each call to mpcmove, but involves considerably less overhead. Using options is equivalent to using an MPC Controller Simulink block in combination with optional input signals that modify controller settings, such as MV and OV constraints.

Output Arguments

mv — Optimal manipulated variable moves
column vector

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

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

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

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

info — Solution details
structure

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

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

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

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

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

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

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

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

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

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

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

Topt — Time intervals
column vector of length p+1

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

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

Slack — Slack variable
nonnegative scalar

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

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

See "Optimization Problem" for more information.

Iterations — Number of solver iterations
positive integer | 0 | -1 | -2

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

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

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

Optimization solution status, returned as one of the following:

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

Cost — Objective function cost
nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. For more information, see "Optimization Problem".

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

Tips

- `mpcmove` updates xc, even though it is an input argument.
- If ym, r or v is specified as `[]`, or if it is missing as a last input argument, `mpcmove` uses the appropriate `mpcobj.Model.Nominal` value instead.
- To view the predicted optimal behavior for the entire prediction horizon, plot the appropriate sequences provided in `Info`.
- To determine the optimization status, check `Info.Iterations` and `Info.QPCode`.

Alternatives

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

Version History

Introduced before R2006a

See Also

Functions
review | cloffset | buildMEX | mpcmoveCodeGeneration | mpcmoveAdaptive | mpcmoveMultiple | mpcmoveExplicit | sim

Objects
mpc | mpcstate | mpcmoveopt | mpcsimopt | explicitMPC

Topics
"Improving Control Performance with Look-Ahead (Previewing)"
"Switching Controllers Based on Optimal Costs"
"Understanding Control Behavior by Examining Optimal Control Sequence"
"What is Model Predictive Control?"
"MPC Prediction Models"

mpcmoveAdaptive

Compute optimal control with prediction model updating

Syntax

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

Description

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

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

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

Input Arguments

`mpcobj` — MPC controller
MPC controller object

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

`x` — Current MPC controller state
`mpcstate` object

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

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

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

If you are using custom state estimation, `mpcmoveAdaptive` expects `x` to represent `x[n|n]`. Therefore, prior to each `mpcmoveAdaptive` command, you must set `x.Plant`, `x.Disturbance`, and

x.Noise to the best estimates of these states (using the latest measurements) at the current control interval.

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

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

Updated prediction model, specified as one of the following:

- A delay-free, discrete-time state-space (ss) model. This plant is the update to mpcobj.Model.Plant and it must:

  - Have the same sample time as the controller; that is, Plant.Ts must match mpcobj.Ts
  - Have the same input and output signal configurations, such as type, order, and dimensions
  - Define the same states as the controller prediction model, mpcobj.Model.Plant

- An array of up to *p*+1 delay-free, discrete-time state-space models, where *p* is the prediction horizon of mpcobj. Use this option to vary the controller prediction model over the prediction horizon.

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

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

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

Nominal — Updated nominal conditions
structure | structure array | [ ]

Updated nominal conditions, specified as one of the following:

- A structure of with the following fields:

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

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

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

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

ym — Current measured outputs
row vector of length N_{ym}

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

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

r — Plant output reference values
p-by-N_y array | [ ]

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

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

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

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

v — Current and anticipated measured disturbances
p-by-N_{md} array | [ ]

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

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

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

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

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

options — Override values for selected controller properties
mpcmoveopt object

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

Output Arguments

mv — Optimal manipulated variable moves
column vector

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

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

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

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

info — Solution details
structure

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

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

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

Uopt(i,:) contains the calculated optimal values at time k+i-1, for i = 1,...,p, where k is the current time. The first row of Info.Uopt contains the same manipulated variable values as output

argument `mv`. Since the controller does not calculate optimal control moves at time `k+p`, `Uopt(p+1,:)` is equal to `Uopt(p,:)`.

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

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

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

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

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

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

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

Topt — Time intervals
column vector of length p+1

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

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

Slack — Slack variable
nonnegative scalar

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

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

See "Optimization Problem" for more information.

Iterations — Number of solver iterations
positive integer | 0 | -1 | -2

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

- Positive integer — Number of iterations needed to solve the optimization problem that determines the optimal sequences.

- `0` — Optimization problem could not be solved in the specified maximum number of iterations.
- `−1` — Optimization problem was infeasible. An optimization problem is infeasible if no solution can satisfy all the hard constraints.
- `−2` — Numerical error occurred when solving the optimization problem.

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

Optimization solution status, returned as one of the following:

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

Cost — Objective function cost
nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. For more information, see "Optimization Problem".

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

Tips

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

Version History
Introduced in R2014b

See Also

Functions
mpcmove | review | sim | setEstimator | getEstimator

Objects
mpc | mpcmoveopt | mpcstate

mpcmoveCodeGeneration

Compute optimal control moves with code generation support

Syntax

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

Description

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

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

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

Examples

Compute Optimal Control Moves Using Code Generation Data Structures

Create a proper plant model.

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

Specify the controller sample time.

```
Ts = 0.1;
```

Create an MPC controller.

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

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Create code generation data structures.

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

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
-->Converting model to discrete time.
```

```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

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

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

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

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

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

```
Code generation successful.
```

Input Arguments

`configData` — MPC configuration parameters
structure

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

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

`stateData` — Controller state
structure

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

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

`stateData` has the following fields:

Plant — Plant model state estimates
mpcobj nominal plant states (default) | column vector of length n_{xp}

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

Note If custom state estimation is enabled, update `Plant` at each control interval. Otherwise, do not change this field. Instead use the values returned by either `getCodeGenerationData` or `mpcmoveCodeGeneration`.

Disturbance — Unmeasured disturbance model state estimates
[] (default) | column vector

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

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

Note If custom state estimation is enabled, update `Disturbance` at each control interval. Otherwise, do not change this field. Instead use the values returned by either `getCodeGenerationData` or `mpcmoveCodeGeneration`.

Noise — Output measurement noise model state estimates
[] (default) | column vector

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

Note If custom state estimation is enabled, update `Noise` at each control interval. Otherwise, do not change this field. Instead use the values returned by either `getCodeGenerationData` or `mpcmoveCodeGeneration`.

LastMove — Manipulated variable control moves from previous control interval
mpcobj nominal MV values (default) | column vector

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

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

Covariance — Covariance matrix for controller state estimates
symmetrical array

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

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

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

---

iA — Active inequality constraints
false (default) | logical vector

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

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

---

onlineData — Online controller data
structure

Online controller data that you must update at run time, specified as a structure with the following fields. Generate the initial structure using `getCodeGenerationData`.

signals — Updated input and output signals
structure

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

ym — Measured outputs
vector

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

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

ref — Output references
row vector | array

Output references, specified as one of the following:

- Row vector of length N_y, where N_y is the number of outputs.
- If you are using reference signal previewing with implicit or adaptive MPC, specify a *p*-by-N_y array, where *p* is the prediction horizon.

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

md — Measured disturbances
row vector | array

Measured disturbances, specified as:

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

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

`mvTarget` — Targets for manipulated variables
[ ] (default) | vector

Targets for manipulated variables, which replace the targets defined in `configData.uTarget`, specified as one of the following:

- Vector of length N_{mv}, where N_{mv} is the number of manipulated variables
- [ ] to use the default targets defined in `configData.uTarget`

This field is ignored when using an explicit MPC controller.

`externalMV` — Manipulated variables externally applied to the plant
[ ] (default) | vector

Manipulated variables externally applied to the plant, specified as:

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

`weights` — Updated QP optimization weights
structure

Updated QP optimization weights, specified as a structure. If you do not expect tuning weights to change at run time, ignore `weights`. This field is ignored when using an explicit MPC controller.

This structure contains the following fields:

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

Output variable tuning weights that replace the original controller output weights at run time at run time, specified as a row vector or array of nonnegative values.

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

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

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

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

Manipulated variable tuning weights that replace the original controller manipulated variable weights at run time, specified as a row vector or array of nonnegative values.

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

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

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

du — Manipulated variable rate tuning weights
[] (default) | row vector | array

Manipulated variable rate tuning weights that replace the original controller manipulated variable rate weights at run time, specified as a row vector or array of nonnegative values.

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

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

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

ecr — Weight on slack variable used for constraint softening
[] (default) | nonnegative scalar

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

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

limits — Updated input and output constraints
structure

Updated input and output constraints, specified as a structure. If you do not expect constraints to change at run time, ignore limits. This field is ignored when using an explicit MPC controller.

This structure contains the following fields:

ymin — Output variable lower bounds
[] (default) | column vector

Output variable lower bounds, specified as a column vector of length N_y. ymin(i) replaces the OutputVariables(i).Min constraint from the original controller. If the OutputVariables(i).Min property of the controller is specified as a vector, ymin(i) replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

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

ymax — Output variable upper bounds
[] (default) | column vector

Output variable upper bounds, specified as a column vector of length N_y. `ymax(i)` replaces the `OutputVariables(i).Max` constraint from the original controller. If the `OutputVariables(i).Max` property of the controller is specified as a vector, `ymax(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

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

umin — Manipulated variable lower bounds
`[]` (default) | column vector

Manipulated variable lower bounds, specified as a column vector of length N_{mv}. `umin(i)` replaces the `ManipulatedVariables(i).Min` constraint from the original controller. If the `ManipulatedVariables(i).Min` property of the controller is specified as a vector, `umin(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

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

umax — Manipulated variable upper bounds
`[]` (default) | column vector

Manipulated variable upper bounds, specified as a column vector of length N_{mv}. `umax(i)` replaces the `ManipulatedVariables(i).Max` constraint from the original controller. If the `ManipulatedVariables(i).Max` property of the controller is specified as a vector, `umax(i)` replaces the first finite entry in this vector, and the remaining values shift to retain the same constraint profile.

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

customconstraints — Updated custom mixed input/output constraints
structure

Updated custom mixed input/output constraints, specified as a structure. This field is ignored when using an explicit MPC controller.

This structure has the following fields:

E — Manipulated variable constraint constant
`[]` (default) | N_c-by-N_{mv} array

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

If `E` is empty, `[]`, the corresponding constraint defined in the original MPC controller are used.

F — Controlled output constraint constant
`[]` (default) | N_c-by-N_y array

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

G — Mixed input/output constraint constant
`[]` (default) | column vector of length N_c

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

S — Measured disturbance constraint constant
[] (default) | N_c-by-N_v array

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

horizons — Updated controller horizons
structure

Updated controller horizons, specified as a structure. To vary horizons at run time, first create your data structures using `getCodeGenerationData` setting the `UseVariableHorizon` name-value pair to `true`. When you vary the horizons, you must specify both the prediction horizon and the control horizon. For more information, see "Adjust Horizons at Run Time".

This field is ignored when using an explicit MPC controller.

This structure has the following fields:

p — Prediction horizon
[] (default) | positive integer

Prediction horizon, which replaces the value of `configData.p` at run time, specified as a positive integer.

Specifying `p` changes the:

- Number of rows in the optimal sequences returned in `info`
- The maximum dimensions of the fields in `model` when `configData.IsLTV` is `true`

m — Control horizon
[] (default) | positive integer | vector of positive integers

Control horizon, which replaces the value of `configData.m` at run time, specified as one of the following:

- Positive integer, m, between `1` and p, inclusive, where p is the prediction horizon (`horizons.p`). In this case, the controller computes m free control moves occurring at times k through $k+m$-1, and holds the controller output constant for the remaining prediction horizon steps from $k+m$ through $k+p$-1. Here, k is the current control interval. For optimal trajectory planning set m equal to p.
- Vector of positive integers, $[m_1, m_2, ...]$, where the sum of the integers equals the prediction horizon, p. In this case, the controller computes M blocks of free moves, where M is the length of the control horizon vector. The first free move applies to times k through $k+m_1$-1, the second free move applies from time $k+m_1$ through $k+m_1+m_2$-1, and so on. Using block moves can improve the robustness of your controller compared to the default case.

model — Updated plant and nominal values
structure

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

This structure contains the following fields:

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

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

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

where N_x is the number of plant states.

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

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

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

where N_u is the number of plant inputs.

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

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

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

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

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

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

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

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

Nominal plant states, specified as:

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

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

Nominal plant inputs, specified as:

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

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

Nominal plant outputs, specified as:

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

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

Nominal plant state derivatives, specified as:

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

Output Arguments

mv — Optimal manipulated variable moves
column vector

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

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

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

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

newStateData — Updated controller state
structure

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

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

info — Controller optimization information
structure

Controller optimization information, returned as a structure.

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

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

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

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

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

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

For more information, see `mpcmoveExplicit`.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can generate code for both implicit and explicit MPC controllers.
- To generate code for computing optimal MPC control moves:

    1 Generate data structures from an MPC controller or explicit MPC controller using `getCodeGenerationData`.
    2 To verify that your controller produces the expected closed-loop results, simulate it using `mpcmoveCodeGeneration` in place of `mpcmove`.
    3 Generate code for `mpcmoveCodeGeneration` using `codegen`. This step requires MATLAB Coder software.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
getCodeGenerationData | mpcmove | mpcmoveExplicit | mpcmoveAdaptive | buildMEX | codegen

Objects
mpc | mpcmoveopt | mpcstate

Topics
"Generate Code to Compute Optimal MPC Moves in MATLAB"
"Generate Code and Deploy Controller to Real-Time Targets"

mpcmoveExplicit

Compute optimal control using explicit MPC

Syntax

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

Description

Use this command to simulate an explicit MPC controller in closed-loop with a plant model. Call `mpcmoveExplicit` repeatedly in a for loop to calculate the manipulated variable and update the controller states at each time step.

`mv = mpcmoveExplicit(empcobj,x,ym,r)` returns the optimal move `mv` and updates the states *xc* of the controller `empcobj`.

The manipulated variable `mv` at the current time is calculated given:

- the controller object, `empcobj`,
- a pointer to the current estimated extended state, `xc`,
- the measured plant outputs, `ym`,
- the output references, `r`,
- and the measured disturbance input, `v`.

If `ym`, `r` or `v` is specified as `[]`, or if it is missing as a last input argument, `mpcmove` uses the appropriate `mpcobj.Model.Nominal` value instead.

When using default state estimation, `mpcmoveExplicit` also updates the controller state referenced by the handle object `xc`. Therefore, when using default state estimation, `xc` always points to the updated controller state. When using custom state estimation, you should update `xc` prior to each `mpcmoveExplicit` call.

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

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

Examples

Simulate explicit MPC using `mpcmoveExplicit`

This example shows how to use `mpcmoveExplicit` to simulate a plant in closed loop with an explicit MPC controller.

First, define the sample time, the plant (for this example, a double integrator), and create a traditional MPC object.

```
Ts = 0.1;
plant = tf(1,[1 0 0]);
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Define constraints on the manipulated variable.

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

The MPC controller states include states from the plant model, the disturbance model noise model, and the last values of the manipulated variables, in that order. To create a range structure where you can specify the range for each state, reference, and manipulated variable, use `generateExplicitRange`.

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

If at run time any of these variables falls outside its range, the controller returns an error status and sets the manipulated variables to their last values. Therefore, it is important that you do not underestimate these ranges. For this example, use the following ranges.

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];

range.Reference.Min = -2;
range.Reference.Max = 2;

range.ManipulatedVariable.Min = mpcobj.MV.Min -1;
range.ManipulatedVariable.Max = mpcobj.MV.Max +1;
```

Create an explicit MPC controller from the traditional MPC object and the range structure.

```
empcobj = generateExplicitMPC(mpcobj, range);
```

```
Regions found / unexplored:          9/        0
```

Set up the number of simulation steps and initialize arrays to store the plant input and output signals (so they can be plotted later).

```
N = round(5/Ts);
U = zeros(N,1);
Y = zeros(N,1);
```

Discretize the plant, and set up its initial condition.

```
dtplant = ss(c2d(plant,Ts));
x = [0 0]';
```

To obtain an handle (that is a pointer) to the controller state, use `mpcstate`.

```
xc = mpcstate(empcobj)

MPCSTATE object with fields
          Plant: [0 0]
    Disturbance: [1x0 double]
          Noise: [1x0 double]
       LastMove: 0
     Covariance: [2x2 double]
```

The controller has two states for the internal plant model, and one to hold the last value of the manipulated variable. All these states are initialized to zero.

Iteratively simulate the closed-loop response to a reference signal of `0.8`. To calculate the explicit MPC controller move, use `mpcmoveExplicit`.

```
for k = 1:N

    % update plant measurement and store signal
    y = dtplant.C*x;
    Y(k)=y;

    % compute explicit MPC action and store signal
    u = mpcmoveExplicit(empcobj,xc,y,0.8);
    U(k)=u;

    % update plant state
    x = dtplant.A*x + dtplant.B*u;
end
```

Plot the resulting plant input and output signals.

```
plot(1:N,[U Y])
title('Closed loop response')
legend('mv','output')
xlabel('steps')
grid
```
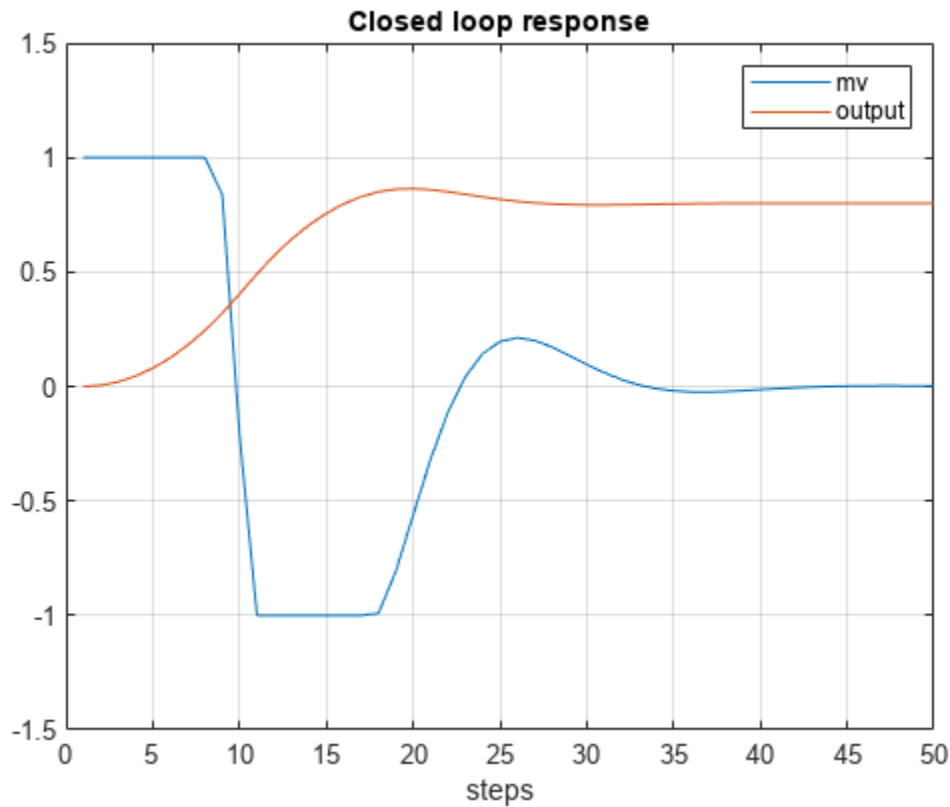
Input Arguments

empcobj — Explicit MPC controller
explicit MPC controller object

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

x — Current MPC controller state
`mpcstate` object

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

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

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

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

ym — Current measured outputs
vector

Current measured outputs, specified as a row vector of length N_{ym}, where N_{ym} is the number of measured outputs. If you are using custom state estimation, ym is ignored. If you set ym = [], then mpcmoveExplicit uses the appropriate nominal value.

r — Plant output reference values
vector

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

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

v — Current and anticipated measured disturbances
vector

Current and anticipated measured disturbances, specified as a vector of length N_{md}, where N_{md} is the number of measured disturbances. In contrast to mpcmove and mpcmoveAdaptive, mpcmoveExplicit does not support disturbance previewing. If your plant model does not include measured disturbances, use v = [].

MVused — Manipulated variable values from previous interval
vector

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

Output Arguments

mv — Optimal manipulated variable moves
column vector

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

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

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

- Remains at its most recent successful solution if the Optimizer.UseSuboptimalSolution property of the controller is false.

- Is the suboptimal solution reached after the final iteration if the Optimizer.UseSuboptimalSolution property of the controller is true. For more information, see "Suboptimal QP Solution".

`info` — Explicit MPC solution status
structure

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

`ExitCode` — Solution status code
1 | 0 | –1

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

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

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

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

Tips

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

Version History
Introduced in R2014b

See Also

Functions
generateExplicitMPC | generateExplicitRange | generateExplicitOptions | simplify | generatePlotParameters | plotSection | sim

Objects
mpc | explicitMPC | mpcstate

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"
"What is Model Predictive Control?"

mpcmoveMultiple

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

Syntax

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

Description

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

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

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

Input Arguments

`MPCArray` — MPC controllers
cell array of MPC controller objects

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

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

`states` — Current MPC controller states
cell array of `mpcstate` objects

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

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

If you are using default state estimation, `mpcmoveMultiple` expects x to represent x[n|n-1] (where x is one entry in `states`, the current state of one MPC controller in `MPCArray`). The

mpcmoveMultiple command updates the state values in the previous control interval with that information. Therefore, you should not programmatically update x at all. The default state estimator employs a steady-state Kalman filter.

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

index — Index of selected controller
positive integer

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

ym — Current measured outputs
row vector

Current measured outputs, specified as a row vector of length N_{ym}, where N_{ym} is the number of measured outputs. If you are using custom state estimation, ym is ignored. If you set ym = [], then mpcmoveMultiple uses the appropriate nominal value.

r — Plant output reference values
array

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

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

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

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

v — Current and anticipated measured disturbances
array

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

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

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

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

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

options — Override values for selected controller properties
mpcmoveopt object

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

Output Arguments

mv — Optimal manipulated variable moves
column vector

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

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

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

- Remains at its most recent successful solution if the Optimizer.UseSuboptimalSolution property of the controller is false.

- Is the suboptimal solution reached after the final iteration if the Optimizer.UseSuboptimalSolution property of the controller is true. For more information, see "Suboptimal QP Solution".

info — Solution details
structure

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

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

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

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

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

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

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

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

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

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

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

Topt — Time intervals
column vector of length p+1

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

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

Slack — Slack variable
nonnegative scalar

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

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

See "Optimization Problem" for more information.

Iterations — Number of solver iterations
positive integer | 0 | -1 | -2

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

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

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

Optimization solution status, returned as one of the following:

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

Cost — Objective function cost
nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives. For more information, see "Optimization Problem".

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

Tips

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

Version History
Introduced in R2014b

See Also

Functions
generateExplicitMPC | mpcmove | review | sim | setEstimator | getEstimator

Objects
mpcstate

mpcprops

Provide help on MPC controller properties

Syntax

```
mpcprops
```

Description

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

Examples

Describe properties of MPC objects

Display all fields of MPC objects, with related explanation.

```
MPC controller properties (with Ny output variables
    Nu manipulated variables, a prediction horizon of p intervals,
    and control horizon of m intervals):

Model -
A structure consisting of plant,
disturbance and noise models and their nominal values.
    Model.Plant -
    Plant model
        (LTI or linear model from System Identification Toolbox).
        Default: none, must be specified.

    Model.Disturbance -
        Model describing unmeasured input disturbances. Must be
        an LTI or linear model from System Identification Toolbox.
        Default: integrator (models step disturbance).
        See also: "getindist" and "setindist" commands.
    Model.Noise -
        Model describing added output measurement noise. Must be
        an LTI or linear model from System Identification Toolbox.
        Default: unity gain (models white noise).
    Model.Nominal -
        Structure containing nominal state, input,
        and output variable values:
        Model.Nominal.X -
            State of Model.Plant at the operating point.
        Model.Nominal.U -
            Input of Model.Plant at the operating point.
        Model.Nominal.Y -
            Output of Model.Plant at the operating point.
        Model.Nominal.DX -
            State derivative (for continuous time models) or update
```

```
             (for discrete time models) at the operating point.
          Default: all nominal values set to zero.

    Define input signal types in Model.Plant.InputGroup:
        ManipulatedVariables (or MV or Manipulated or Input) -
            Indices of manipulated variables.
        UnmeasuredDisturbances (or UD or Unmeasured) -
            Indices of unmeasured disturbances.
        MeasuredDisturbances (or MD or Measured) -
            Indices of measured disturbances.
    By default, all the plant inputs are manipulated variables.
    See also:  the "setmpcsignals" command.

    Define output signal types in Model.Plant.OutputGroup:
        MeasuredOutputs (or MO or Measured) -
            Indices of measured outputs.
        UnmeasuredOutputs (or UO or Unmeasured) -
            Indices of unmeasured outputs.
    By default, all the plant outputs are measured outputs.
    See also:  the "setmpcsignals" command.

Ts -
   Sample time of the MPC controller
   (in the same time units as Model.Plant).
   Default: if Model.Plant.Ts > 0, MPC.Ts = Model.Plant.Ts;
            otherwise, MPC.Ts must be specified.

PredictionHorizon -
    Intervals in the prediction horizon (scalar)
    Default: 10 + max intervals of delay in Model.Plant

ControlHorizon -
    Intervals in the control horizon.
    Either a scalar or a vector of blocked moves.
    Default: 2

Weights -
    A structure defining dimensionless
    MPC weights with the following fields:
    Weights.ManipulatedVariables
        (or MV or Manipulated or Input) -
        Must be a (min 1, max p) x Nu matrix of weights
        on the manipulated variables.
        Default: zeros(1,Nu)
    Weights.ManipulatedVariablesRate
        (or MVRate or ManipulatedRate or InputRate) -
        Must be a (min 1, max p) x Nu matrix of weights on the rates
        of the manipulated variables. MV Rates are defined as the
        differences between current and ptrevious MV values.
        Default: 0.1*ones(1,Nu)
    Weights.OutputVariables (or OV or Output) -
        Must be a (min 1, max p) x Ny matrix of weights
        on the plant outputs.
        Default: if Ny<=Nu, ones(1,Ny); otherwise,
                 only Nu outputs are weighted,
                 with preference on measured outputs.
    Weights.ECR - Scalar weight on the slack variable
        Equal Concern for Relaxation, used for constraint softening.
```

```
            Default: 1e5*max(Weights)

        Alternative weighting:
        This syntax allows for off-diagonal weights, but requires the
        weights to be identical at each prediction horizon step.
        Using this syntax, Weights.MV={R}, where R is a Nu x Nu
        symmetric and positive semi-definite matrix,
        which must be constant over the prediction horizon.
        The syntax for Weights.MVRate and Weights.OV is similar.

    ManipulatedVariables (or MV or Manipulated or Input) -
        Array of structures with fields:
        MV(i).Min -
            1 to p dimensional vector of lower bounds on MV #i
            Ddefault: -Inf
             .Max -
            1 to P dimensional vector of upper bounds on MV #i
            Default:  Inf
             .MinECR -
            1 to p dimensional vector of weights for softening the
            lower bounds on MV #i
            Default: 0, (hard constraint).
             .MaxECR -
            1 to p dimensional vector of weights for softening the
            upper bounds on MV #i
            Default: 0, (hard constraint).
             .RateMin -
            1 to p dimensional vector of lower bounds on the rate of MV #i
            Default: -Inf
             .RateMax -
            1 to p dimensional vector of upper bounds on the rate of MV #i
            Default: Inf
             .MinECR -
            1 to p dimensional vector of weights for softening the
            lower bounds on the MV #i rate
            Default: 0, (hard constraint).
             .MaxECR -
            1 to p dimensional vector of weights for softening the
            upper bounds on the MV #i rate
            Default: 0, (hard constraint).
             .Target -
            1 to p dimensional vector of target values for MV #i
            Default: Model.Nominal.U
             .Name -
            Name of MV #i
            Default: from Model.Plant.InputName
             .Units -
            String specifying the engineering units for MV #i
             .ScaleFactor -
            A scalar in engineering units.
            Each MV will be divided by its scale factor to form the
            dimensionless signal used in the MPC computations.
            Default: 1
               .Type -
            Type of variable MV #i (default: 'continuous').
            Type can be 'continuous', 'binary', 'integer',
            or an array containing all the values MV #i can take.
```

```
OutputVariables (or OV or Controlled or Output) -
        Array of structures with fields:
    OV(i).Min -
        1 to p dimensional vector of lower bounds on OV #i.
        Default: -Inf
         .Max -
        1 to p dimensional vector of upper bounds on OV #i.
        Default:  Inf
         .MinECR -
        1 to p dimensional vector of weights for softening the
        lower bounds on OV #i.
        Default: 1, (soft constraint).
         .MaxECR -
        1 to p dimensional vector of weights for softening the
        upper bounds on OV #i.
        Default: 1, (soft constraint).
         .Name -
        Name of OV #i (default: Model.Plant.OutputName{i})
         .Units -
        String specifying the engineering units for OV #i
         .ScaleFactor -
        A scalar in engineering units.
        Each OV will be divided by its scale factor to form the
        dimensionless signal used in the MPC computations.
        Default: 1

DisturbanceVariables (or DV or Disturbance) -
    Array of structures with fields:
    DV(i).Name -
        Name of DV #i (default: from Model.Plant.InputName).
         .Units -
        String specifying the engineering units for DV #i
         .ScaleFactor -
        A scalar in engineering units.
        Each DV will be divided by its scale factor to form the
        dimensionless signal used in all the MPC computations.
        Default: 1
    Note: DV consists in all the measured disturbance inputs
          followed by all the unmeasured disturbance inputs.

Optimizer -
    QP optimizer parameter structure with fields:
    Optimizer.Algorithm -
        Algorithm used by the solver.
        Default: 'active-set'
    Optimizer.ActiveSetOptions -
        Active-set solver options.
    Optimizer.InteriorPointOptions -
        Interior-point solver options.
    Optimizer.MixedIntegerOptions -
        Mixed-integer solver options.
    Optimizer.MinOutputECR -
        Minimum value of output MinECR and MaxECR.
        Default: 0
    Optimizer.UseSuboptimalSolution -
        If True the controller applies the sub-optimal solution
        when the maximum number of iterations is exceeded.
        Default: false.
```

```
    Optimizer.CustomSolver -
        If true the custom QP solver is used for simulation.
        Default: false.
    Optimizer.CustomSolverCodeGen -
        If true the custom QP solver is used for code generation.
        Default: false.

 Notes - User's notes.
     It can be a string or a cell array of strings.

 UserData - Additional information or data.
     It can be any MATLAB data type.

 History - Date and time of the MPC object creation.

 See the "mpc" command for construction syntax.
```

Version History

Introduced before R2006a

See Also

Functions
set | get | mpcverbosity | review

Blocks
MPC Controller

mpcqpsolver

(To be removed) Solve a quadratic programming problem using the KWIK algorithm

---

Note `mpcqpsolver` will be removed in a future release. Use `mpcActiveSetSolver` instead. For more information, see "Compatibility Considerations".

Syntax

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

Description

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

$$J = \frac{1}{2}x^\mathsf{T}Hx + f^\mathsf{T}x$$

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

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

Examples

Solve Quadratic Programming Problem Using Active-Set Solver

Find the values of x that minimize

$$f(x) = 0.5x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2,$$

subject to the constraints

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

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

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

Specify the inequality constraint parameters.

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

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

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

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

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

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

```
p
```

```
p = 0
```

Create a default option set for `mpcActiveSetSolver`.

```
opt = mpcqpsolverOptions;
```

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

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

Solve the QP problem.

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

Examine the solution, `x`.

```
x
```

```
x = 2×1

    0.6667
    1.3333
```

Check Active Inequality Constraints for QP Solution

Find the values of x that minimize

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

subject to the constraints

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

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

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

Specify the inequality constraint parameters.

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

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

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

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

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

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

```
p
```

```
p = 0
```

Create a default option set for `mpcqpsolver`.

```
opt = mpcqpsolverOptions;
```

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

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

Solve the QP problem.

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

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

```
iA
```

```
iA = 3x1 logical array

   1
   0
   0
```

There is a single active inequality constraint.

View the Lagrange multiplier for this constraint.

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

```
ans = 5.0000
```

Input Arguments

Linv — Inverse of lower-triangular Cholesky decomposition of Hessian matrix
n-by-*n* matrix

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

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

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

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

---

f — Multiplier of objective function linear term
column vector

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

A — Linear inequality constraint coefficients
m-by-*n* matrix | [ ]

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

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

b — Right-hand side of inequality constraints
column vector of length *m*

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

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

Aeq — Linear equality constraint coefficients
q-by-*n* matrix | [ ]

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

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

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

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

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

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

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

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

options — Option set for `mpcqpsolver`
structure

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

Output Arguments

x — Optimal solution to the QP problem
column vector

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

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

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

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

iA — Active inequalities
logical vector of length *m*

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

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

lambda — Lagrange multipliers
structure

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

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

Tips

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

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

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

- `mpcqpsolver` provides access to the QP solver used by Model Predictive Control Toolbox software. Use this command to solve QP problems in your own custom MPC applications.

Algorithms

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

Version History

Introduced in R2015b

R2020a: `mpcqpsolver` will be removed
Warns starting in R2020a

`mpcqpsolver` will be removed in a future release. Use `mpcActiveSetSolver` instead. There are differences between these functions that require updates to your code.

Update Code

The following differences require updates to your code:

- For `mpcActiveSetSolver`, you define inequality constraints in the form $Ax \leq b$. Previously, for `mpcqpsolver`, you defined inequality constraints in the form $Ax \geq b$

- For `mpcActiveSetSolver`, you specify solver options with a structure created using the `mpcActiveSetOptions` function. Previously, for `mpcqpsolver`, you created an option structure using the `mpcqpsolverOptions` function. These option structures contain the same options, though some option names have changed.

- By default, you pass the Hessian matrix to `mpcActiveSetSolver`. Previously, you passed the inverse of lower-triangular Cholesky decomposition (`Linv`) of the Hessian matrix to `mpcqpsolver`. To continue to use `Linv`, set the `UseHessianAsInput` field of the structure returned by `mpcActiveSetSolver` to `false`.

- When your QP problem has either no inequality constraints or no equality constraints, the corresponding `A` or `Aeq` input argument to `mpcActiveSetSolver` must be `zeros(0,n)`, where `n` is the number of decision variables. Previously, for `mpcqpsolver`, you specified these input arguments as `[]`.

This table shows some typical usages of `mpcqpsolver` and how to update your code to use `mpcActiveSetSolver` instead.

Not Recommended	Recommended

```opt = mpcqpsolverOptions;```
```[x,status] = mpcqpsolver(Linv,f,A,b,...```
```   Aeq,beq,iA0,opt);```
	```opt = mpcActiveSetOptions;```
	```opt.UseHessianAsInput = false;```
	```[x,status] = mpcActiveSetSolver(Linv,f,...```
	``` -A,-b,Aeq,beq,iA0,opt);```
	
	Alternatively, you can use the Hessian matrix, `H`.
	
	```opt = mpcActiveSetOptions;```
	```[x,status] = mpcActiveSetSolver(H,f,...```
	``` -A,-b,Aeq,beq,iA0,opt);```

```opt = mpcqpsolverOptions('single');```
```[x,status] = mpcqpsolver(Linv,f,A,b,...```
```   Aeq,beq,iA0,opt);```
	```opt = mpcActiveSetOptions('single');```
	```opt.UseHessianAsInput = false;```
	```[x,status] = mpcActiveSetSolver(Linv,f,...```
	``` -A,-b,Aeq,beq,iA0,opt);```

```opt = mpcqpsolverOptions;```
```opt.MaxIter = 300;```
```opt.FeasibilityTol = 1e-5;```
```[x,status] = mpcqpsolver(Linv,f,A,b,...```
```   Aeq,beq,iA0,opt);```
	```opt = mpcActiveSetOptions;```
	```opt.UseHessianAsInput = false;```
	```opt.MaxIterations = 300;```
	```opt.ContraintTolerance = 1e-5;```
	```[x,status] = mpcActiveSetSolver(Linv,f,...```
	``` -A,-b,Aeq,beq,iA0,opt);```

```[x,status] = mpcqpsolver(Linv,f,[],...```
```   zeros(0,1),Aeq,beq,iA0,opt);```
	```n = length(f);```
	```opt.UseHessianAsInput = false;```
	```[x,status] = mpcActiveSetSolver(Linv,f,...```
	``` zeros(0,n),zeros(0,1),Aeq,beq,iA0,opt);```

```[x,status] = mpcqpsolver(Linv,f,A,b,...```
```   [],zeros(0,1),iA0,opt);```
	```n = length(f);```
	```opt.UseHessianAsInput = false;```
	```[x,status] = mpcActiveSetSolver(Linv,f,...```
	``` -A,-b,zeros(0,n),zeros(0,1),iA0,opt);```

References

[1] Schmid, C., and L.T. Biegler. 'Quadratic Programming Methods for Reduced Hessian SQP'. *Computers & Chemical Engineering* 18, no. 9 (September 1994): 817–32. https://doi.org/10.1016/0098-1354(94)E0001-4.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

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

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

Generate C code with MATLAB Coder.

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

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

See Also

Functions
`mpcqpsolverOptions` | `mpcActiveSetSolver` | `mpcActiveSetOptions` | `mpcInteriorPointSolver` | `mpcInteriorPointOptions` | `setCustomSolver` | `quadprog`

Topics
"Solve Custom MPC Quadratic Programming Problem and Generate Code"
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

mpcqpsolverOptions

(To be removed) Create default option set for `mpcqpsolver`

Note `mpcqpsolverOptions` will be removed in a future release. Use `mpcActiveSetOptions` instead. For more information, see "Compatibility Considerations".

Syntax

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

Description

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

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

Examples

Create Default Option Set for MPC QP Solver

```
opt = mpcqpsolverOptions;
```

Create and Modify Default MPC QP Solver Option Set

Create default option set.

```
opt = mpcqpsolverOptions;
```

Specify the maximum number of iterations allowed during computation.

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

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

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

Create Option Set Specifying Input Argument Type

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

Input Arguments

type — MPC QP solver input argument data type
'double' (default) | 'single'

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

Output Arguments

options — Option set for mpcqpsolver
structure

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

Field	Description	Default
DataType	Input argument data type, specified as either 'double' or 'single'. This data type is used for both simulation and code generation, and all real input arguments to mpcqpsolver must match this type.	'double'
MaxIter	Maximum number of iterations allowed when computing the QP solution, specified as a positive integer.	200
Feasibil ityTol	Tolerance used to verify that inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger FeasibilityTol value allows for larger constraint violations.	1.0e-6
Integrit yChecks	Indicator of whether integrity checks are performed on the mpcqpsolver input data, specified as a logical value. If IntegrityChecks is true, then integrity checks are performed and diagnostic messages are displayed. Use false for code generation only.	true

Version History
Introduced in R2015b

R2020a: mpcqpsolverOptions will be removed
Warns starting in R2020a

mpcqpsolverOptions will be removed in a future release. Use mpcActiveSetOptions instead. There are differences between these functions that require updates to your code.

Update Code

To update your code:

• Change the function name from mpcqpsolverOptions to mpcActiveSetOptions. The syntaxes are equivalent.

- Some field names of the returned structure have changed. The default field values are the same. This table shows the new property names.

Previous Property Name	New Property Name
MaxIter	MaxIterations
FeasibilityTol	ConstraintTolerance

- The returned structure of `mpcActiveSetOptions` contains the new field `UseHessianAsInput`. To continue to use the inverse of the lower-triangular decomposition of the Hessian matrix with `mpcActiveSetSolver`, you must set `UseHessianAsInput` to `false`.

For syntax examples showing how to update your code, see `mpcqpsolver`.

See Also

Functions
mpcqpsolver | mpcActiveSetSolver | mpcActiveSetOptions | mpcInteriorPointSolver | mpcInteriorPointOptions | setCustomSolver | quadprog

Topics
"Solve Custom MPC Quadratic Programming Problem and Generate Code"
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

mpcverbosity

Change toolbox verbosity level

Syntax

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

Description

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

By default, messages are turned on.

`mpcverbosity off` turns messages off.

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

`mpcverbosity` shows the verbosity status.

Examples

Turn MPC verbosity off

Turn verbosity off and suppress output argument.

```
mpcverbosity off;
```

Turn MPC verbosity on

Turn verbosity on and save the old status in the workspace variable `old`

```
old = mpcverbosity on;
```

Show MPC verbosity status

Show MPC verbosity and suppress output argument.

```
mpcverbosity;
MPC verbosity is off
```

Input Arguments

new_status — new MPC verbosity status
'on' (default) | 'off'

Char array, being either 'on' or 'off'.

Example: 'off'

Output Arguments

old_status — old MPC verbosity status
'on' (default) | 'off'

Char array, being either 'on' or 'off'.

Example: 'off'

Version History
Introduced before R2006a

See Also

Functions
set | get | mpcprops | getname | setname | review

Objects
mpc

Blocks
MPC Controller

nlmpcmove

Compute optimal control action for nonlinear MPC controller

Syntax

```
mv = nlmpcmove(nlobj,x,lastmv)
mv = nlmpcmove(nlobj,x,lastmv,ref)
mv = nlmpcmove(nlobj,x,lastmv,ref,md)
mv = nlmpcmove(nlobj,x,lastmv,ref,md,options)
[mv,opt] = nlmpcmove( ___ )
[mv,opt,info] = nlmpcmove( ___ )

mv = nlmpcmove(msobj,x,lastmv)
mv = nlmpcmove(msobj,x,lastmv,simdata)
[mv,simdata] = nlmpcmove( ___ )
[mv,simdata,info] = nlmpcmove( ___ )
```

Description

Nonlinear MPC

`mv = nlmpcmove(nlobj,x,lastmv)` computes the optimal control action for the current time. To simulate closed-loop nonlinear MPC control, call `nlmpcmove` repeatedly.

`mv = nlmpcmove(nlobj,x,lastmv,ref)` specifies reference values for the plant outputs. If you do not specify reference values, `nlmpcmove` uses zeros by default.

`mv = nlmpcmove(nlobj,x,lastmv,ref,md)` specifies run-time measured disturbance values. If your controller has measured disturbances, you must specify `md`.

`mv = nlmpcmove(nlobj,x,lastmv,ref,md,options)` specifies additional run-time options for computing optimal control moves. Using `options`, you can specify initial guesses for state and manipulated variable trajectories, update tuning weights at constraints, or modify prediction model parameters.

`[mv,opt] = nlmpcmove( ___ )` returns an `nlmpcmoveopt` object that contains initial guesses for the state and manipulated trajectories to be used in the next control interval.

`[mv,opt,info] = nlmpcmove( ___ )` returns additional solution details, including the final optimization cost function value and the optimal manipulated variable, state, and output trajectories.

Multistage Nonlinear MPC

`mv = nlmpcmove(msobj,x,lastmv)` computes the optimal control action for the current time. To simulate closed-loop nonlinear MPC control, call `nlmpcmove` repeatedly.

`mv = nlmpcmove(msobj,x,lastmv,simdata)` specifies the additional `simdata` structure, which contains measured disturbances, run-time bounds, parameters for the state and stage functions, and initial guesses for state and manipulated variable trajectories. In general use the following syntax to return a new `simdata` (containing updated initial guesses) as a second output argument.

[mv,simdata] = nlmpcmove(___) returns an updated `simdata` structure that contains new initial guesses for the state and manipulated trajectories to be used in the next control interval. Good initial guesses are important since they help the solver to converge to a solution faster.

[mv,simdata,info] = nlmpcmove(___) returns additional solution details, including the final optimization cost function value and the optimal manipulated variable, state, and output trajectories.

Examples

Plan Optimal Trajectory Using Nonlinear MPC

Create a nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Specify linear constraints on the manipulated variables.

```
for ct = 1:nu
    nlobj.MV(ct).Min = 0;
    nlobj.MV(ct).Max = 1;
end
```

Validate the prediction model and custom functions at the initial states (x0) and initial inputs (u0) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Compute the optimal state and manipulated variable trajectories, which are returned in the `info`.

```
[~,~,info] = nlmpcmove(nlobj,x0,u0);
```

```
Slack variable unused or zero-weighted in your custom cost function.
All constraints will be hard.
```

Plot the optimal trajectories.

```
FlyingRobotPlotPlanning(info,Ts)
```

```
Optimal fuel consumption =   1.884953
```

Simulate Closed-Loop Control using Nonlinear MPC Controller

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

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

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

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

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

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

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

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

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

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

Define standard constraints for the controller.

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

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});

Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Kalman filter for estimating the four plant states. Its state transition function is defined in `pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero initial manipulated variable value.

```
x = [0;0;-pi;0];
y = [x(1);x(3)];
EKF.State = x;
mv = 0;
```

Specify the output reference value.

```
yref = [0 0];
```

Create an `nlmpcmoveopt` object, and specify the sample time parameter.

```
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};
```

Run the simulation for `10` seconds. During each control interval:

1   Correct the previous prediction using the current measurement.
2   Compute optimal control moves using `nlmpcmove`. This function returns the computed optimal sequences in `nloptions`. Passing the updated options object to `nlmpcmove` in the next control interval provides initial guesses for the optimal sequences.
3   Predict the model states.
4   Apply the first computed optimal control move to the plant, updating the plant states.

5 Generate sensor data with white noise.

6 Save the plant states.

```
Duration = 10;
xHistory = x;
for ct = 1:(Duration/Ts)
    % Correct previous prediction
    xk = correct(EKF,y);
    % Compute optimal control moves
    [mv,nloptions] = nlmpcmove(nlobj,xk,mv,yref,[],nloptions);
    % Predict prediction model states for the next iteration
    predict(EKF,[mv; Ts]);
    % Implement first optimal control move
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data
    y = x([1 3]) + randn(2,1)*0.01;
    % Save plant states
    xHistory = [xHistory x];
end
```

Plot the resulting state trajectories.

```
figure
subplot(2,2,1)
plot(0:Ts:Duration,xHistory(1,:))
xlabel('time')
ylabel('z')
title('cart position')
subplot(2,2,2)
plot(0:Ts:Duration,xHistory(2,:))
xlabel('time')
ylabel('zdot')
title('cart velocity')
subplot(2,2,3)
plot(0:Ts:Duration,xHistory(3,:))
xlabel('time')
ylabel('theta')
title('pendulum angle')
subplot(2,2,4)
plot(0:Ts:Duration,xHistory(4,:))
xlabel('time')
ylabel('thetadot')
title('pendulum velocity')
```

Create and Simulate Multistage Nonlinear MPC Controller

This example shows how to create and simulate a simple multistage MPC controller in closed loop, without using initial guesses, with the MATLAB® function `nlmpcmove`.

Create Multistage MPC Controller

Create a multistage nonlinear MPC object with a five-step horizon, one state, and one manipulated variable.

```
msobj = nlmpcMultistage(5,1,1);
```

Write a simple state function as a string.

```
sstr = "function xdot = mystatefcn(x,u)" + newline + ...
       "xdot = u-sin(x);" + newline + "end";
```

Write the string to the `mystatefcn.m` file.

```
sfid=fopen("mystatefcn.m","w");
fwrite(sfid,sstr,"char");
fclose(sfid);
```

Write a simple cost function as a string.

```
cstr = "function c = mycostfcn(k,x,u)" + newline + ...
       "c = abs(u)/k+k*x^2;" + newline + "end";
```

Write the string to the mycostfcn.m file.

```
cfid=fopen("mycostfcn.m","w");
fwrite(cfid,cstr,"char");
fclose(cfid);
```

Alternatively, you can write your state and cost MATLAB functions in the current folder (which is recommended because local functions are not supported for the generation of deployment code).

If files have not yet been written to the hard disk pause five second before accessing them.

```
if ~exist("mystatefcn","file")
    pause(5)
end
```

Specify the state transition function for the prediction model.

```
msobj.Model.StateFcn = "mystatefcn";
```

Specify the cost functions for last three stages.

```
for i=3:6
    msobj.Stages(i).CostFcn = "mycostfcn";
end
```

Note that, because the Jacobian of the state function is not supplied here, nlmpcmove needs to numerically calculate them at each time step, which negatively affects performance. As a best practice, supply the Jacobian of the state function, as shown in "Simulate Multistage Nonlinear MPC Controller Using Initial Guesses" on page 2-177. You can also automatically generate a Jacobian function using generateJacobianFunction.

Simulate Controller in Closed Loop

Initialize the plant state and input.

```
x=3;
mv=0;
```

Validate functions.

```
validateFcns(msobj,x,mv);
```

```
Model.StateFcn is OK.
"CostFcn" of the following stages [3 4 5 6] are OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Simulate the control loop for 5 steps, without updating the initial guess. Use the Euler method, that is dx(t)/dt ~ (x(t+1)-x(t))/Ts, to simulate the plant.

```
for k=1:5
    mv = nlmpcmove(msobj, x, mv);    % calculate move (without initial guess)
    x=x+0.2*mystatefcn(x,mv);        % update plant state assuming Ts=0.2s
end
```

Note that, because initial guesses are not supplied as an input argument, nlmpcmove needs to recalculate them at each time step, which negatively affects performance. Not supplying initial

guesses can be an acceptable starting point, but in general is not suggested. As a best practice, use updated initial guesses at each time step, as shown in "Simulate Multistage Nonlinear MPC Controller Using Initial Guesses" on page 2-177, so that `nlmpcmove` does not need to recalculate them at each time step (in the same example you also use `ode45` to calculate the evolution of the plant state until the next control interval).

Display the final values of the state and manipulated variables.

```
disp(['Final value of x =' num2str(x)])
```

```
Final value of x =1.3074
```

```
disp(['Final value of mv =' num2str(mv)])
```

```
Final value of mv =-0.35101
```

Simulate Multistage Nonlinear MPC Controller Using Initial Guesses

This example shows how to create and simulate a simple multistage MPC controller in closed loop using initial guesses, with the MATLAB® function `nlmpcmove`.

Create Multistage MPC Controller

Create a multistage MPC object with a seven-steps horizon, one state, and one manipulated variable.

```
msobj = nlmpcMultistage(7,1,1);
```

Defining your state and cost functions as separate files in the current folder on in a folder on the MATLAB path is recommended, as local functions are not supported for the generation of C/C++ deployment code. However, for this example, the state, cost, and state Jacobian functions are defined as local functions at the end of the example.

Specify the state transition function for the prediction model.

```
msobj.Model.StateFcn = @mystatefcn;
```

As a best practice, use Jacobians whenever they are available, otherwise the solver must compute it numerically. You can also automatically generate a Jacobian function using `generateJacobianFunction`.

Specify the Jacobian of the state transition function.

```
msobj.Model.StateJacFcn = @mystatejac;
```

Specify the cost functions for all stages except the first

```
for i=2:8
    msobj.Stages(i).CostFcn = @mycostfcn;
end
```

Define Initial Conditions, Create Data Structure, and Validate Functions

Initialize the plant state and input.

```
x=3;
mv=0;
```

Create the initial simulation data structure.

```
simdata = getSimulationData(msobj)
```

```
simdata = struct with fields:
    InitialGuess: []
```

Validate functions and the data structure.

```
validateFcns(msobj,x,mv,simdata);
```

```
Model.StateFcn is OK.
Model.StateJacFcn is OK.
"CostFcn" of the following stages [2 3 4 5 6 7 8] are OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Simulate Controller in Closed Loop

Simulate the control loop for 5 steps.

```
for k=1:5
    [mv,simdata] = nlmpcmove(msobj, x, mv, simdata);                    % calculate move and update
    [~,xhist] = ode45(@(t,xode) mystatefcn(xode,mv),[0 msobj.Ts],x);   % simulate plant for one s
    x = xhist(end);                                                     % update plant state
end
```

Since updated initial guesses are supplied as an input argument within the `simdata` structure, `nlmpcmove` does not need to recalculate them at each time step, which saves computation time and improves performance. Updating initial guesses at every time step is a best practice.

Display the last values of the state and manipulated variables.

```
disp(['Final value of x =' num2str(x)])
```

```
Final value of x =-0.039545
```

```
disp(['Final value of mv =' num2str(mv)])
```

```
Final value of mv =-0.066672
```

Support Functions

State transition function.

```
function xdot = mystatefcn(x,u)
    xdot = u-sin(x);
end
```

Jacobian of the state transition function.

```
function [A,B] = mystatejac(x,~)
    A = -cos(x);
    B = 1;
end
```

Stage cost functions.

```
function j = mycostfcn(s,x,u)
    j = abs(u)/s+s*x^2;
end
```

Input Arguments

nlobj — Nonlinear MPC controller
nlmpc object

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

x — Current prediction model states
vector

Current prediction model states, specified as a vector of lengthN_x, where N_x is the number of prediction model states. Since the nonlinear MPC controller does not perform state estimation, you must either measure or estimate the current prediction model states at each control interval. For more information on nonlinear MPC prediction models, see "Specify Prediction Model for Nonlinear MPC".

lastmv — Control signals used in plant at previous control interval
vector

Control signals used in plant at previous control interval, specified as a vector of lengthN_{mv}, where N_{mv} is the number of manipulated variables.

Note Specify `lastmv` as the manipulated variable signals applied to the plant in the previous control interval. Typically, these signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode; that is, the controller output is not driving the plant, then feeding the actual control signal to `last_mv` can help achieve bumpless transfer when the controller is switched back online.

---

ref — Plant output reference values
[ ] (default) | row vector | array

Plant output reference values, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. If you do not specify `ref`, the default reference values are zero.

To use the same reference values across the prediction horizon, specify a row vector.

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

md — Measured disturbance values
[ ] (default) | row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your controller has measured disturbances, you must specify `md`. If your controller has no measured disturbances, specify `md` as [ ].

To use the same disturbance values across the prediction horizon, specify a row vector.

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

options — Run-time options
nlmpcmoveopt object

Run-time options, specified as an `nlmpcmoveopt` object. Using these options, you can:

- Tune controller weights
- Update linear constraints
- Set manipulated variable targets
- Specify prediction model parameters
- Provide initial guesses for state and manipulated variable trajectories

These options apply to only the current `nlmpcmove` time instant.

To improve solver efficiency, it is best practice to specify initial guesses for the state and manipulated variable trajectories.

msobj — Nonlinear Multistage MPC controller
nlmpcMultistage object

Multistage nonlinear MPC controller, specified as an `nlmpcMultistage` object.

simdata — Run-time simulation data
structure

Run-time simulation data, specified as structure. It must be initially created by `getSimulationData`, and then populated (if needed) before being passed to `nlmpcmove` as an input argument. An updated version is then always returned as a second output argument of `nlmpcmove`. Note that the `MVMin`, `MVMax`, `StateMin`, `StateMax`, `MVRateMin`, `MVRateMax` fields are needed only if you want to change these bounds at run time. These fields exist in the structure returned by `getSimulationData` only if you enable them explicitly when calling `getSimulationData`. The `simdata` structure has the following fields.

MeasuredDisturbance — Measured disturbance values
[] (default) | row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your multistage MPC object has any measured disturbance channel defined, you must specify `MeasuredDisturbance`. If your controller has no measured disturbances, this field does not exist in the structure generated by `getSimulationData`.

To use the same disturbance values across the prediction horizon, specify a row vector.

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

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

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

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

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

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

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVRateMin — Manipulated variable rate lower bounds
[ ] (default) | row vector | matrix

Manipulated variable rate lower bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMin(:,i)` replaces the `ManipulatedVariables(i).RateMin` property of the controller at run time. `MVRateMin` bounds must be nonpositive.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVRateMax — Manipulated variable rate upper bounds
[ ] (default) | row vector | matrix

Manipulated variable rate upper bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMax(:,i)` replaces the `ManipulatedVariables(i).RateMax` property of the controller at run time. `MVRateMax` bounds must be nonnegative.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for

one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateMin — State lower bounds
[ ] (default) | row vector | matrix

State lower bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMin(:,i)` replaces the `States(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateMax — State upper bounds
[ ] (default) | row vector | matrix

State upper bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMax(:,i)` replaces the `States(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateFcnParameters — State function parameter values
[ ] (default) | vector

State function parameter values, specified as a vector with length equal to the value of the `Model.ParameterLength` property of the multistage controller object. If `Model.StateFcn` needs a parameter vector, you must provide its value at runtime using this field. If `Model.ParameterLength` is `0` this field does not exist in the structure returned by `getSimulationData`.

StageFcnParameters — Stage function parameter values
[ ] (default) | vector

Stage functions parameter values, specified as a vector with length equal to the sum of all the values in the `Stages(i).ParameterLength` properties of the multistage controller object. If any cost or constraint function defined in the `Stages` property needs a parameter vector, you must provide all the parameter vectors at runtime (stacked in a single column) using this field. If none of your stage functions have parameters, this field does not exist in the structure returned by `getSimulationData`.

You must stack the parameter vectors for all stages in the column vector `StateFcnParameters` as follows.

```
[parameter vector for stage 1;
 parameter vector for stage 2;
 ...
```

```
 parameter vector for stage p+1;
]
```

TerminalState — Terminal state
[ ] (default) | vector

Terminal state, specified as a column vector with as many elements as the number of states. The terminal state is the desired state at the last prediction step. To specify desired terminal states at run-time via this field, you must specify finite values in the `TerminalState` field of the `Model` property of `msobj`. Specify `inf` for the states that do not need to be constrained to a terminal value. At run time, `nlmpcmove` ignores any values in the `TerminalState` field of `simdata` that correspond to `inf` values in `msobj`. If you do not specify any terminal value condition in `msobj`, this field is not created in `simdata`.

If there is no `TerminalState` in `simdata` then the terminal state constraint (if present) does not change at run time.

InitialGuess — Initial guesses for the decision variables
[ ] (default) | vector

Initial guesses for the decision variables, specified as a column vector of length equal to the sum of the lengths of all the decision variable vectors for each stage. Good initial guesses are important since they help the solver to converge to a solution faster. Therefore, when simulating a control loop by calling `nlmpcmove` repeatedly in a loop, pass `simdata` as an input argument (so initial guesses can be used), and at the same time return an updated version of `simdata` (with new initial guesses for the next control interval) as an output argument.

You must be stack the initial guesses for all stages in the column vector `InitialGuess` as follows.

```
[state vector guess for stage 1;
 manipulated variable vector guess for stage 1;
 manipulated variable vector rate guess for stage 1; % if used
 slack variable vector guess for stage 1; % if used
 state vector guess for stage 2;
 manipulated variable vector guess for stage 2;
 manipulated variable vector rate guess for stage 2; % if used
 slack variable vector guess for stage 2; % if used
 ...
 state vector guess for stage p;
 manipulated variable vector guess for stage p;
 manipulated variable vector rate guess for stage p; % if used
 slack variable vector guess for stage p; % if used
 state vector guess for stage p+1;
 slack variable vector guess for stage p+1; % if used
]
```

If `InitialGuess` is [], the default initial guesses are calculated from the `x` and `lastmv` arguments passed to `nlmpcmove`.

In general, during closed-loop simulation, you do not specify `InitialGuess` yourself. Instead, when calling `nlmpcmove`, return the `simdata` output argument, which contains the calculated initial guesses for the next control interval. You can then pass `simdata` as an input argument to `nlmpcmove` for the next control interval. These steps are a best practice, even if you do not specify any other run-time options.

Output Arguments

mv — Optimal manipulated variable control action
column vector

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

If the solver converges to a local optimum solution (`info.ExitFlag` is positive), then `mv` contains the optimal solution.

If the solver reaches the maximum number of iterations without finding an optimal solution (`info.ExitFlag = 0`) and:

- `nlobj.Optimization.UseSuboptimalSolution` is `true`, then `mv` contains the suboptimal solution
- `nlobj.Optimization.UseSuboptimalSolution` is `false`, then `mv` contains `lastmv`

If the solver fails (`info.ExitFlag` is negative), then `mv` contains `lastmv`.

opt — Run-time options with initial guesses
`nlmpcmoveopt` object

Run-time options with initial guesses for the state and manipulated variable trajectories to be used in the next control interval, returned as an `nlmpcmoveopt` object. Any run-time options that you specified using `options`, such as weights, constraints, or parameters, are copied to `opt`.

The initial guesses for the states (`opt.X0`) and manipulated variables (`opt.MV0`) are the optimal trajectories computed by `nlmpcmove` and correspond to the last p-1 rows of `info.Xopt` and `info.MVopt`, respectively.

To use these initial guesses in the next control interval, specify `opt` as the `options` input argument to `nlmpcmove`.

info — Solution details
structure

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

MVopt — Optimal manipulated variable sequence
array

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

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

Xopt — Optimal prediction model state sequence
array

Optimal prediction model state sequence, returned as a (p+1)-by-N_x array, where p is the prediction horizon and N_x is the number of states in the prediction model.

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

Yopt — Optimal output variable sequence
array

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

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

Topt — Prediction horizon time sequence
column vector

Prediction horizon time sequence, returned as a column vector of length $p+1$, where p is the prediction horizon. Topt contains the time sequence from time k to time $k+p$, where k is the current time.

Topt(1) = 0 represents the current time. Subsequent time steps Topt(i) are Ts*(i-1), where Ts is the controller sample time.

Use Topt when plotting the MVopt, Xopt, or Yopt sequences.

Slack — Stacked slack variables vector
nonnegative vector

Stacked slack variables vector, used in constraint softening. If all elements are zero, then all soft constraints are satisfied over the entire prediction horizon. If any element is greater than zero, then at least one soft constraint is violated.

The slack variable vector for all stages are stacked as:

```
[slack variable vector for stage 1; % if used
 slack variable vector for stage 2; % if used
 ...
 slack variable vector for stage p+1; % if used
]
```

ExitFlag — Optimization exit code
integer

Optimization exit code, returned as one of the following:

- Positive Integer — Optimal solution found
- 0 — Feasible suboptimal solution found after the maximum number of iterations
- Negative integer — No feasible solution found

Iterations — Number of iterations
positive integer

Number of iterations used by the nonlinear programming solver, returned as a positive integer.

Cost — Objective function cost
nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives.

The cost value is only meaningful when `ExitFlag` is nonnegative.

`simdata` — Run-time simulation data structure
structure

Updated run-time simulation data, returned as a structure, containing new initial guesses for the state and manipulated trajectories to be used in the next control interval. It is a structure with the following fields.

`MeasuredDisturbance` — Measured disturbance values
[ ] (default) | row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your multistage MPC object has any measured disturbance channel defined, you must specify `MeasuredDisturbance`. If your controller has no measured disturbances, this field does not exist in the structure generated by `getSimulationData`.

To use the same disturbance values across the prediction horizon, specify a row vector.

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

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

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

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

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

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

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVRateMin — Manipulated variable rate lower bounds
[ ] (default) | row vector | matrix

Manipulated variable rate lower bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMin(:,i)` replaces the `ManipulatedVariables(i).RateMin` property of the controller at run time. `MVRateMin` bounds must be nonpositive.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVRateMax — Manipulated variable rate upper bounds
[ ] (default) | row vector | matrix

Manipulated variable rate upper bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMax(:,i)` replaces the `ManipulatedVariables(i).RateMax` property of the controller at run time. `MVRateMax` bounds must be nonnegative.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateMin — State lower bounds
[ ] (default) | row vector | matrix

State lower bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMin(:,i)` replaces the `States(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateMax — State upper bounds
[ ] (default) | row vector | matrix

State upper bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMax(:,i)` replaces the `States(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for

one prediction horizon step. If you specify fewer than *p* rows, the final bounds are used for the remaining steps of the prediction horizon.

StateFcnParameters — State function parameter values
`[ ]` (default) | vector

State function parameter values, specified as a vector with length equal to the value of the `Model.ParameterLength` property of the multistage controller object. If `Model.StateFcn` needs a parameter vector, you must provide its value at runtime using this field. If `Model.ParameterLength` is `0` this field does not exist in the structure returned by `getSimulationData`.

StageFcnParameters — Stage function parameter values
`[ ]` (default) | vector

Stage functions parameter values, specified as a vector with length equal to the sum of all the values in the `Stages(i).ParameterLength` properties of the multistage controller object. If any cost or constraint function defined in the `Stages` property needs a parameter vector, you must provide all the parameter vectors at runtime (stacked in a single column) using this field. If none of your stage functions have parameters, this field does not exist in the structure returned by `getSimulationData`.

You must stack the parameter vectors for all stages in the column vector `StateFcnParameters` as follows.

```
[parameter vector for stage 1;
 parameter vector for stage 2;
 ...
 parameter vector for stage p+1;
]
```

TerminalState — Terminal state
`[ ]` (default) | vector

Terminal state, specified as a column vector with as many elements as the number of states. The terminal state is the desired state at the last prediction step. To specify desired terminal states at run-time via this field, you must specify finite values in the `TerminalState` field of the `Model` property of `msobj`. Specify `inf` for the states that do not need to be constrained to a terminal value. At run time, `nlmpcmove` ignores any values in the `TerminalState` field of `simdata` that correspond to `inf` values in `msobj`. If you do not specify any terminal value condition in `msobj`, this field is not created in `simdata`.

If there is no `TerminalState` in `simdata` then the terminal state constraint (if present) does not change at run time.

InitialGuess — Initial guesses for the decision variables
`[ ]` (default) | vector

Initial guesses for the decision variables, specified as a column vector of length equal to the sum of the lengths of all the decision variable vectors for each stage. Good initial guesses are important since they help the solver to converge to a solution faster. Therefore, when simulating a control loop by calling `nlmpcmove` repeatedly in a loop, pass `simdata` as an input argument (so initial guesses can be used), and at the same time return an updated version of `simdata` (with new initial guesses for the next control interval) as an output argument.

You must be stack the initial guesses for all stages in the column vector `InitialGuess` as follows.

```
[state vector guess for stage 1;
 manipulated variable vector guess for stage 1;
 manipulated variable vector rate guess for stage 1; % if used
 slack variable vector guess for stage 1; % if used
 state vector guess for stage 2;
 manipulated variable vector guess for stage 2;
 manipulated variable vector rate guess for stage 2; % if used
 slack variable vector guess for stage 2; % if used
 ...
 state vector guess for stage p;
 manipulated variable vector guess for stage p;
 manipulated variable vector rate guess for stage p; % if used
 slack variable vector guess for stage p; % if used
 state vector guess for stage p+1;
 slack variable vector guess for stage p+1; % if used
]
```

If `InitialGuess` is `[]`, the default initial guesses are calculated from the `x` and `lastmv` arguments passed to `nlmpcmove`.

In general, during closed-loop simulation, you do not specify `InitialGuess` yourself. Instead, when calling `nlmpcmove`, return the `simdata` output argument, which contains the calculated initial guesses for the next control interval. You can then pass `simdata` as an input argument to `nlmpcmove` for the next control interval. These steps are a best practice, even if you do not specify any other run-time options.

Tips

During closed-loop simulations, it is best practice to *warm start* the nonlinear solver by using the predicted state and manipulated variable trajectories from the previous control interval as the initial guesses for the current control interval. To use these trajectories as initial guesses:

1 Return the `opt` output argument when calling `nlmpcmove`. This `nlmpcmoveopt` object contains any run-time options you specified in the previous call to `nlmpcmove`, along with the initial guesses for the state (`opt.X0`) and manipulated variable (`opt.MV0`) trajectories.

2 Pass this object in as the `options` input argument to `nlmpcmove` for the next control interval.

These steps are a best practice, even if you do not specify any other run-time options.

Version History
Introduced in R2018b

See Also

Functions
validateFcns | nlmpcmoveCodeGeneration | getSimulationData | getCodeGenerationData

Objects
nlmpc | nlmpcMultistage | nlmpcmoveopt

Topics
"Swing-Up Control of a Pendulum Using Nonlinear Model Predictive Control"

"Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC"
"Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC"
"Nonlinear MPC"

nlmpcmoveCodeGeneration

Compute nonlinear MPC control moves with code generation support

Syntax

```
[mv,newOnlineData] = nlmpcmoveCodeGeneration(coreData,x,lastMV,onlineData)
[ ___ ,info] = nlmpcmoveCodeGeneration( ___ )
```

Description

`[mv,newOnlineData] = nlmpcmoveCodeGeneration(coreData,x,lastMV,onlineData)` computes optimal nonlinear MPC control moves and supports code generation for deployment to real-time targets. Control moves are calculated using the current prediction model states (`x`), the control moves from the previous control interval (`lastMV`), and input data structures (`coreData` and `nlOnlineData`) generated using `getCodeGenerationData`.

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

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

Examples

Compute Nonlinear MPC Control Moves Using Code Generation Data Structures

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

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

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

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

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

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

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

```
nlobj.Model.NumberOfParameters = 1;
params = {Ts};
```

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

```
nlobj.Model.OutputFcn = "pendulumOutputFcn";
```

Define standard constraints for the controller.

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

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj,x0,u0,[],params);
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Kalman filter for estimating the four plant states. Its state transition function is defined in `pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero initial manipulated variable value.

```
x0 = [0;0;-pi;0];
y0 = [x0(1);x0(3)];
EKF.State = x0;
mv0 = 0;
```

Create code generation data structures for the controller, specifying the initial conditions and parameters.

```
[coreData,onlineData] = getCodeGenerationData(nlobj,x0,mv0,params);
```

Specify the output reference value in the online data structure.

```
onlineData.ref = [0 0];
```

To verify the controller operation, run a simulation for 10 seconds. During each control interval:

1   Correct the previous prediction using the current measurement.
2   Compute optimal control moves using `nlmpcmoveCodeGeneration`. This function returns the computed optimal sequences in `onlineData`. Passing the updated data structure to `nlmpcmoveCodeGeneration` in the next control interval provides initial guesses for the optimal sequences.
3   Predict the model states.
4   Apply the first computed optimal control move to the plant, updating the plant states.
5   Generate sensor data with white noise.

6 Save the plant states.

```matlab
mv = mv0;
y = y0;
x = x0;
Duration = 10;
xHistory = x0;
for ct = 1:(Duration/Ts)
    % Correct previous prediction
    xk = correct(EKF,y);
    % Compute optimal control move
    [mv,onlineData] = nlmpcmoveCodeGeneration(coreData,xk,mv,onlineData);
    % Predict prediction model states for the next iteration
    predict(EKF,[mv; Ts]);
    % Implement first optimal control move
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data
    y = x([1 3]) + randn(2,1)*0.01;
    % Save plant states
    xHistory = [xHistory x];
end
```

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

```matlab
func = 'nlmpcmoveCodeGeneration';
funcOutput = 'nlmpcmoveMEX';
Cfg = coder.config('mex');
Cfg.DynamicMemoryAllocation = 'off';
codegen('-config',Cfg,func,'-o',funcOutput,'-args',...
    {coder.Constant(coreData),xk,mv,onlineData});
```

```
Code generation successful.
```

Input Arguments

coreData — Nonlinear MPC configuration parameters
structure

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

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

---

x — Current prediction model states
column vector

Current prediction model states, specified as a vector of lengthN_x, where N_x is the number of prediction model states. The prediction model state function is defined in `nlobj.Model.StateFcn`.

Since the nonlinear MPC controller does not perform state estimation, you must either measure or estimate the current prediction model states at each control interval. For more information on nonlinear MPC prediction models, see "Specify Prediction Model for Nonlinear MPC".

`lastMV` — Control signals used in plant at previous control interval
column vector

Control signals used in plant at previous control interval, specified as a column vector of lengthN_{mv}, where N_{mv} is the number of manipulated variables.

Note Specify `lastMV` as the manipulated variable signals applied to the plant in the previous control interval. Typically, these signals are the values generated by the controller (`mv`). However, this is not always the case. For example, if your controller is offline and running in tracking mode; that is, the controller output is not driving the plant, then feeding the actual control signal to `last_mv` can help achieve bumpless transfer when the controller is switched back online.

`onlineData` — Online controller data
structure

Online controller data that you must update at run time, specified as a structure with the following fields. Generate the initial structure using `getCodeGenerationData`. Some structure fields are not required, depending on the configuration of the controller and what weights or constraints vary at run time.

`ref` — Output reference values
row vector | array

Plant output reference values, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables.

To use the same reference values across the prediction horizon, specify a row vector.

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

If your controller cost function does not use `ref`, leave `ref` at its default value.

`mvTarget` — Manipulated variable targets
row vector | array

Manipulated variable targets, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables.

To use the same manipulated variable targets across the prediction horizon, specify a row vector.

To vary the targets over the prediction horizon (previewing) from time k to time $k+p$-1, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the targets for one prediction horizon step. If you specify fewer than p rows, the final targets are used for the remaining steps of the prediction horizon.

If your controller cost function does not use `mvTarget`, leave `mvTarget` at its default value.

`X0` — Initial guesses for the optimal state solutions
vector | array

Initial guesses for the optimal state solutions, specified as a row vector of length N_x or an array with N_x columns, where N_x is the number of states.

To use the same initial guesses across the prediction horizon, specify a row vector.

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

In general, during closed-loop simulation, you do not specify X0 yourself. Instead, when calling `nlmpcmoveCodeGeneration`, return the `newOnlineData` output argument, which contains updated X0 estimates. You can then pass `newOnlineData` in as the `onlineData` input argument to `nlmpcmoveCodeGeneration` for the next control interval.

MV0 — Initial guesses for the optimal manipulated variable solutions
vector | array

Initial guesses for the optimal manipulated variable solutions, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables.

To use the same initial guesses across the prediction horizon, specify a row vector.

To vary the initial guesses over the prediction horizon from time k to time $k+p$-1, specify an array with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the final guesses are used for the remaining steps of the prediction horizon.

In general, during closed-loop simulation, you do not specify MV0 yourself. Instead, when calling `nlmpcmoveCodeGeneration`, return the `newOnlineData` output argument, which contains updated MV0 estimates. You can then pass `newOnlineData` in as the `onlineData` input argument to `nlmpcmoveCodeGeneration` for the next control interval.

Slack0 — Initial guess for the slack variable at the solution
nonnegative scalar

Initial guess for the slack variable at the solution, specified as a nonnegative scalar.

In general, during closed-loop simulation, you do not specify Slack0 yourself. Instead, when calling `nlmpcmoveCodeGeneration`, return the `newOnlineData` output argument, which contains updated Slack0 estimates. You can then pass `newOnlineData` in as the `onlineData` input argument to `nlmpcmoveCodeGeneration` for the next control interval.

md — Measured disturbance values
row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your controller has measured disturbances, you must specify md. If your controller has no measured disturbances, then `getCodeGenerationData` omits this field.

To use the same disturbance values across the prediction horizon, specify a row vector.

To vary the disturbance values over the prediction horizon from time k to time $k+p$, specify an array with up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains

the disturbance values for one prediction horizon step. If you specify fewer than *p* rows, the values in the final row are used for the remaining steps of the prediction horizon.

Parameters — Parameter values
cell vector

Parameter values used by the prediction model, custom cost function, and custom constraints, specified as a cell vector with length equal to the `Model.NumberOfParameters` property of the controller. If the controller has no parameters, then `getCodeGenerationData` omits this field.

The order of the parameters must match the order defined for the prediction model, custom cost function, and custom constraints.

OutputWeights — Output variable tuning weights
row vector | array

Output variable tuning weights that replace the default tuning weights at run time, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. If you expect your output variable weights to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same weights across the prediction horizon, specify a row vector.

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

MVWeights — Manipulated variable tuning weights
row vector | array

Manipulated variable tuning weights that replace the default tuning weights at run time, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. If you expect your manipulated variable weights to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same weights across the prediction horizon, specify a row vector.

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

MVRateWeights — Manipulated variable rate tuning weights
row vector | array

Manipulated variable rate tuning weights that replace the default tuning weights at run time, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. If you expect your manipulated variable rate weights to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same weights across the prediction horizon, specify a row vector.

To vary the weights over the prediction horizon from time *k*+1 to time *k*+*p*, specify an array with up to *p* rows. Here, *k* is the current time and *p* is the prediction horizon. Each row contains the weights

for one prediction horizon step. If you specify fewer than p rows, the final weights are used for the remaining steps of the prediction horizon.

ECRWeight — Slack variable tuning weight
positive scalar

Slack variable rate tuning weight that replaces the default tuning weight at run time, specified as a positive scalar. If you expect your slack variable weight to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

OutputMin — Output variable lower bounds
row vector | array

Output variable lower bounds that replace the default lower bounds at run time, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. If you expect your output variable lower bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

OutputMax — Output variable upper bounds
row vector | array

Output variable upper bounds that replace the default upper bounds at run time, specified as a row vector of length N_y or an array with N_y columns, where N_y is the number of output variables. If you expect your output variable upper bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

StateMin — State lower bounds
row vector | array

State lower bounds that replace the default lower bounds at run time, specified as a row vector of length N_x or an array with N_x columns, where N_x is the number of states. If you expect your state lower bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

StateMax — State upper bounds
row vector | array

State upper bounds that replace the default upper bounds at run time, specified as a row vector of length N_x or an array with N_x columns, where N_x is the number of states. If you expect your state upper bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

MVMin — Manipulated variable lower bounds
row vector | array

Manipulated variable lower bounds that replace the default lower bounds at run time, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. If you expect your manipulated variable lower bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

MVMax — Manipulated variable upper bounds
row vector | array

Manipulated variable upper bounds that replace the default upper bounds at run time, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. If you expect your manipulated variable upper bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

MVRateMin — Manipulated variable rate lower bounds
row vector | array

Manipulated variable rate lower bounds that replace the default lower bounds at run time, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. If you expect your manipulated variable rate lower bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

MVRateMax — Manipulated variable rate upper bounds
row vector | array

Manipulated variable rate upper bounds that replace the default upper bounds at run time, specified as a row vector of length N_{mv} or an array with N_{mv} columns, where N_{mv} is the number of manipulated variables. If you expect your manipulated variable rate upper bounds to vary at run time, you must add this field to the online data structure when you call `getCodeGenerationData`.

To use the same bounds across the prediction horizon, specify a row vector.

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

Output Arguments

mv — Optimal manipulated variable control action
column vector

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

If the solver converges to a local optimum solution (`info.ExitFlag` is positive), then `mv` contains the optimal solution.

If the solver reaches the maximum number of iterations, finds a feasible suboptimal solution (`info.ExitFlag = 0`) and:

- `coredata.usesuboptimalsolution` is `true`, then `mv` contains the suboptimal solution
- `coredata.usesuboptimalsolution` is `false`, then `mv` contains `lastMV`

If the solver fails to find a feasible solution (`info.ExitFlag` is negative), then `mv` contains `lastMV`.

newOnlineData — Updated online controller data
structure

Updated online controller data, returned as a structure. This structure is the same as `onlineData`, except that the decision variable initial guesses (`X0`, `MV0`, and `Slack0`) are updated.

For subsequent control intervals, *warm start* the solver by modifying the online data in `newOnlineData` and passing the updated structure to `nlmpcmoveCodeGeneration` as `onlineData`. Doing so allows the solver to use the decision variable initial guesses as a starting point for its solution.

info — Solution details
structure

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

MVopt — Optimal manipulated variable sequence
array

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

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

Xopt — Optimal prediction model state sequence
array

Optimal prediction model state sequence, returned as a $(p+1)$-by-N_x array, where p is the prediction horizon and N_x is the number of states in the prediction model.

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

Yopt — Optimal output variable sequence
array

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

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

Topt — Prediction horizon time sequence
column vector

Prediction horizon time sequence, returned as a column vector of length $p+1$, where p is the prediction horizon. `Topt` contains the time sequence from time k to time $k+p$, where k is the current time.

`Topt(1) = 0` represents the current time. Subsequent time steps `Topt(i)` are `Ts*(i-1)`, where `Ts` is the controller sample time.

Use `Topt` when plotting the `MVopt`, `Xopt`, or `Yopt` sequences.

Slack — Slack variable at optimum
nonnegative scalar

Slack variable at optimum, ε, used in constraint softening, returned as a nonnegative scalar value.

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

ExitFlag — Optimization exit code
integer

Optimization exit code, returned as one of the following:

- Positive Integer — Optimal solution found
- `0` — Feasible suboptimal solution found after the maximum number of iterations
- Negative integer — No feasible solution found

`Iterations` — Number of iterations
positive integer

Number of iterations used by the solver, returned as a positive integer.

`Cost` — Objective function cost
nonnegative scalar

Objective function cost, returned as a nonnegative scalar value. The cost quantifies the degree to which the controller has achieved its objectives.

The cost value is only meaningful when `ExitFlag` is nonnegative.

Version History
Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `nlmpcmoveCodeGeneration` supports generating code only for nonlinear MPC controllers that use the default `fmincon` solver with the SQP algorithm. However, you can simulate controllers using other `fmincon` algorithms.
- When used for code generation, nonlinear MPC controllers do not support anonymous functions for the prediction model, custom cost function, or custom constraint functions. However, `nlmpcmoveCodeGeneration` can still simulate controllers that use anonymous functions.
- Your custom functions must be on the MATLAB path and compatible with MATLAB Coder. For more information on checking compatibility, see "Check Code by Using the Code Generation Readiness Tool" (MATLAB Coder).
- Code generation for nonlinear MPC controllers supports only double-precision data.
- To generate code for computing optimal control moves for a nonlinear MPC controller:

  1 Generate data structures from a nonlinear MPC controller using `getCodeGenerationData`.
  2 To verify that your controller produces the expected closed-loop results, simulate it using `nlmpcmoveCodeGeneration` in place of `nlmpcmove`.
  3 Generate code for `nlmpcmoveCodeGeneration` using `codegen`. This step requires MATLAB Coder software.

See Also

Functions
nlmpcmove | getSimulationData | getCodeGenerationData | buildMEX | codegen

Objects
nlmpc | nlmpcMultistage | nlmpcmoveopt

Topics
"Nonlinear MPC"
"Generate Code and Deploy Controller to Real-Time Targets"

plot

Plot responses generated by MPC simulations

Syntax

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

Description

Use the Model Predictive Control Toolbox `plot` function to plot responses generated by MPC simulations.

To create 2-D line plots of data points instead, see `plot`.

`plot(mpcobj,t,y,r,u,v,d)` plots the results of a simulation based on the MPC object `mpcobj`.

Examples

Plot Responses from MPC Simulation

Create a plant, a corresponding MPC object, and convert it to zero/pole/gain form.

```
mpcverbosity off;                       % turn off mpc messaging
plant=tf(1,[1 -1 1],0.2);               % create plant (0.2 seconds sampling time)
mpcobj=mpc(plant,0.2);                  % create mpc object (0.2 second sampling time)
[y,t,u,xp]=sim(mpcobj,10,1);            % simulate closed loop for 10 steps

plot(mpcobj,t,y,ones(size(y)),u);       % plot response

% You can plot other data. The signal type definitions and labels are contained in mpcobj
plot(mpcobj,1:10,rand(10,1),zeros(10,1),sin(1:10)');                  % random response
```

Input Arguments

mpcobj — Model predictive controller
mpc object

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

t — Time sequence
double vector

Time sequence, specified as an Nt-by-1 array, where Nt is the number of simulation steps.

Example: `1:10`

y — Sequence of plant outputs values
double array

Sequence of plant outputs values, specified as an array of output responses of size `Nt`-by-`Ny`, where `Ny` is the number of measured outputs of the plant.

Example: `rand(10,1)`

r — Sequence of reference values
`double` array

Sequence of reference values for the plant output. It is an array of setpoints and has the same size as `y`.

Example: `ones(10,1)`

u — Sequence of manipulated variables
`double` array

Sequence of manipulated variables, specified as an array of manipulated variable inputs of size `Nt`-by-`Nu`, where `Nu` is the number of manipulated variables.

Example: `sin(1:10)'`

v — Sequence of measured disturbances inputs
`[]` (default) | `double` array

Sequence of measured disturbances input, specified as a matrix of size `Nt`-by-`Nv`, where `Nv` is the number of measured disturbance inputs.

Example: `zeros(10,1)`

d — Sequence of unmeasured disturbances inputs
`[]` (default) | `double` array

Sequence of unmeasured disturbances inputs, specified as an array of size `Nt`-by-`Nd`, where `Nd` is the number of unmeasured disturbances inputs.

Example: `zeros(10,1)`

Version History
Introduced before R2006a

See Also

Functions
`sim`

Objects
`mpc` | `mpcsimopt`

Topics
"Model Predictive Control of Multi-Input Single-Output Plant"

plotSection

Visualize explicit MPC control law as 2-D sectional plot

Syntax

```
plotSection(empcobj,plotParams)
```

Description

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

Examples

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

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

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

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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

Define the parameter bounds for generating an explicit MPC controller.

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
range.State.Min(:) = [-4;-4];
range.State.Max(:) = [ 4; 4];
range.Reference.Min(:) = -2;
range.Reference.Max(:) = 2;
range.ManipulatedVariable.Min(:) = -1.1;
range.ManipulatedVariable.Max(:) = 1.1;
```

Create an explicit MPC controller.

```
empcobj = generateExplicitMPC(mpcobj,range);
```

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

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

```
plotParams = generatePlotParameters(empcobj);
```

Allow the controller states to vary when creating a plot.

```
plotParams.State.Index = [];
plotParams.State.Value = [];
```

Fix the manipulated variable and reference signal to 0 for plotting.

```
plotParams.ManipulatedVariable.Index(1) = 1;
plotParams.ManipulatedVariable.Value(1) = 0;
plotParams.Reference.Index(1) = 1;
plotParams.Reference.Value(1) = 0;
```

Generate the 2-D section plot for the explicit MPC controller.

```
plotSection(empcobj,plotParams)
```


2-D Plot of Explicit MPC Polyhedral Partition

```
ans = 
  Figure (1: PiecewiseAffineSectionPlot) with properties:

      Number: 1
        Name: 'PiecewiseAffineSectionPlot'
       Color: [1 1 1]
    Position: [360 402 560 420]
```

```
      Units: 'pixels'

  Show all properties
```

Input Arguments

empcobj — Explicit MPC controller
explicit MPC controller object

Explicit MPC controller for which you want to create a 2-D sectional plot, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

plotParams — Parameters for sectional plot
structure

Parameters for sectional plot of explicit MPC control law, specified as a structure. Use `generatePlotParameters` to create an initial structure in which all the parameters of the controller are fixed at their nominal values. Then, modify this structure as necessary before invoking `plotSection`. See `generatePlotParameters` for more information.

Version History
Introduced in R2014b

See Also

Functions
generatePlotParameters | generateExplicitMPC | generateExplicitRange | generateExplicitOptions | simplify | mpcmoveExplicit | sim

Objects
mpc | explicitMPC | mpcstate

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"
"What is Model Predictive Control?"

review

Examine MPC controller for design errors and stability problems at run time

Syntax

```
review(mpcobj)
```

```
results = review(mpcobj)
```

Description

`review(mpcobj)` checks for potential design issues in the model predictive controller with defined sample time, `mpcobj`, and generates a testing report. The testing report provides information about each test, highlights test warnings and failures, and suggests possible solutions. For more information on the tests performed by the `review` function, see "Algorithms" on page 2-212.

`results = review(mpcobj)` returns the test results and suppresses the testing report.

Examples

Examine MPC Controller for Design Errors or Stability Problems

Define a plant model, and create an MPC controller.

```
plant = tf(1,[10 1]);
mpcobj = mpc(plant,2);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Set hard upper and lower bounds on the manipulated variable and its rate of change.

```
MV = mpcobj.MV;
MV.Min = -2;
MV.Max =  2;
MV.RateMin = -4;
MV.RateMax =  4;
mpcobj.MV = MV;
```

Review the controller design. The review function generates and opens a report in the Web Browser window.

```
review(mpcobj)

-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Design Review for Model Predictive Controller "mpcobj"

Summary of Performed Tests

Test	Status
MPC Object Creation	Pass
QP Hessian Matrix Validity	Pass
Closed-Loop Internal Stability	Pass
Closed-Loop Nominal Stability	Pass
Closed-Loop Steady-State Gains	Pass
Hard MV Constraints	Warning
Other Hard Constraints	Pass
Soft Constraints	Pass
Memory Size for MPC Data	Pass

`review` flags a potential constraint conflict that could result if this controller was used to control a real process. To view details about the warning, click **Hard MV Constraints**.

Hard MV Constraints

The controller should always satisfy hard bounds on a manipulated variable *OR* its rate-of-change. If you specify both constraint types simultaneously, however, they might conflict during real-time use.

For example, if an event pushes an MV outside a specified hard bound and the hard MV rate bounds are too small, the resulting QP will be *infeasible*.

Avoid such conflicts by specifying hard MV bounds *OR* hard MV rate bounds, but not both. Or if you want to specify both, soften the lower-priority constraint by setting its ECR to a value greater than zero.

Warning: your constraint definitions may conflict. The following table lists potential conflicts for each MV. The tabular entries show the location of each conflict in the prediction horizon and the type of conflict.

MV name	Horizon k	Conflict Type
MV1	1	Min & RateMax
MV1	1	Max & RateMin

Return to list of tests

Obtain Test Results and Suppress Testing Report

Define a plant model with no direct feedthrough.

2-209

```
plant = rss(3,1,1);
plant.D = 0;
Ts = 0.1;
```

Create an MPC controller.

```
mpcobj = mpc(plant,Ts);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Specify constraints for the controller.

```
MV = mpcobj.MV;
MV.Min = -2;
MV.Max =  2;
MV.RateMin = -4;
MV.RateMax =  4;
mpcobj.MV = MV;
```

Review the controller design, and suppress the testing report.

```
results = review(mpcobj)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

results = struct with fields:
           ObjectCreation: 1
            HessianMatrix: 1
        InternalStability: 1
         NominalStability: 1
               SteadyState: 1
         HardMVConstraints: 0
      HardOtherConstraints: 1
            SoftConstraints: 1
```

All of the tests passed, except for the hard MV constraints test, which generated a warning.

Obtain Test Results for Multiple Controllers

Create and review designs for gain-scheduled model predictive controllers for two plant operating conditions.

Define the model parameters.

```
M1 = 1;
M2 = 5;
k1 = 1;
k2 = 0.1;
b1 = 0.3;
```

```
b2 = 0.8;
yeq1 = 10;
yeq2 = -10;
```

Create plant models for each of the two operating conditions.

```
A1 = [0 1; -k1/M1 -b1/M1];
B1 = [0 0; -1/M1 k1*yeq1/M1];
C1 = [1 0];
D1 = [0 0];
sys1 = ss(A1,B1,C1,D1);
sys1 = setmpcsignals(sys1,'MV',1,'MD',2);

A2 = [0 1; -(k1+k2)/(M1+M2) -(b1+b2)/(M1+M2)];
B2 = [0 0; -1/(M1+M2) (k1*yeq1+k2*yeq2)/(M1+M2)];
C2 = [1 0];
D2 = [0 0];
sys2 = ss(A2,B2,C2,D2);
sys2 = setmpcsignals(sys2,'MV',1,'MD',2);
```

Design an MPC controller for each operating condition.

```
Ts = 0.2;
p = 6;
m = 2;
MPC1 = mpc(sys1,Ts,p,m);
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

```
MPC2 = mpc(sys2,Ts,p,m);
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

```
controllers = {MPC1,MPC2};
```

Review the controller designs, and store the test result structures.

```
for i = 1:2
    results(i) = review(controllers{i});
end
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Input Arguments

mpcobj — MPC controller
mpc object

MPC controller object, specified as an `mpc` object.

Output Arguments

`results` — Test results
structure

Test results, returned as a structure with the following fields:

- `ObjectCreation` — MPC object creation test
- `HessianMatrix` — QP Hessian matrix validity test
- `InternalStability` — Internal stability test
- `NominalStability` — Nominal stability test
- `SteadyState` — Closed-loop steady-state gains test
- `HardMVConstraints` — Hard MV constraints test
- `HardOtherConstraints` — Other hard constraints test
- `SoftConstraints` — Soft constraints test

For more information on the tests performed by the `review` function, see "Algorithms" on page 2-212.

The `results` structure does not contain a field for the **Memory Size for MPC Data** test.

For each test, the result is returned as one of the following:

- `1` — Pass
- `0` — Warning
- `-1` — Fail

If a given test generates a warning or fails, generate a testing report by calling `review` without an output argument. The testing report provides details about the warnings and failures, and suggests possible solutions.

Tips

- You can also review your controller design in the **MPC Designer** app. On the **Tuning** tab, in the **Analysis** section, click **Review Design**.
- Test your controller design using techniques such as simulations, since `review` cannot detect all possible performance factors.

Algorithms

The review command performs the following tests.

Test	Description
MPC Object Creation	Tests whether the controller specifications generate a valid MPC controller. If the controller is invalid, additional tests are not performed.

Test	Description
QP Hessian Matrix Validity	Tests whether the MPC quadratic programming (QP) problem for the controller has a unique solution. You must choose cost function parameters (penalty weights) and horizons such that the QP Hessian matrix is positive-definite.
Closed-Loop Internal Stability	Extracts the A matrix from the state-space realization of the unconstrained controller, and then calculate its eigenvalues. If the absolute value of each eigenvalue is less than or equal to 1 and the plant is stable, then your feedback system is internally stable.
Closed-Loop Nominal Stability	Extracts the A matrix from the discrete-time state-space realization of the closed-loop system; that is, the plant and controller connected in a feedback configuration. Then calculate the eigenvalues of A. If the absolute value of each eigenvalue is less than or equal to 1, then the nominal (unconstrained) system is stable. This test is not performed if mpcobj is set to use a custom state estimator.
Closed-Loop Steady-State Gains	Tests whether the controller forces all controlled output variables to their targets at steady state in the absence of constraints. This test is not performed if mpcobj is set to use a custom state estimator. This test is not performed if mpcobj is set to use a custom state estimator.
Hard MV Constraints	Tests whether the controller has hard constraints on both a manipulated variable and its rate of change, and if so, whether these constraints may conflict at run time.
Other Hard Constraints	Tests whether the controller has hard output constraints or hard mixed input/output constraints, and if so, whether these constraints may become impossible to satisfy at run time.
Soft Constraints	Tests whether the controller has the proper balance of hard and soft constraints by evaluating the constraint ECR parameters.
Memory Size for MPC Data	Estimates the memory size required by the controller at run time.

Alternatives

review automates certain tests that you can perform at the command line.

- To test for steady-state tracking errors, use cloffset.
- To test the internal stability of a controller, check the eigenvalues of the mpc object. Convert the mpc object to a state-space model using ss, and call isstable.

Version History
Introduced in R2011b

See Also

Functions
trim | cloffset | sensitivity | ss | tf | zpk | size | d2d | mpcverbosity | mpcprops | set | get | getname | setname | getindist | getoutdist | setindist | setoutdist | getEstimator | setEstimator | sim | trim

Objects
mpc | mpcsimopt

Blocks
MPC Controller

Topics
"Model Predictive Control of Multi-Input Single-Output Plant"
"Simulation and Code Generation Using Simulink Coder"
"What is Model Predictive Control?"
"Review Model Predictive Controller for Stability and Robustness Issues"

sensitivity

Calculate the value of a performance metric and its sensitivity to the diagonal weights of an MPC controller

Syntax

```
[J,sens] = sensitivity(mpcobj,PerfFcn,PerfWeights,Ns,r,v,SimOptions,utarget)
[J,sens] = sensitivity(mpcobj,customPerFcn,Par1,...,ParN)
```

Description

`[J,sens] = sensitivity(mpcobj,PerfFcn,PerfWeights,Ns,r,v,SimOptions,utarget)` calculates the value `J` and sensitivity `sens` of a predefined closed-loop, cumulative performance metric with respect to the diagonal weights defined in the MPC controller object `mpcobj`. You chose the shape of the performance metric, among the available options, using `PerfFcn`. The optional arguments `PerfWeights`, `Ns`, `r`, `v`, `SimOptions`, and `utarget` specify the performance metric weights, number of simulation points, reference and disturbance signals, simulation options, and manipulated variables targets, respectively. If you omit any of these arguments, then default values are used.

`[J,sens] = sensitivity(mpcobj,customPerFcn,Par1,...,ParN)` calculates the value `J` and sensitivity `sens` of the performance metric defined in the custom function `customPerFcn`, with respect to the diagonal weights defined in the MPC controller object `mpcobj`. The remaining input arguments `Par1,Par2,...,ParN` specify the value of the parameters needed by `customPerFnc`.

Examples

Calculate Value of Predefined Performance Metric and its Sensitivity to Controller Weights

Fix the random number generator seed for reproducibility.

```
rng(0)
```

Define a third-order plant model with three manipulated variables and two controlled outputs. Then create an MPC controller for the plant, with sample time of 1.

```
plant = rss(3,2,3);
plant.D = 0;
mpcobj = mpc(plant,1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Specify an integral absolute error performance function and set the performance weights. The performance weights emphasize tracking the first output variable.

```
PerfFunc = 'IAE';
PerfWts.OutputVariables = [2 0.5];
```

```
PerfWts.ManipulatedVariables = zeros(1,3);
PerfWts.ManipulatedVariablesRate = zeros(1,3);
```

Define a 20 second simulation scenario with a unit step as setpoint for the first output and zero as a setpoint for the second output.

```
Tstop = 20;
r = [1 0];
```

Calculate the closed-loop performance metric, J, and its sensitivities, sens, to the weights defined in mpcobj, for the specified simulation scenario. For this example, do not specify the last three input argument of sensitivity. This means that no disturbance signal or simulation option is used and the nominal value of the manipulated variables is kept to its default value of zero.

```
[J,sens] = sensitivity(mpcobj,PerfFunc,PerfWts,Tstop,r)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

J = 2.2977

sens = struct with fields:
              OutputVariables: [-0.1897 -0.0894]
         ManipulatedVariables: [0.0311 0.0826 0.0014]
    ManipulatedVariablesRate: [0.4268 2.3482 0.0149]
```

The positive, and relatively higher, values of the sensitivities to the first and last manipulated variable rate suggest that decreasing the corresponding weights defined in mpcobj would contribute the most to decrease the IAE performance metric defined by PerfWts. At the same time, since the sensitivity to the weight of the second manipulated variable is negative, increasing the corresponding weight would also contribute to decrease the performance metric.

Modify the manipulated variable rate weights in mpcobj and recalculate the value of the performance metric.

```
mpcobj.Weights.ManipulatedVariablesRate = [1e-2 1 1e-2];
sensitivity(mpcobj,PerfFunc,PerfWts,Tstop,r)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

ans = 2.0116
```

As expected the value of the performance metric decreased, indicating an improved tracking performance.

Calculate Value of Custom Performance Metric and its Sensitivity to Controller Weights

Define a third-order plant model with three manipulated variables and two controlled outputs. Then create an MPC controller for the plant, with sample time of 1.

```
plant = rss(3,2,3);
plant.D = 0;
mpcobj = mpc(plant,1);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Define a custom performance function and write it to a file. The function must take an MPC object as a first input argument. The simulation time and the output set point are the second and third input arguments, respectively. Internally, the function performs a closed loop simulation using the given MPC object, simulation time and set point. The norm of the difference between the set point and the output signal is then returned as the value of the performance metric (note that this norm depends on the number of simulation steps).

```
% write a function to the char vector "str"
str = ['function J = mypfun(mpcobj,T,ySetPnt)', ...
        newline, ...
        'y = sim(mpcobj,T,ySetPnt); J = norm(ySetPnt-y);', ...
        newline, ...
        'end'];

% create the function file
fid=fopen('mypfun.m','w');  % open a file for writing
fwrite(fid,str,'char');     % write "str" to the file
fclose(fid);                % close the file
```

Calculate the custom performance metric, J, and its sensitivities, sens, to the weights defined in mpcobj, using a simulation time of 10 seconds and an output setpoint of [1 1].

```
[J,sens] = sensitivity(mpcobj,'mypfun',10,[1 1])

-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

J = 1.5145

sens = struct with fields:
              OutputVariables: [-0.0507 -0.0888]
         ManipulatedVariables: [0.0013 0.0055 3.7527e-05]
     ManipulatedVariablesRate: [0.1918 1.2004 0.0034]
```

The comparatively higher values of the sensitivities to the manipulated variable rates suggest that decreasing the corresponding weights defined in mpcobj would contribute the most to decrease the custom performance metric calculated in the function mypfun.

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

PerfFcn — Performance metric function shape
`'ISE'` | `'IAE'` | `'ITSE'` | `'ITAE'`

Performance metric function shape, specified as one of the following:

*   `'ISE'` (integral squared error), for which the performance metric is

$$ J = \sum_{i=1}^{Ns} \left(\sum_{j=1}^{n_y} (w_j^y e_{yij})^2 + \sum_{j=1}^{n_u} [(w_j^u e_{uij})^2 + (w_j^{\Delta u} \Delta u_{ij})^2] \right) $$

*   `'IAE'` (integral absolute error), for which the performance metric is

$$ J = \sum_{i=1}^{Ns} \left(\sum_{j=1}^{n_y} \left| w_j^y e_{yij} \right| + \sum_{j=1}^{n_u} (\left| w_j^u e_{uij} \right| + \left| w_j^{\Delta u} \Delta u_{ij} \right|) \right) $$

*   `'ITSE'` (integral of time-weighted squared error), for which the performance metric is

$$ J = \sum_{i=1}^{Ns} i\Delta t \left(\sum_{j=1}^{n_y} (w_j^y e_{yij})^2 + \sum_{j=1}^{n_u} [(w_j^u e_{uij})^2 + (w_j^{\Delta u} \Delta u_{ij})^2] \right) $$

*   `'ITAE'` (integral of time-weighted absolute error), for which the performance metric is

$$ J = \sum_{i=1}^{Ns} i\Delta t \left(\sum_{j=1}^{n_y} \left| w_j^y e_{yij} \right| + \sum_{j=1}^{n_u} (\left| w_j^u e_{uij} \right| + \left| w_j^{\Delta u} \Delta u_{ij} \right|) \right) $$

In these 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 the manipulated variable j and its target at time interval i.

The w parameters are nonnegative performance weights defined by the structure `PerfWeights`.

Example: `'ITAE'`

PerfWeights — Performance function weights
`mpcobj.Weights` (default) | structure

Performance function weights w, specified as a structure with the following fields:

*   `OutputVariables` — n_y-element row vector that contains the w_j^y values

*   `ManipulatedVariables` — n_u-element row vector that contains the w_j^u values

*   `ManipulatedVariablesRate` — n_u-element row vector that contains the $w_j^{\Delta u}$ values

If `PerfWeights` is empty or unspecified, it defaults to the corresponding weights in `mpcobj`.

Note The performance index is not related to the quadratic cost function that the MPC controller tries to minimize by choosing the values of the manipulated variables.

One clear difference is that the performance index is based on a *closed loop* simulation running until a time that is generally different than the prediction horizon, while the MPC controller calculates the

moves which minimize its internal cost function up to the prediction horizon and in *open loop* fashion. Furthermore, even when the performance index is chosen to be of ISE type, its weights should be squared to match the weights defined in the MPC cost function.

Therefore, the performance weights and those used in the controller have different purposes; define these weights accordingly.

Ns — Number of simulation points, including zero
positive integer

Number of simulation points, including zero, specified as a positive integer. The simulation runs for `Ns-1` steps, between `0` and `Ts*(Ns-1)`.

If you omit `Ns`, the default value is the number of rows of whichever of the following arrays has the largest number of rows:

- The input argument `r`
- The input argument `v`
- The `UnmeasuredDisturbance` property of `SimOptions`, if specified
- The `OutputNoise` property of `SimOptions`, if specified
- The greatest value between the prediction horizon `mpcobj.P` and `10`.

Example: `20`

r — Reference signal
`mpcobj.Model.Nominal.Y` (default) | `double` array

Reference signal, specified as an array. This array has `ny` columns, where `ny` is the number of total (measured and unmeasured) plant outputs. `r` can have anywhere from 1 to `Ns` rows. If the number of rows is less than `Ns`, the missing rows are set equal to the last row.

If Ns is empty or unspecified, it defaults to the nominal output vector `mpcobj.Model.Nominal.Y`.

Example: `ones(20,3)`

v — Measured input disturbance signal
`mpcobj.Model.Nominal.U(md)` (default) | `double` array

Measured input disturbance signal, specified as an array. This array has `nv` columns, where `nv` is the number of measured input disturbances. `v` can have anywhere from 1 to `Ns` rows. If the number of rows is less than `Ns`, the missing rows are set equal to the last row.

If `v` is empty or unspecified, it defaults to the nominal value of the measured input disturbance, `mpcobj.Model.Nominal.U(md)`, where md is the vector containing the indices of the measured disturbance signals, as defined by `setmpcsignals`.

Example: `[zeros(50,1);ones(50,1)]`

SimOptions — Simulation options object
`[]` (default) | `mpcsimopt` object

Use a simulation options objects to specify options such as noise and disturbance signals that feed into the plant but are unknown to the controller. You can also use this object to specify an open loop scenario, or a plant model in the loop that is different from the one in `mpcobj.Model.Plant`.

For more information, see `mpcsimopt`.

`utarget` — Target for manipulated variables
`mpcobj.Model.Nominal.U` (default) | vector

The optional input `utarget` is a vector of n_u manipulated variable targets. Their defaults are the nominal values of the manipulated variables.

Example: `[0.1;0;-0.2]`

`customPerFcn` — Name of the custom performance function
character vector

Name of the custom performance function, specified as a character vector. The character vector must be different than `'ISE'`, `'IAE'`, `'ITSE'`, or `'ITAE'`, and specify the name of a file in the MATLAB path containing a custom function.

The custom function must have the following signature:

`J = customPerFcn(mpcobj,Par1,...,ParN)`

where `J` is a scalar indicating the value of the performance index `mpcobj` is an `mpc` object. The remaining arguments `Par1,...,ParN` are parameters that, if needed by `customPerFcn`, you must pass to `sensitivity` after the `customPerFcn` argument.

For example, inside `customPerFcn`, you can use `mpcobj` and, if needed, `Par1,...,ParN`, to perform a simulation and calculate `J` based on the simulation results.

Example: `'myPerFcn(mpcobj,Ts,Setpoint)'`

`Par1,...,ParN` — Values of the parameters used by the custom performance function
values of any needed parameter

Values of the parameters used by the custom performance function `customPerFcn`, specified as needed.

Example: `10,[1 1]`

Output Arguments

`J` — Performance metric for the given controller
`'double'`

Depending on the `PerfFcn` argument, this performance measure can be a function of the integral (time-weighted or not) of either the square or the absolute value or the (output and input) error. See "PerfFcn" on page 2-0 for more detail.

`sens` — Sensitivity of the performance metric
structure

This structure contains and the numerical partial derivatives of the performance measure `J` with respect to its diagonal weights. These partial derivatives, also called *sensitivities*, suggest weight adjustments that should improve performance; that is, reduce `J`.

Version History
Introduced in R2009a

See Also

Functions
sim | trim | cloffset | review | mpcprops | mpcverbosity

Objects
mpc | mpcsimopt

Blocks
MPC Controller

Topics
"Adjust Input and Output Weights Based on Sensitivity Analysis"

set

Set or modify MPC object properties

Syntax

```
set(mpcobj,Name,Value)
set(mpcobj,PropertyName)
set(mpcobj)
```

Description

Use the Model Predictive Control Toolbox `set` function to assign property values of an MPC controller (see `mpc` for background).

To implement Get/Set interface of standard MATLAB object, see "Implement Set/Get Interface for Properties".

`set(mpcobj,Name,Value)` set properties of `mpcobj` using one or more `Name,Value` pair arguments. For example, `set(mpcobj,"ControlHorizon",4)` assigns the value 3 to the `ControlHorizon` property of the MPC controller `mpcobj`.

`set(mpcobj,PropertyName)` displays admissible values for the property specified by the character vector `Propertyname`. See `mpc` for an overview of legitimate MPC property values.

`set(mpcobj)` displays all assignable properties of `mpcobj` and their admissible values.

Examples

Change Signal Types of Existing Controller

To modify the signal types for an existing MPC controller, you must simultaneously modify any controller properties that depend on the signal type configuration.

Create a plant model with two outputs, one manipulated variable, one measured disturbance, and two unmeasured disturbances.

```
plant = rss(3,2,5);
plant.D = 0;
plant = setmpcsignals(plant,MV=[1 2],MD=3,UD=[4 5]);
```

Create an MPC controller using this plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Configure the controller properties. For example, set the scaling factors for the disturbance signals.

```
mpcobj.DisturbanceVariables(1).ScaleFactor = 10;
mpcobj.DisturbanceVariables(2).ScaleFactor = 5;
mpcobj.DisturbanceVariables(3).ScaleFactor = 20;
```

Suppose you want to change the second unmeasured disturbance to be a measured disturbance. To do so, you must simultaneously update the `DisturbanceVariables` property of the controller, since the order of its entries depend on the disturbance types (measured disturbances followed by unmeasured disturbances).

Create an updated disturbance variable structure array. To do so, move the third element to be the second element.

```
DV = mpcobj.DisturbanceVariables;
DV = [DV(1) DV(3) DV(2)];
DV(2).Name = "MD2";
```

To set the internal plant model signal types, obtain the `Model` property from the controller, and modify the signal types of its `Plant` element.

```
model = mpcobj.Model;
model.Plant = setmpcsignals(model.Plant,MV=[1 2],MD=[3 5],UD=4);
```

Set the model and disturbance variable properties of the controller to their updated values.

```
set(mpcobj,Model=model,DisturbanceVariables=DV);
```

In general, it is best practice to not modify the signal types after controller creation. Instead, create and configure a new controller object with the new signal configuration.

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

PropertyName — Name of the property to be assigned
character array (default) | string

`PropertyName` can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`).

Example: `'PredictionHorizon'`

Version History

Introduced before R2006a

See Also

Functions
get | mpcprops | getname | setname | getindist | getoutdist | setindist | setoutdist | getconstraint | setconstraint | getEstimator | setEstimator | setterminal | review | sim

Objects
mpc | mpcsimopt

Blocks
MPC Controller

setconstraint

Set mixed input/output constraints for model predictive controller

Syntax

```
setconstraint(mpcobj,E,F,G)
setconstraint(mpcobj,E,F,G,V)
setconstraint(mpcobj,E,F,G,V,S)

setconstraint(mpcobj)
```

Description

`setconstraint(mpcobj,E,F,G)` specifies mixed input/output constraints of the following form for the MPC controller, `mpcobj`:

$$Eu(k + j|k) + Fy(k + j|k) \leq G + \varepsilon$$

For more information, see "Mixed Input/Output Constraints" on page 2-229.

`setconstraint(mpcobj,E,F,G,V)` adds constraints of the following form:

$$Eu(k + j|k) + Fy(k + j|k) \leq G + \varepsilon V$$

Use this syntax to specify hard custom constraints or to change the default constraint softening.

`setconstraint(mpcobj,E,F,G,V,S)` adds constraints of the following form:

$$Eu(k + j|k) + Fy(k + j|k) + Sv(k + j|k) \leq G + \varepsilon V$$

Use this syntax if your mixed input/output constraints include measured disturbances.

`setconstraint(mpcobj)` removes all mixed input/output constraints from the MPC controller.

Examples

Specify Custom Constraints on Linear Combination of Inputs and Outputs

Specify a constraint of the form $0 \leq u_2 - 2u_3 + y_2 \leq 15$ on an MPC controller.

Create a third-order plant model with three manipulated variables and two measured outputs.

```
plant = rss(3,2,3);
plant.D = 0;
```

Create an MPC controller for this plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Formulate the constraint in the required form:

$$
\begin{bmatrix} 0 & -1 & 2 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \le \begin{bmatrix} 0 \\ 15 \end{bmatrix} + \varepsilon \begin{bmatrix} 1 \\ 1 \end{bmatrix}
$$

Specify the constraint matrices.

```
E = [0 -1 2;0 1 -2];
F = [0 -1;0 1];
G = [0;15];
```

Set the constraints in the MPC controller.

```
setconstraint(mpcobj,E,F,G)
```

Specify Custom Hard Constraints for MPC Controller

Create a third-order plant model with two manipulated variables and two measured outputs.

```
plant = rss(3,2,2);
plant.D = 0;
```

Create an MPC controller for this plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Assume that you have two hard constraints.

$$u_1 + u_2 \le 5$$

$$y_1 + y_2 \le 10$$

Specify the constraint matrices.

```
E = [1 1; 0 0];
F = [0 0; 1 1];
G = [5;10];
```

Specify the constraints as hard by setting V to zero for both constraints.

```
V = [0;0];
```

Set the constraints in the MPC controller.

```
setconstraint(mpcobj,E,F,G,V)
```

Specify Custom Constraints for MPC Controller with Measured Disturbances

Create a third-order plant model with two manipulated variables, two measured disturbances, and two measured outputs.

```
plant = rss(3,2,4);
plant.D = 0;
plant = setmpcsignals(plant,mv=[1 2],md=[3 4]);
```

Create an MPC controller for this plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Assume that you have three soft constraints.

$$u_1 + u_2 \le 5$$

$$y_1 + v_1 \le 10$$

$$y_2 + v_2 \le 12$$

Specify the constraint matrices.

```
E = [1 1; 0 0; 0 0];
F = [0 0; 1 0; 0 1];
G = [5;10;12];
S = [0 0; 1 0; 0 1];
```

Set the constraints in the MPC controller using the default value for V.

```
setconstraint(mpcobj,E,F,G,[],S)
```

Remove All Custom Constraints from MPC Controller

Define a plant model.

```
plant = rss(3,2,2);
plant.D = 0;
```

Create an MPC controller.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Define controller custom constraints.

```
E = [-1 2; 1 -2];
F = [0 1; 0 -1];
G = [0; 10];
setconstraint(mpcobj,E,F,G)
```

Remove the custom constraints.

```
setconstraint(mpcobj)

-->Removing mixed input/output constraints.
```

Input Arguments

`mpcobj` — Model predictive controller
`mpc` object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

E — Manipulated variable constraint constant
array of zeros (default) | N_c-by-N_{mv} array

Manipulated variable constraint constant, specified as an N_c-by-N_{mv} array, where N_c is the number of constraints, and N_{mv} is the number of manipulated variables.

F — Controlled output constraint constant
array of zeros (default) | N_c-by-N_y array

Controlled output constraint constant, specified as an N_c-by-N_y array, where N_y is the number of controlled outputs (measured and unmeasured).

G — Mixed input/output constraint constant
column vector of zeros (default) | column vector of length N_c

Mixed input/output constraint constant, specified as a column vector of length N_c.

V — Constraint softening constant
column vector of ones (default) | specified as a column vector of length N_c

Constraint softening constant representing the equal concern for the relaxation (ECR), specified as a column vector of length N_c.

If V is not specified, a default value of 1 is applied to all constraint inequalities and all constraints are soft. This behavior is the same as the default behavior for output bounds, as described in "Standard Cost Function".

To make the ith constraint hard, specify $V(i) = 0$.

To make the ith constraint soft, specify $V(i) > 0$ in keeping with the constraint violation magnitude you can tolerate. The magnitude violation depends on the numerical scale of the variables involved in the constraint.

In general, as $V(i)$ decreases, the controller hardens the constraints by decreasing the constraint violation that is allowed.

Note If a constraint is difficult to satisfy, reducing its $V(i)$ value to make it harder can be counterproductive. Doing so can lead to erratic control actions, instability, or failure of the QP solver that determines the control action.

S — Measured disturbance constraint constant
array of zeros (default) | N_c-by-N_{md} array

Measured disturbance constraint constant, specified as an N_c-by-N_{md} array, where N_{md} is the number of measured disturbances.

Tips

- The outputs, y, are being predicted using a model. If the model is imperfect, there is no guarantee that a constraint can be satisfied.
- Since the MPC controller does not optimize $u(k + p|k)$, the last constraint at time $k + p$ assumes that $u(k+p|k) = u(k+p-1|k)$.
- When simulating an MPC controller, you can update the E, F, G, and S constraint arrays at run time. For more information, see "Update Constraints at Run Time".

Algorithms

Mixed Input/Output Constraints

The general form of the mixed input/output constraints is:
$$Eu(k + j) + Fy(k + j) + Sv(k + j) \le G + \varepsilon V$$

Here, $j = 0,...,p$, and:

- p is the prediction horizon.
- k is the current time index.
- u is a column vector manipulated variables.
- y is a column vector of all plant output variables.
- v is a column vector of measured disturbance variables.
- ε is a scalar slack variable used for constraint softening (as in "Standard Cost Function").
- E, F, G, V, and S are constant matrices.

Version History
Introduced in R2011a

See Also

Functions
getconstraint | get | set | mpcprops | setterminal

Objects
mpc

Topics
"Design MPC Controller at the Command Line"
"Constraints on Linear Combinations of Inputs and Outputs"
"Update Constraints at Run Time"
"What is Model Predictive Control?"

setCustomSolver

Configures an MPC object to use the QP solver from Optimization Toolbox as a custom solver

Syntax

```
setCustomSolver(mpcobj,'quadprog')
setCustomSolver(mpcobj,'none')
```

Description

setCustomSolver(mpcobj,'quadprog') configures mpcobj to use quadprog from Optimization Toolbox™ as a custom QP solver for both simulation and code generation. Specifically, this syntax generates, in the current folder, the files mpcCustomSolver.m and mpcCustomSolverCodeGen.m, which internally call the active-set quadprog solver. It then sets mpcobj.Optimizer.CustomSolver and mpcobj.Optimizer.CustomSolverCodeGen to true.

setCustomSolver(mpcobj,'none') sets mpcobj.Optimizer.CustomSolver and mpcobj.Optimizer.CustomSolverCodeGen to false, thereby reverting mpcobj back to use the built-in algorithm specified in mpcobj.Optimizer.Algorithm for both simulation and code generation.

Examples

Set quadprog as Custom MPC Solver Using setCustomSolver

This example shows how to use the setCustomSolver function to automatically configure an mpc object to use the Optimization Toolbox™ quadprog function as custom MPC solver for both simulation and code generation.

Create an mpc object.

```
mpcobj = mpc(tf(1,[2 1],0.1));
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

As a default, the controller is set to use the active-set solver for both simulation and code generation.

```
mpcobj.Optimizer
```

```
ans = struct with fields:
                Algorithm: 'active-set'
          ActiveSetOptions: [1x1 struct]
      InteriorPointOptions: [1x1 struct]
       MixedIntegerOptions: [1x1 struct]
              MinOutputECR: 0
     UseSuboptimalSolution: 0
```

```
            CustomSolver: 0
     CustomSolverCodeGen: 0
```

Configure `mpcobj` to Use `quadprog` as Custom Solver

To set the `quadprog` function as custom MPC solver for both simulation and code generation, call `setCustomSolver` with `'quadprog'` as second argument.

```
setCustomSolver(mpcobj,'quadprog')
```

The function generates, in the current folder, the files `mpcCustomSolver.m` and `mpcCustomSolverCodeGen.m`. To display the MATLAB files in the current folder, use the `ls` command.

```
ls *.m
```

```
mpcCustomSolver.m          mpcCustomSolverCodeGen.m
```

To display the content of `mpcCustomSolver.m`, and `mpcCustomSolverCodeGen.m` use the `type` command. Both files internally call `quadprog`, which is configured to use the active-set solver, as other algorithms are not supported.

```
type mpcCustomSolver
```

```
function [x, status] = mpcCustomSolver(H, f, A, b, x0)
% "mpcCustomSolver" enables using "quadprog" from Optmization Toolbox
% as a custom QP solver with linear MPC controller for simulation.

%% Specify solver algorithm and options
options = optimoptions('quadprog','Algorithm','active-set');
if coder.target('MATLAB')
    options.Display = 'none';
end
%% Process solver inputs
% Use -A and -b in "quadprog" because MPC QP uses Ax>=b instead
A_custom = -A;
b_custom = -b;
% ensure Hessian is symmetric
H = (H+H')/2;
%% Call "quadprog"
[x, ~, exitflag, output] = quadprog(H, f, A_custom, b_custom, [], [], [], [], x0, options);
%% Converts exit flag to MPC "status"
switch exitflag
    case 1
        status = output.iterations;
    case 0
        status = 0;
    case -2
        status = -1;
    otherwise
        status = -2;
end
%% If "quadprog" fails to find a solution, set x to the initial guess
if status <= 0
    x = x0;
end
```

```
type mpcCustomSolverCodeGen.m
```

```
function [x, status] = mpcCustomSolverCodeGen(H, f, A, b, x0)
% "mpcCustomSolverCodeGen" enables using "quadprog" from Optmization
% Toolbox as a custom QP solver with linear MPC controller for code generation.

%#codegen
%% Specify solver algorithm (must be "active-set") and options
options = optimoptions('quadprog','Algorithm','active-set');
if coder.target('MATLAB')
    options.Display = 'none';
end
%% Process solver inputs
% Use -A and -b in "quadprog" because MPC QP uses Ax>=b instead
A_custom = -A;
b_custom = -b;
% ensure Hessian is symmetric
H = (H+H')/2;
%% Call "quadprog"
[x, ~, exitflag, output] = quadprog(H, f, A_custom, b_custom, [], [], [], [], x0, options);
%% Converts exit flag to MPC "status"
switch exitflag
    case 1
        status = output.iterations;
    case 0
        status = 0;
    case -2
        status = -1;
    otherwise
        status = -2;
end
%% If "quadprog" fails to find a solution, set x to the initial guess
if status <= 0
    x = x0;
end
```

The setCustomSolver function also sets mpcobj.Optimizer.CustomSolver and
mpcobj.Optimizer.CustomSolverCodeGen to true, thereby setting up the mpcobj object to use
the custom solver in the related files for simulation and code generation.

mpcobj.Optimizer

```
ans = struct with fields:
                  Algorithm: 'active-set'
           ActiveSetOptions: [1x1 struct]
       InteriorPointOptions: [1x1 struct]
        MixedIntegerOptions: [1x1 struct]
               MinOutputECR: 0
      UseSuboptimalSolution: 0
               CustomSolver: 1
        CustomSolverCodeGen: 1
```

Revert mpcobj to Use a Built-In Solver

To revert mpcobj back to use a built in solver, call setCustomSolver function with 'none' as
second argument.

```
setCustomSolver(mpcobj,'none')
```

This sets `mpcobj.Optimizer.CustomSolver` and `mpcobj.Optimizer.CustomSolverCodeGen` to `false`.

```
mpcobj.Optimizer
```

```
ans = struct with fields:
                  Algorithm: 'active-set'
           ActiveSetOptions: [1x1 struct]
       InteriorPointOptions: [1x1 struct]
         MixedIntegerOptions: [1x1 struct]
               MinOutputECR: 0
      UseSuboptimalSolution: 0
               CustomSolver: 0
        CustomSolverCodeGen: 0
```

The controller will now use the active-set built in solver for both simulation and code generation.

Input Arguments

mpcobj — MPC controller
MPC controller object

MPC controller, specified as an MPC controller object. Use the `mpc` command to create the MPC controller.

Version History
Introduced in R2021b

See Also

Functions
mpcqpsolver | mpcqpsolverOptions | mpcInteriorPointSolver | mpcInteriorPointOptions | mpcActiveSetSolver | mpcActiveSetOptions | quadprog

Objects
mpc

Topics
"Simulate MPC Controller with a Custom QP Solver"
"Simulate and Generate Code for MPC Controller with Custom QP Solver"
"Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver"
"Optimization Problem"
"QP Solvers"

setEstimator

Modify a model predictive controller's state estimator

Syntax

```
setEstimator(mpcobj,L,M)
setEstimator(mpcobj,'default')
setEstimator(mpcobj,'custom')
```

Description

`setEstimator(mpcobj,L,M)` sets the gain matrices used for estimation of the states of an MPC controller. For more information, see "State Estimator Equations" on page 2-237.

`setEstimator(mpcobj,'default')` restores the gain matrices `L` and `M` to their default values. The default values are the optimal static gains calculated using `kalmd` for the plant, disturbance, and measurement noise models specified in `mpcobj`.

`setEstimator(mpcobj,'custom')` specifies that controller state estimation will be performed by a user-supplied procedure. This option suppresses calculation of `L` and `M`. When the controller is operating in this way, the procedure must supply the state estimate `x[n|n]` to the controller at the beginning of each control interval.

Examples

Design State Estimator by Pole Placement

Design an estimator using pole placement, assuming the linear system *AM* = *L* is solvable.

Create a plant model.

```
G = tf({1,1,1},{[1 .5 1],[1 1],[.7 .5 1]});
```

To improve the clarity of this example, call `mpcverbosity` to suppress messages related to working with an MPC controller.

```
old_status = mpcverbosity('off');
```

Create a model predictive controller for the plant. Specify the controller sample time as 0.2 seconds.

```
mpcobj = mpc(G, 0.2);
```

Obtain the default state estimator gain.

```
[~,M,A1,Cm1] = getEstimator(mpcobj);
```

Calculate the default observer poles.

```
e = eig(A1-A1*M*Cm1);
abs(e)
```

```
ans = 6×1

    0.9402
    0.9402
    0.8816
    0.8816
    0.7430
    0.9020
```

Specify faster observer poles.

```
new_poles = [.8 .75 .7 .85 .6 .81];
```

Compute a state-gain matrix that places the observer poles at `new_poles`.

```
L = place(A1',Cm1',new_poles)';
```

`place` returns the controller-gain matrix, whereas you want to compute the observer-gain matrix. Using the principle of duality, which relates controllability to observability, you specify the transpose of `A1` and `Cm1` as the inputs to `place`. This function call yields the observer gain transpose.

Obtain the estimator gain from the state-gain matrix.

```
M = A1\L;
```

Specify `M` as the estimator for `mpcobj`.

```
setEstimator(mpcobj,L,M)
```

The pair, (A_1, C_{m1}), describing the overall state-space realization of the combination of plant and disturbance models must be observable for the state estimation design to succeed. Observability is checked in Model Predictive Control Toolbox software at two levels: (1) observability of the plant model is checked *at construction* of the MPC object, provided that the model of the plant is given in state-space form; (2) observability of the overall extended model is checked *at initialization* of the MPC object, after all models have been converted to discrete-time, delay-free, state-space form and combined together.

Restore `mpcverbosity`.

```
mpcverbosity(old_status);
```

Input Arguments

`mpcobj` — MPC controller
MPC controller object

MPC controller, specified as an MPC controller object. Use the `mpc` command to create the MPC controller.

`L` — Kalman gain matrix for time update
A*M (default) | matrix

Kalman gain matrix for the time update, specified as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs.

If L is empty, it defaults to L = A*M, where A is the state-transition matrix.

M — Kalman gain matrix for measurement update
0 (default) | matrix

Kalman gain matrix for the measurement update, specified as a matrix. The dimensions of L are n_x-by-n_{ym}, where n_x is the total number of controller states, and n_{ym} is the number of measured outputs.

If M is omitted or empty, it defaults to a zero matrix, and the state estimator becomes a Luenberger observer.

Algorithms

State Estimator Equations

In general, the controller states are unmeasured and must be estimated. By default, the controller uses a steady-state Kalman filter that derives from the state observer. For more information, see "Controller State Estimation".

At the beginning of the kth control interval, the controller state is estimated with the following steps:

1   Obtain the following data:

- $x_c(k|k$–$1)$ — Controller state estimate from previous control interval, k–1
- $u^{act}(k$–$1)$ — Manipulated variable (MV) actually used in the plant from k–1 to k (assumed constant)
- $u^{opt}(k$–$1)$ — Optimal MV recommended by MPC and assumed to be used in the plant from k–1 to k
- $v(k)$ — Current measured disturbances
- $y_m(_k)$ — Current measured plant outputs
- B_u, B_v — Columns of observer parameter B corresponding to $u(k)$ and $v(k)$ inputs
- C_m — Rows of observer parameter C corresponding to measured plant outputs
- D_{mv} — Rows and columns of observer parameter D corresponding to measured plant outputs and measured disturbance inputs
- L, M — Constant Kalman gain matrices

Plant input and output signals are scaled to be dimensionless prior to use in calculations.

2   Revise $x_c(k|k$–$1)$ when $u^{act}(k$–$1)$ and $u^{opt}(k$–$1)$ are different.

$$x_c^{rev}(k\,|k-1) = x_c(k\,|k-1) + B_u\big[u^{act}(k-1) - u^{opt}(k-1)\big]$$

3   Compute the innovation.

$$e(k) = y_m(k) - \big[C_m x_c^{rev}(k\,|k-1) + D_{mv}v(k)\big]$$

4   Update the controller state estimate to account for the latest measurements.

$$x_c(k\,|k) = x_c^{rev}(k\,|k-1) + Me(k)$$

Then, the software uses the current state estimate $x_c(k|k)$ to solve the quadratic program at interval k. The solution is $u^{opt}(k)$, the MPC-recommended manipulated-variable value to be used between control intervals k and $k+1$.

2-237

Finally, the software prepares for the next control interval assuming that the unknown inputs, $w_{id}(k)$, $w_{od}(k)$, and $w_n(k)$ assume their mean value (zero) between times k and $k+1$. The software predicts the impact of the known inputs and the innovation as follows:

$$x_c(k+1 \mid k) = Ax_c^{rev}(k \mid k-1) + B_u u^{opt}(k) + B_v v(k) + Le(k)$$

Version History
Introduced in R2014b

See Also

Functions
getEstimator | kalman | getindist | getoutdist | get | set | sim

Objects
mpc | mpcstate | mpcsimopt | mpcmoveopt

Blocks
MPC Controller | Adaptive MPC Controller | Multiple MPC Controllers

Topics
"Custom State Estimation"
"Implement Custom State Estimator Equivalent to Built-In Kalman Filter"
"Controller State Estimation"
"MPC Prediction Models"

setindist

Modify unmeasured input disturbance model

Syntax

```
setindist(mpcobj,'model',model)
setindist(mpcobj,'integrators')
```

Description

`setindist(mpcobj,'model',model)` sets the input disturbance model used by the model predictive controller, `mpcobj`, to a custom model.

`setindist(mpcobj,'integrators')` sets the input disturbance model to its default value. Use this syntax if you previously set a custom input disturbance model and you want to change back to the default model. For more information on the default input disturbance model, see "MPC Prediction Models".

Examples

Specify Input Disturbance Model Using Transfer Functions

Define a plant model with no direct feedthrough.

```
plant = rss(3,4,4);
plant.D = 0;
```

Set the first input signal as a manipulated variable and the remaining inputs as input disturbances.

```
plant = setmpcsignals(plant,MV=1,UD=[2 3 4]);
```

Create an MPC controller for the defined plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Define disturbance models such that:

- Input disturbance 1 is random white noise with a magnitude of 2.
- Input disturbance 2 is random step-like noise with a magnitude of `0.5`.
- Input disturbance 3 is random ramp-like noise with a magnitude of 1.

```
mod1 = tf(2,1);
mod2 = tf(0.5,[1 0]);
mod3 = tf(1,[1 0 0]);
```

Construct the input disturbance model using the above transfer functions. Use a separate noise input for each input disturbance.

```
indist = [mod1 0 0; 0 mod2 0; 0 0 mod3];
```

Set the input disturbance model in the MPC controller.

```
setindist(mpcobj,model=indist)
```

View the controller input disturbance model.

```
getindist(mpcobj)
```

```
ans =

  A =
        x1    x2    x3
    x1    1     0     0
    x2    0     1     0
    x3    0   0.1     1

  B =
        Noise#1  Noise#2  Noise#3
    x1        0     0.05        0
    x2        0        0      0.1
    x3        0        0    0.005

  C =
        x1   x2   x3
    UD1   0    0    0
    UD2   1    0    0
    UD3   0    0    1

  D =
        Noise#1  Noise#2  Noise#3
    UD1       2        0        0
    UD2       0        0        0
    UD3       0        0        0

Sample time: 0.1 seconds
Discrete-time state-space model.
```

The controller converts the continuous-time transfer function model, `indist`, into a discrete-time state-space model.

Remove Input Disturbance for Particular Channel

Define a plant model with no direct feedthrough.

```
plant = rss(3,4,4);
plant.D = 0;
```

Set the first input signal as a manipulated variable and the remaining inputs as input disturbances.

```
plant = setmpcsignals(plant,MV=1,UD=[2 3 4]);
```

Create an MPC controller for the defined plant.

```
mpcobj = mpc(plant,0.1);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2 y3 y4
```

Retrieve the default input disturbance model from the controller.

```
distMod = getindist(mpcobj);

-->Converting model to discrete time.
-->The "Model.Disturbance" property is empty:
   Assuming unmeasured input disturbance #2 is integrated white noise.
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming unmeasured input disturbance #4 is integrated white noise.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
   Assuming no disturbance added to measured output #2.
   Assuming no disturbance added to measured output #3.
   Assuming no disturbance added to measured output #4.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Remove the integrator from the second input disturbance. Construct the new input disturbance model by removing the second input channel and setting the effect on the second output by the other two inputs to zero.

```
distMod = sminreal([distMod(1,1) distMod(1,3); 0 0; distMod(3,1) distMod(3,3)]);
setindist(mpcobj,'model',distMod)
```

When removing an integrator from the input disturbance model in this way, use `sminreal` to make the custom model structurally minimal.

View the input disturbance model.

```
tf(getindist(mpcobj))

ans =

  From input "UD1-wn" to output...
          0.1
   UD1:  -----
         z - 1

   UD2:  0

   UD3:  0

  From input "UD3-wn" to output...
   UD1:  0

   UD2:  0

          0.1
   UD3:  -----
         z - 1
```

```
Sample time: 0.1 seconds
Discrete-time transfer function.
```

The integrator has been removed from the second channel. The first and third channels of the input disturbance model remain at their default values as discrete-time integrators.

Set Input Disturbance Model to Default Value

Define a plant model with no direct feedthrough.

```
plant = rss(2,2,3);
plant.D = 0;
```

Set the second and third input signals as input disturbances.

```
plant = setmpcsignals(plant,MV=1,UD=[2 3]);
```

Create an MPC controller for the defined plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

Set the input disturbance model to unity gain for both channels.

```
setindist(mpcobj,model=tf(eye(2)))
```

Restore the default input disturbance model.

```
setindist(mpcobj,'integrators')
```

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use mpc.

model — Custom input disturbance model
[] (default) | ss object | tf object | zpk object

Custom input disturbance model, specified as a state-space (ss), transfer function (tf), or zero-pole-gain (zpk) model. The MPC controller converts the model to a discrete-time, delay-free, state-space model. Omitting model or specifying model as [] is equivalent to using setindist(mpcobj,'integrators').

The input disturbance model has:

- Unit-variance white noise input signals. For custom input disturbance models, the number of inputs is your choice.
- n_d outputs, where n_d is the number of unmeasured disturbance inputs defined in `mpcobj.Model.Plant`. Each disturbance model output is sent to the corresponding plant unmeasured disturbance input.

This model, in combination with the output disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and prediction errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see "MPC Prediction Models" and "Controller State Estimation".

`setindist` does not check custom input disturbance models for violations of state observability. This check is performed later in the MPC design process when the internal state estimator is constructed using commands such as `sim` or `mpcmove`. If the controller states are not fully observable, these commands generate an error.

This syntax is equivalent to `mpcobj.Model.Disturbance = model`.

Tips

- To view the current input disturbance model, use the `getindist` command.

Version History

Introduced before R2006a

See Also

Functions
getindist | getoutdist | setoutdist | getEstimator | setEstimator | get | set | review | sim

Objects
mpc | mpcsimopt

Blocks
MPC Controller

setmpcsignals

Set signal types in LTI plant model

Syntax

```
outPlant = setmpcsignals(inPlant)
outPlant = setmpcsignals(inPlant,Name,Value)
```

Description

`outPlant = setmpcsignals(inPlant)` sets the MPC signal types of `inPlant` to their default values, returning the result in `outPlant`. By default, all inputs are manipulated variables, and all outputs are measured outputs.

`outPlant = setmpcsignals(inPlant,Name,Value)` sets the MPC signal types for the input and output signals of the LTI system `inPlant`, returning the result in `outPlant`. Specify the signal types and indices using one or more name-value pair arguments. If you do not specify the type for input or output channels, they are configured as manipulated variables and output variables, respectively.

Examples

Set MPC Signal Types and Create MPC Controller

Create a four-input, two output state-space plant model. By default all input signals are manipulated variables and all outputs are measured outputs.

```
plant = rss(3,2,4);
plant.D = 0;
```

Configure the plant input/output channels such that:

- The second and third inputs are measured disturbances.
- The fourth input is an unmeasured disturbance.
- The second output is unmeasured.

```
plant = setmpcsignals(plant,MD=[2 3],UD=4,UO=2);
```

```
-->Assuming unspecified input signals are manipulated variables.
-->Assuming unspecified output signals are measured outputs.
```

Create an MPC controller.

```
mpcobj = mpc(plant,1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

Change Signal Types of Existing Controller

To modify the signal types for an existing MPC controller, you must simultaneously modify any controller properties that depend on the signal type configuration.

Create a plant model with two outputs, one manipulated variable, one measured disturbance, and two unmeasured disturbances.

```
plant = rss(3,2,5);
plant.D = 0;
plant = setmpcsignals(plant,MV=[1 2],MD=3,UD=[4 5]);
```

Create an MPC controller using this plant.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Configure the controller properties. For example, set the scaling factors for the disturbance signals.

```
mpcobj.DisturbanceVariables(1).ScaleFactor = 10;
mpcobj.DisturbanceVariables(2).ScaleFactor = 5;
mpcobj.DisturbanceVariables(3).ScaleFactor = 20;
```

Suppose you want to change the second unmeasured disturbance to be a measured disturbance. To do so, you must simultaneously update the `DisturbanceVariables` property of the controller, since the order of its entries depend on the disturbance types (measured disturbances followed by unmeasured disturbances).

Create an updated disturbance variable structure array. To do so, move the third element to be the second element.

```
DV = mpcobj.DisturbanceVariables;
DV = [DV(1) DV(3) DV(2)];
DV(2).Name = "MD2";
```

To set the internal plant model signal types, obtain the `Model` property from the controller, and modify the signal types of its `Plant` element.

```
model = mpcobj.Model;
model.Plant = setmpcsignals(model.Plant,MV=[1 2],MD=[3 5],UD=4);
```

Set the model and disturbance variable properties of the controller to their updated values.

```
set(mpcobj,Model=model,DisturbanceVariables=DV);
```

In general, it is best practice to not modify the signal types after controller creation. Instead, create and configure a new controller object with the new signal configuration.

Input Arguments

`inPlant` — Input plant model
LTI model | identified linear model

Input plant model, specified as either an LTI model or a linear System Identification Toolbox model.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `'UnmeasuredDisturbances',[2 3]` configures the second and third input arguments as measured disturbances

`ManipulatedVariables` — Manipulated variable indices
vector of positive integers

Manipulated variable indices, specified as the comma-separated pair `'ManipulatedVariables'` followed by a vector of positive integers. The maximum index value must not exceed the number of input channels in `inPlant`. The indices specified using `ManipulatedVariables`, `MeasuredDisturbances`, and `UnmeasuredDisturbances` must not overlap.

Instead of `'ManipulatedVariables'`, you can use the abbreviation `'MV'`.

`MeasuredDisturbances` — Measured disturbance indices
vector of positive integers

Measured disturbance indices, specified as the comma-separated pair `'MeasuredDisturbances'` followed by a vector of positive integers. The maximum index value must not exceed the number of input channels in `inPlant`. The indices specified using `ManipulatedVariables`, `MeasuredDisturbances`, and `UnmeasuredDisturbances` must not overlap.

Instead of `'MeasuredDisturbances'`, you can use the abbreviation `'MD'`.

`UnmeasuredDisturbances` — Unmeasured disturbance indices
vector of positive integers

Unmeasured disturbance indices, specified as the comma-separated pair `'UnmeasuredDisturbances'` followed by a vector of positive integers. The maximum index value must not exceed the number of input channels in `inPlant`. The indices specified using `ManipulatedVariables`, `MeasuredDisturbances`, and `UnmeasuredDisturbances` must not overlap.

Instead of `'UnmeasuredDisturbances'`, you can use the abbreviation `'UD'`.

`MeasuredOutputs` — Measured output indices
vector of positive integers

Measured output indices, specified as the comma-separated pair `'MeasuredOutputs'` followed by a vector of positive integers. The maximum index value must not exceed the number of output channels in `inPlant`. The indices specified using `MeasuredOutputs` and `UnmeasuredOutputs` must not overlap.

Instead of `'MeasuredOutputs'`, you can use the abbreviation `'MO'`.

UnmeasuredOutputs — Unmeasured output indices
vector of positive integers

Unmeasured output indices, specified as the comma-separated pair `'UnmeasuredOutputs'` followed by a vector of positive integers. The maximum index value must not exceed the number of output channels in `inPlant`. The indices specified using `MeasuredOutputs` and `UnmeasuredOutputs` must not overlap.

Instead of `'UnmeasuredOutputs'`, you can use the abbreviation `'UO'`.

Output Arguments

outPlant — Output plant model
linear dynamic model

Output plant model, returned as a linear dynamic model. `outPlant` has the specified input and output channel types. Otherwise, `outPlant` is identical to `inPlant`.

Tips

In general, set the plant signal types using `setmpcsignals` before creating your controller object.

If you modify the signal types of the internal plant model of an existing controller, you must ensure that the new input/output channel types are consistent with the following controller properties:

- `Weights`
- `ManipulatedVariables`
- `OutputVariables`
- `DisturbanceVariables`
- `Model.Noise`

Version History
Introduced before R2006a

See Also

Functions
get | set | getname | setname | review | sensitivity | mpcprops | ss | tf | zpk | d2d | sim

Objects
mpc | mpcsimopt

Blocks
MPC Controller

Topics
"MPC Signal Types"

"MPC Prediction Models"

setname

Set I/O signal names in MPC plant model

Syntax

```
setname(mpcobj,'input',i,name)
setname(mpcobj,'output',i,name)
```

Description

setname(mpcobj,'input',i,name) changes the name of the ith input signal in mpcobj to name. This is equivalent to mpcobj.Model.Plant.InputName{i}=name, and both commands and also update the read-only Name field of the corresponding structure in mpcobj.ManipulatedVariables (if the input is a manipulate variable), or mpcobj.DisturbanceVariables (if the input is a disturbance variable).

setname(mpcobj,'output',i,name) changes the name of the ith output signal in mpcobj to name. This is equivalent to mpcobj.Model.Plant.OutputName{i} =name, and both commands and also update the read-only Name field of the corresponding structure in mpcobj.OutputVariables.

Examples

Set names of input and output signals in MPC object

Create a plant and an MPC object, and then set the names of some input and output signals.

```
mpcverbosity off;                          % turn off mpc messages

% create plant model
plant = rss(4,4,4);                        % random state space
plant.D = 0;                               % set D matrix to zero

% set signals type in plant model
plant = setmpcsignals(plant,'MV',1,'MD',3,'UD',4,'MO',1,'UO',[3 4]);

% create MPC object
mpcobj=mpc(plant,1);                       % sampling time = 1 second
```

Set names of input signals

```
% set input signal names
setname(mpcobj,'input',1,'Jim')            % set name of first input signal
setname(mpcobj,'input',2,'Joe')            % set name of second input signal
setname(mpcobj,'input',3,'Jeff')           % set name of third input signal
```

Check names of input signals

```
% get input signal names
getname(mpcobj,'input',2)                  % get name of second input signal
```

2-249

```
ans =
    'Joe'

% alternatively
mpcobj.ManipulatedVariables(2).Name
ans =
    'Joe'

mpcobj.DisturbanceVariables(1).Name
ans =
    'Jeff'

mpcobj.Model.Plant.InputName{3}
ans =
    'Jeff'

mpcobj.Model.Plant.InputName
ans =
  4×1 cell array
    {'Jim' }
    {'Joe' }
    {'Jeff'}
    {'UD1' }
```

Set and check names of output signals

```
% set output signal names
setname(mpcobj,'output',1,'Laura')        % set name of first output signal
setname(mpcobj,'output',2,'Diana')        % set name of second output signal
setname(mpcobj,'output',3,'Emily')        % set name of third output signal

% get output signal names
getname(mpcobj,'output',2)                % get name of second input signal
ans =
    'Diana'

% alternatively
mpcobj.OutputVariables(2).Name
ans =
    'Diana'

mpcobj.Model.Plant.OutputName{2}
ans =
    'Diana'

mpcobj.Model.Plant.OutputName
ans =
  4×1 cell array
    {'Laura'}
    {'Diana'}
    {'Emily'}
    {'UO2'  }
```

Note that signals not specified with `setmpcsignals` are assumed to be measured inputs (for non-specified inputs) or measured outputs (for non-specified outputs).

Input Arguments

`mpcobj` — Model predictive controller
`mpc` object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`i` — Signal number selection
`'integer'` greater than zero

This integer specify that the name of the `i`th signal needs to be set.

Signal number to be set.

Example: 2

`name` — Name to be assigned to the specified signal
character array | string

This is the name to be assigned to the `i`th input or output signal in `mpcobj`. This does not affect whether the signal is categorized as a manipulated variable, measured or unmeasured disturbance, measured or unmeasured output.

For input signals `name` replaces the content of `mpcobj.Model.Plant.InputName{i}`, as well as the read-only `Name` field of the corresponding structure in `mpcobj.ManipulatedVariables` (if the input is a manipulate variable), or `mpcobj.DisturbanceVariables` (if the input is a disturbance variable).

For output signals `name` replaces the content of `mpcobj.Model.Plant.OutputName{i}`, as well as the read-only `Name` field of the corresponding structure in `mpcobj.OutputVariables`.

Tips

Note The `Name` fields of the variable-related structures in `ManipulatedVariables`, `OutputVariables`, and `DisturbanceVariables` in `mpcobj` 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.

Note Neither of the `Name` properties for the signals in `mpcobj` affects whether the signal is categorized as a manipulated variable, measured or unmeasured disturbance, measured or unmeasured output. To change the signal type you need to either reassign it using `setmpcsignal` on the plant object, and recreate the MPC object for that plant, or you need to recreate all the affected controller signal structures and use `set` to assign them to the MPC object (not recommended).

Version History
Introduced before R2006a

See Also

Functions
getname | setmpcsignals | set | get | mpcverbosity | review

Objects
mpc

Blocks
MPC Controller

setoutdist

Modify unmeasured output disturbance model

Syntax

```
setoutdist(mpcobj,'model',model)
setoutdist(mpcobj,'integrators')
```

Description

`setoutdist(mpcobj,'model',model)` sets the output disturbance model used by the model predictive controller, `mpcobj`, to a custom model.

`setoutdist(mpcobj,'integrators')` sets the output disturbance model to its default value. Use this syntax if you previously set a custom output disturbance model and you want to change back to the default model. For more information on the default output disturbance model, see "MPC Prediction Models".

Examples

Specify Output Disturbance Model Using Transfer Functions

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.D = 0;
mpcobj = mpc(plant,0.1);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Define disturbance models for each output such that the output disturbance for:

- Channel 1 is random white noise with a magnitude of 2.
- Channel 2 is random step-like noise with a magnitude of `0.5`.
- Channel 3 is random ramp-like noise with a magnitude of `1`.

```
mod1 = tf(2,1);
mod2 = tf(0.5,[1 0]);
mod3 = tf(1,[1 0 0]);
```

Construct the output disturbance model using these transfer functions. Use a separate noise input for each output disturbance.

```
outdist = [mod1 0 0; 0 mod2 0; 0 0 mod3];
```

Set the output disturbance model in the MPC controller.

```
setoutdist(mpcobj,model=outdist)
```

View the controller output disturbance model.

```
getoutdist(mpcobj)
```

```
ans =

  A =
        x1    x2    x3
    x1    1     0     0
    x2    0     1     0
    x3    0   0.1     1

  B =
        Noise#1  Noise#2  Noise#3
    x1        0     0.05        0
    x2        0        0      0.1
    x3        0        0    0.005

  C =
        x1  x2  x3
    MO1   0   0   0
    MO2   1   0   0
    MO3   0   0   1

  D =
        Noise#1  Noise#2  Noise#3
    MO1       2        0        0
    MO2       0        0        0
    MO3       0        0        0

Sample time: 0.1 seconds
Discrete-time state-space model.
```

The controller converts the continuous-time transfer function model, `outdist`, into a discrete-time state-space model.

Remove Output Disturbance from Particular Output Channel

Define a plant model with no direct feedthrough, and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.D = 0;
mpcobj = mpc(plant,0.1);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Retrieve the default output disturbance model from the controller.

```
distMod = getoutdist(mpcobj);
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->Assuming output disturbance added to measured output #3 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Remove the integrator from the second output channel. Construct the new output disturbance model by removing the second input channel and setting the effect on the second output by the other two inputs to zero.

```
distMod = sminreal([distMod(1,1) distMod(1,3); 0 0; distMod(3,1) distMod(3,3)]);
setoutdist(mpcobj,'model',distMod)
```

When removing an integrator from the output disturbance model in this way, use `sminreal` to make the custom model structurally minimal.

View the output disturbance model.

```
tf(getoutdist(mpcobj))
```

```
ans =

  From input "Noise#1" to output...
          0.1
    MO1:  -----
          z - 1

    MO2:  0

    MO3:  0

  From input "Noise#2" to output...
    MO1:  0

    MO2:  0

          0.1
    MO3:  -----
          z - 1

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The integrator has been removed from the second channel. The disturbance models for channels 1 and 3 remain at their default values as discrete-time integrators.

Remove Output Disturbances from All Output Channels

Define a plant model with no direct feedthrough and create an MPC controller for that plant.

```
plant = rss(3,3,3);
plant.D = 0;
mpcobj = mpc(plant,1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Set the output disturbance model to zero for all three output channels.

```
setoutdist(mpcobj,model=tf(zeros(3,1)))
```

View the output disturbance model.

```
getoutdist(mpcobj)

ans =

  D =
        Noise#1
    MO1        0
    MO2        0
    MO3        0

Static gain.
```

A static gain of `0` for all output channels indicates that the output disturbances were removed.

Set Output Disturbance Model to Default Value

Define a plant model with no direct feedthrough and create an MPC controller for that plant.

```
plant = rss(2,2,2);
plant.D = 0;
mpcobj = mpc(plant,0.1);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Remove the output disturbances for all channels.

```
setoutdist(mpcobj,model=tf(zeros(2,1)))
```

Restore the default output disturbance model.

```
setoutdist(mpcobj,'integrators')
```

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`model` — Custom output disturbance model
`[]` (default) | `ss` object | `tf` object | `zpk` object

Custom output disturbance model, specified as a state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) model. The MPC controller converts the model to a discrete-time, delay-free, state-space model. Omitting `model` or specifying `model` as `[]` is equivalent to using `setoutdist(mpcobj,'integrators')`.

The output disturbance model has:

* Unit-variance white noise input signals. For custom output disturbance models, the number of inputs is your choice.
* n_y outputs, where n_y is the number of plant outputs defined in `mpcobj.Model.Plant`. Each disturbance model output is added to the corresponding plant output.

This model, along with the input disturbance model (if any), governs how well the controller compensates for unmeasured disturbances and modeling errors. For more information on the disturbance modeling in MPC and about the model used during state estimation, see "MPC Prediction Models" and "Controller State Estimation".

`setoutdist` does not check custom output disturbance models for violations of state observability. This check is performed later in the MPC design process when the internal state estimator is constructed using commands such as `sim` or `mpcmove`. If the controller states are not fully observable, these commands will generate an error.

Tips

* To view the current output disturbance model, use the `getoutdist` command.

Version History
Introduced in R2006a

See Also

Functions
`getoutdist` | `getindist` | `setindist` | `getEstimator` | `setEstimator` | `get` | `set` | `review` | `sim`

Objects
`mpc` | `mpcsimopt`

Blocks
MPC Controller

setterminal

Terminal weights and constraints

Syntax

```
setterminal(mpcobj,Y,U)
setterminal(mpcobj,Y,U,Pt)
```

Description

`setterminal(mpcobj,Y,U)` specifies diagonal quadratic penalty weights and constraints at the last step in the prediction horizon. The weights and constraints are on the terminal output $y(t+p)$ and terminal input $u(t+p-1)$, where p is the prediction horizon of the MPC controller `mpcobj`.

`setterminal(mpcobj,Y,U,Pt)` specifies diagonal quadratic penalty weights and constraints from step Pt to the horizon end. By default, Pt is the last step in the horizon.

Examples

Specify Constraints and Penalty Weights at Last Prediction Horizon Step

Create an MPC controller for a plant with three output variables and two manipulated variables.

```
plant = rss(3,3,2);
plant.D = 0;
mpcobj = mpc(plant,0.1);

-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 y2 and zero weight for output(s) y3
```

Specify a prediction horizon of 8.

```
mpcobj.PredictionHorizon = 8;
```

Define the following penalty weights and constraints:

*   Diagonal penalty weights of 1 and 10 on the first two output variables
*   Lower bounds of 0 and -1 on the first and third outputs respectively
*   Upper bound of 2 on the second output
*   Lower bound of 1 on the first manipulated variable

```
Y = struct('Weight',[1,10,0],'Min',[0,-Inf,-1],'Max',[Inf,2,Inf]);
U = struct('Min',[1,-Inf]);
```

Specify the constraints and penalty weights at the last step of the prediction horizon.

```
setterminal(mpcobj,Y,U)
```

Specify Terminal Constraints For Final Prediction Horizon Range

Create an MPC controller for a plant with three output variables and two manipulated variables.

```
plant = rss(3,3,2);
plant.D = 0;
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
    for output(s) y1 y2 and zero weight for output(s) y3
```

Specify a prediction horizon of 10.

```
mpcobj.PredictionHorizon = 10;
```

Define the following terminal constraints:

- Lower bounds of 0 and -1 on the first and third outputs respectively
- Upper bound of 2 on the second output
- Lower bound of 1 on the first manipulated variable

```
Y = struct('Min',[0,-Inf,-1],'Max',[Inf,2,Inf]);
U = struct('Min',[1,-Inf]);
```

Specify the constraints beginning at step 5 and ending at the last step of the prediction horizon.

```
setterminal(mpcobj,Y,U,5)
```

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use mpc.

Y — Terminal weights and constraints for the output variables
structure

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

2-259

MinECR	1-by-n_y vector of constraint-softening Equal Concern for the Relaxation (ECR) values for the lower bounds
MaxECR	1-by-n_y vector of constraint-softening ECR values for the upper bounds

n_y is the number of controlled outputs of the MPC controller.

If the `Weight`, `Min` or `Max` field is empty, the values in `mpcobj` are used at all prediction horizon steps including the last. For the standard bounds, if any element of the `Min` or `Max` field is infinite, the corresponding variable is unconstrained at the terminal step.

Off-diagonal weights are zero (as described in "Standard Cost Function"). To apply nonzero off-diagonal terminal weights, you must augment the plant model. See "Provide LQR Performance Using Terminal Penalty Weights".

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
structure

Terminal weights and constraints for the manipulated variables, specified as a structure with the following fields:

Weight	1-by-n_u vector of nonnegative weights
Min	1-by-n_u vector of lower bounds
Max	1-by-n_u vector of upper bounds
MinECR	1-by-n_u vector of constraint-softening Equal Concern for the Relaxation (ECR) values for the lower bounds
MaxECR	1-by-n_u vector of constraint-softening ECR values for the upper bounds

n_u is the number of manipulated variables of the MPC controller.

If the `Weight`, `Min` or `Max` field is empty, the values in `mpcobj` are used at all prediction horizon steps including the last. For the standard bounds, if individual elements of the `Min` or `Max` fields are infinite, the corresponding variable is unconstrained at the terminal step.

Off-diagonal weights are zero (as described in "Standard Cost Function"). To apply nonzero off-diagonal terminal weights, you must augment the plant model. See "Provide LQR Performance Using Terminal Penalty Weights".

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 — Initial application step for terminal weight and constraints
prediction horizon *p* (default) | integer less than *p*

Step in the prediction horizon, specified as an integer between 1 and *p*, where *p* is the prediction horizon. The terminal weights and constraints are applied from prediction step `Pt` to the end.

Version History
Introduced in R2011a

See Also

Functions
cloffset | sensitivity | ss | tf | zpk | d2d | review | mpcprops | setconstraint | mpcverbosity | set | get

Objects
mpc

Blocks
MPC Controller

Topics
"Provide LQR Performance Using Terminal Penalty Weights"
"Terminal Weights and Constraints"

sim

Simulate an MPC controller in closed loop with a linear plant

Syntax

```
sim(mpcobj,Ns,r)
sim(mpcobj,Ns,r,v)
sim( ___ ,SimOptions)
[y,t,u,xp,xc,SimOptions] = sim( ___ )
```

Description

Use the Model Predictive Control Toolbox `sim` function to simulate, in discrete time, the closed-loop or open-loop response of a plant and an MPC controller with constraints and weights that do not change at run time. The MPC controller can be implicit or explicit, the controlled plant must be linear and time-invariant, and you must specify the reference and disturbance signals in advance. By default, the plant used in the simulation is the one in `mpcobj.Model.Plant`, but you can use a different plant model to assess the controller robustness to model mismatch.

To run simulink models programmatically instead, see `sim` (Simulink).

`sim(mpcobj,Ns,r)` simulates the closed-loop response to the specified reference signal, `r`. The simulation runs in discrete time, with sample time `mpcobj`, for `Ns-1` simulation steps, and simulation results are plotted. The plant model is the one specified in `mpcobj.Model.Plant` (which is discretized or resampled, if needed). The MPC controller `mpcobj` can be either a traditional MPC controller (`mpc`) or explicit MPC controller (`explicitMPC`). If you omit `Ns` or `r` then default values are used.

`sim(mpcobj,Ns,r,v)` also specifies the measured disturbance signal `v`.

`sim( ___ ,SimOptions)` specifies additional simulation options. This syntax allows you to alter the default simulation options, such as initial states, input/output noise, and unmeasured disturbances, plant mismatch, etc. It also allows you to simulate the plant in open loop. You can use `SimOptions` with any of the previous input combinations.

`[y,t,u,xp,xc,SimOptions] = sim( ___ )` suppresses plotting and instead returns:

- the sequence of plant outputs `y`,
- the time sequence `t` (equally spaced by `mpcobj.Ts`),
- the manipulated variables `u` generated by the MPC controller,
- the sequence `xp` of states of the model of the plant used for simulation,
- the sequence `xmpc` of states of the MPC controller (provided by the state observer),
- and the simulation options object, `SimOptions`.

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);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

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 is empty:
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Plant Output: MO1
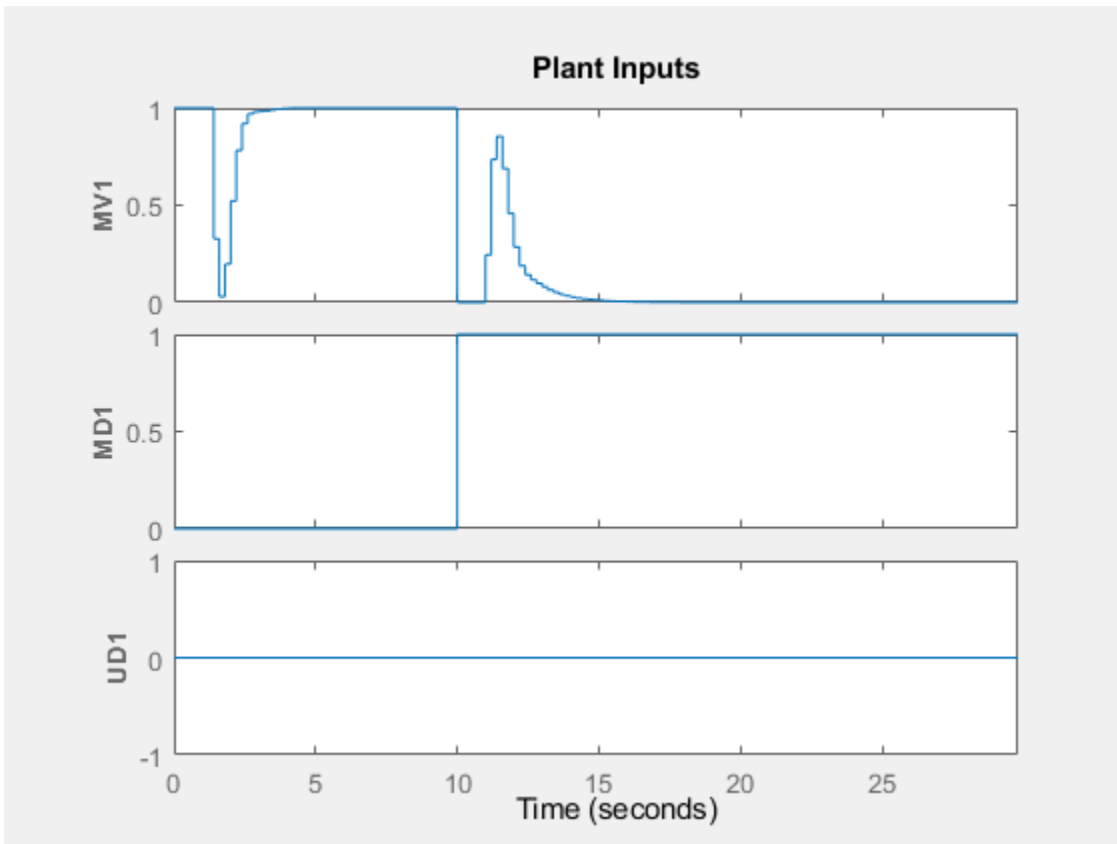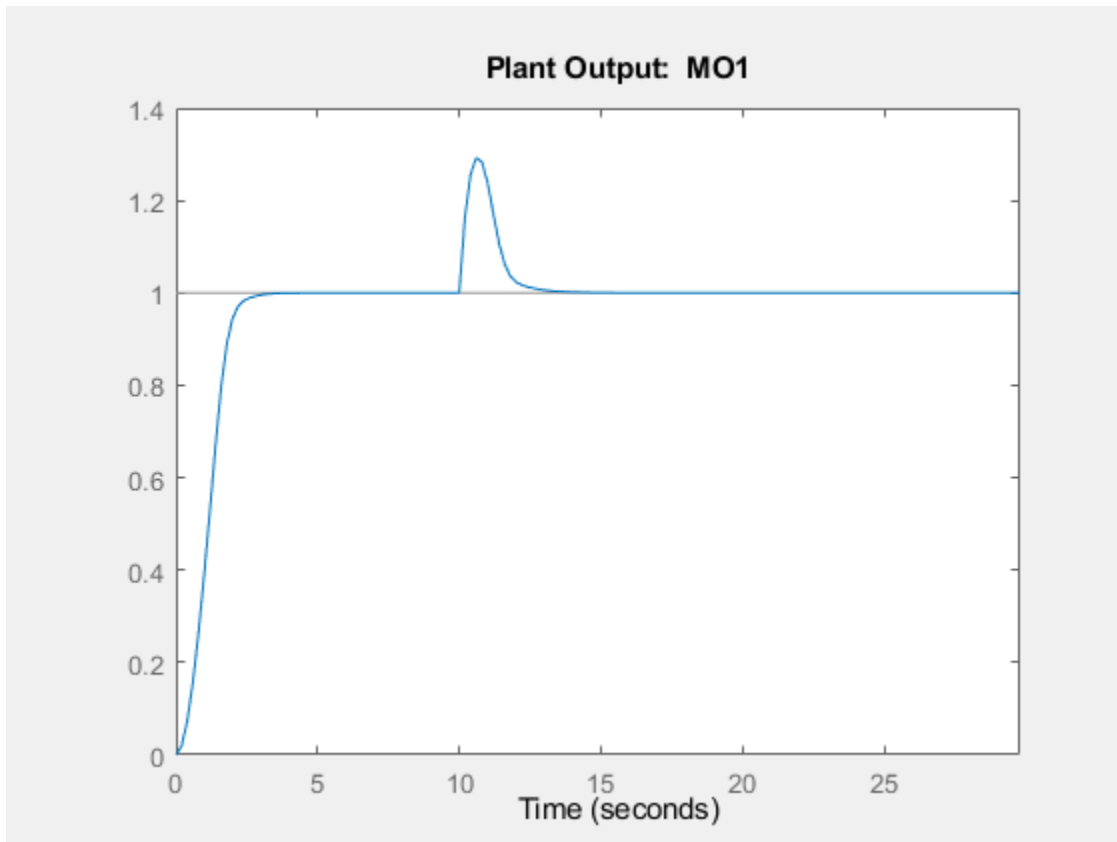
Input Arguments

`mpcobj` — Model predictive controller
mpc object | explicitMPC object

Model predictive controller, specified as one of the following:

- mpc object — Implicit MPC controller
- explicitMPC object — Explicit MPC controller created using generateExplicitMPC.

`Ns` — Number of simulation points, including zero
positive integer

Number of simulation points, including zero, specified as a positive integer. The simulation runs for Ns-1 steps, between 0 and Ts*(Ns-1).

If you omit Ns, the default value is the number of rows of whichever of the following arrays has the largest number of rows:

- The input argument r
- The input argument v
- The UnmeasuredDisturbance property of SimOptions, if specified
- The OutputNoise property of SimOptions, if specified

- The greatest value between the prediction horizon `mpcobj.P` and `10`.

Example: `20`

r — Reference signal
`mpcobj.Model.Nominal.Y` (default) | `double` array

Reference signal, specified as an array. This array has `ny` columns, where `ny` is the number of total (measured and unmeasured) plant outputs. `r` can have anywhere from 1 to `Ns` rows. If the number of rows is less than `Ns`, the missing rows are set equal to the last row.

If Ns is empty or unspecified, it defaults to the nominal output vector `mpcobj.Model.Nominal.Y`.

Example: `ones(20,3)`

v — Measured input disturbance signal
`mpcobj.Model.Nominal.U(md)` (default) | `double` array

Measured input disturbance signal, specified as an array. This array has `nv` columns, where `nv` is the number of measured input disturbances. `v` can have anywhere from 1 to `Ns` rows. If the number of rows is less than `Ns`, the missing rows are set equal to the last row.

If `v` is empty or unspecified, it defaults to the nominal value of the measured input disturbance, `mpcobj.Model.Nominal.U(md)`, where `md` is the vector containing the indices of the measured disturbance signals, as defined by `setmpcsignals`.

Example: `[zeros(50,1);ones(50,1)]`

SimOptions — Simulation options
`[]` (default) | `mpcsimopt` object

Simulation options, used to specify additional simulation options as well as noise and disturbance signals that feed into the plant but are unknown to the controller. You can also use this object to simulate the plant in open loop, or to specify a plant model to be used in simulation that is different from the one in `mpcobj.Model.Plant`, which allows you to assess the robustness of the control loop response to model mismatch.

For more information, see `mpcsimopt`.

Output Arguments

y — Sequence of plant outputs values
`double` array

Sequence of plant outputs values, returned as a `Ns`-by-`Ny` array, where `Ns` is the number of simulation steps and `Ny` is the number of plant outputs. The values in `y` include neither additive output disturbances nor additive measurement noise (if any).

t — Time sequence
`double` column vector

Time sequence, returned as a `Ns`-by-1 array, where `Ns` is the number of simulation steps. The values in `t` are equally spaced by `mpcobj.Ts`.

u — Sequence of manipulated variables
`double` array

Sequence of manipulated variables values generated by the MPC controller, returned as a `Ns`-by-`Nu` array, where `Ns` is the number of simulation steps and `Nu` is the number of manipulated variables.

xp — Sequence of plant model states values
`struct` array

Sequence of plant model states values, returned as an `Ns`-by-`Nxp` array, where `Ns` is the number of simulation steps and `Nxp` is the number of states in the plant model. The plant model is either `mpcobj.Model` or `SimOptions.Model`, if the latter is specified.

xc — Controller states
`struct` array

Sequence of MPC controller states, returned as an `Ns`-by-1 structure array. Each entry in the structure array has the same fields as an `mpcstate` object. The controller uses a built-in linear Kalman filter to estimate the state of the plant, augmented by the disturbance and noise models. The state of the controller is the state of its internal Kalman filter. For open-loop simulations, `xc` is empty.

SimOptions — Simulation options object
`mpcsimopt` object

Simulation options objects used for the simulation. This object can specify noise and disturbance signals that feed into the plant but are unknown to the controller. It can also specify if the simulated system is open loop or if the plant model used in the simulation is different from the one in `mpcobj.Model.Plant`.

For more information, see `mpcsimopt`.

Version History
Introduced before R2006a

See Also

Functions
plot | mpcmove | trim | cloffset

Objects
mpc | explicitMPC | mpcstate | mpcsimopt

Blocks
MPC Controller

Topics
"Model Predictive Control of Multi-Input Single-Output Plant"
"What is Model Predictive Control?"
"MPC Prediction Models"

simplify

Reduce explicit MPC controller complexity and memory requirements

Syntax

```
EMPCreduced = simplify(empcobj,'exact')
EMPCreduced = simplify(empcobj,'exact',uniteps)
EMPCreduced = simplify(empcobj,'radius',r)
EMPCreduced = simplify(empcobj,'sequence',index)
simplify(empcobj,___ )
```

Description

`EMPCreduced = simplify(empcobj,'exact')` attempts to reduce the number of piecewise affine (PWA) regions in an explicit MPC controller by merging regions that have identical controller gains and whose union is a convex set. Reducing the number of PWA regions reduces memory requirements of the controller. This command returns a reduced controller, `EMPCreduced`. If the second argument is omitted then it is assumed to be 'exact'.

`EMPCreduced = simplify(empcobj,'exact',uniteps)` specifies the tolerance for identifying regions that can be merged.

`EMPCreduced = simplify(empcobj,'radius',r)` retains only regions whose Chebyshev radius (the radius of the largest ball contained in the region) is larger than `r`.

`EMPCreduced = simplify(empcobj,'sequence',index)` eliminates all regions except those specified in an index vector.

`simplify(empcobj, ___ )` applies the reduction to the explicit MPC controller `empcobj`, rather than returning a new controller object. You can use this syntax with any of the previous reduction options.

Examples

Simplify Explicit MPC Controller

Define a plant model. For this example, define the plant model as a double integrator.

```
plant = tf(1,[1 0 0])                    % plant model

plant =

  1
  ---
  s^2

Continuous-time transfer function.
```

Create an MPC controller with a sampling time of 0.1 seconds, a prediction horizon or 10 steps, and a control horizon of 3 steps. Also define a constraint on the manipulated variable.

```
mpcobj = mpc(plant, 0.1, 10, 3);          % MPC controller

-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.

mpcobj.ManipulatedVariables = struct('Min',-1,'Max',1);   % hard constraint on manipulated varial
```

Create a range structure to specify the ranges for input, state, and reference signals.

```
range.ManipulatedVariable.Min = -1.1;    % input signal min
range.ManipulatedVariable.Max = 1.1;     % input signal max

range.State.Min(:) = [-10;-10];          % states min
range.State.Max(:) = [10;10];            % states max

range.Reference.Min = -2;                % reference min
range.Reference.Max = 2;                 % reference max
```

Generate an explicit MPC controller with the specified signal ranges using the generateExplicitMPC function, and display the resulting controller.

```
mpcobjExplicit = generateExplicitMPC(mpcobj,range)

-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.


Regions found / unexplored:        19/        0


Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        19
Number of parameters:      4
Is solution simplified:    No
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'mpcobjExplicit.MPC' for the original implicit MPC design.
Type 'mpcobjExplicit.Range' for the valid range of parameters.
Type 'mpcobjExplicit.OptimizationOptions' for the options used in multi-parametric QP computation
Type 'mpcobjExplicit.PiecewiseAffineSolution' for regions and gain in each solution.
```

Note that the resulting explicit controller has 19 polyhedral regions.

Use simplify to simplify the explicit MPC controller, and display the resulting controller.

```
reducedEMPC = simplify(mpcobjExplicit)

Regions to analyze:        15/        15


Explicit MPC Controller
---------------------------------------------
Controller sample time:    0.1 (seconds)
Polyhedral regions:        15
```

```
Number of parameters:      4
Is solution simplified:    Yes
State Estimation:          Default Kalman gain
---------------------------------------------
Type 'reducedEMPC.MPC' for the original implicit MPC design.
Type 'reducedEMPC.Range' for the valid range of parameters.
Type 'reducedEMPC.OptimizationOptions' for the options used in multi-parametric QP computation.
Type 'reducedEMPC.PiecewiseAffineSolution' for regions and gain in each solution.
```

Note that the simplified explicit controller has 15 polyhedral regions.

Input Arguments

empcobj — Explicit MPC controller
explicit MPC controller object

Explicit MPC controller to reduce, specified as an Explicit MPC controller object. Use `generateExplicitMPC` to create an explicit MPC controller.

uniteeps — Tolerance for joining regions
0.001 (default) | positive scalar

Tolerance for joining PWA regions, specified as a positive scalar.

r — Minimum Chebyshev radius
0 (default) | nonnegative scalar

Minimum Chebyshev radius for retaining PWA regions, specified as a nonnegative scalar. When you use the `'radius'` option, `simplify` keeps only the regions whose Chebyshev radius is larger than `r`. The default value is 0, which causes all regions to be retained.

index — Indices of PWA regions to retain
1:nr (default) | vector

Indices of PWA regions to retain, specified as a vector. The default value is `[1:nr]`, where `nr` is the number of PWA regions in `empcobj`. Thus, by default, all regions are retained. You can obtain a sequence of regions to retain by performing simulations using `empcobj` and recording the indices of regions actually encountered.

Output Arguments

EMPCreduced — Reduced MPC controller
explicit MPC controller object

Reduced MPC controller, returned as an Explicit MPC controller object.

Version History
Introduced in R2014b

See Also

Functions
generateExplicitMPC | generateExplicitRange | generateExplicitOptions | generatePlotParameters | plotSection | mpcmoveExplicit | sim

Objects
mpc | explicitMPC | mpcstate

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"
"What is Model Predictive Control?"

size

Size and order of MPC Controller

Syntax

```
mpcSize = size(mpcobj)
signalSize = size(mpcobj,SignalType)
size( ___ )
```

Description

Use the Model Predictive Control Toolbox `size` function to return size and order of an MPC controller (see `mpc` for background).

To return the dimensions of an generic array or table instead, see `size`.

`mpcSize = size(mpcobj)` returns a row vector specifying the number of manipulated variables and the number of measured plant outputs associated with `mpcobj`.

`signalSize = size(mpcobj,SignalType)` returns the number of the element of the specified signal type associated with `mpcobj`.

`size( ___ )` displays the corresponding size information for any of the previous syntaxes.

Examples

Get size of MPC controller

Create a plant, a corresponding MPC object, and get the size of the MPC signals.

```
mpcverbosity off;                          % turn off mpc messages
plant = rss(5,2,3);plant.D=0;              % random state space
mpcobj=mpc(plant,1);                       % create mpc object (1 second sampling time)

mpcSize = size(mpcobj)                     % size of the MPC controller
mpcSize =
     3     2

nMV = size(mpcobj,'MV')                    % size of manipulated variables vector
nMV =
     3
nMO = size(mpcobj,"MO")                    % size of measured output vector
nMO =
     2
nMD = size(mpcobj,'md')                    % size of measured (input) disturbance vector
nMD =
     0

size(mpcobj)                               % size of MPC controller, printout
```

```
MPC controller with 2 measured output(s), 0 unmeasured output(s),
3 manipulated input(s), 0 measured disturbance(s), 0 unmeasured disturbance(s)
```

Input Arguments

mpcobj — Model predictive controller
mpc object | explicitMPC object

Model predictive controller, specified as one of the following:

- mpc object — Implicit MPC controller
- explicitMPC object — Explicit MPC controller created using generateExplicitMPC.

SignalType — Type of the MPC signal
character array | string

You can specify SignalType as one of the following (in lower or upper case):

- 'uo' — Unmeasured controlled outputs
- 'md' — Measured disturbances
- 'ud' — Unmeasured disturbances
- 'mv' — Manipulated variables
- 'mo' — Measured controlled outputs

Example: "MV"

Output Arguments

mpcSize — Size of the MPC controller
row vector

This row vector contains the two positive integers, n_u and n_{ym}, where n_u is the number of manipulated variables (controlled plant inputs) and n_{ym} is the number of measured plant outputs.

signalSize — Size of the MPC signal
nonnegative integer

This positive integer is the number of elements of the specified signal type associated with mpcobj

Version History

Introduced before R2006a

See Also

Functions
ss | tf | zpk | d2d | trim | cloffset | sensitivity | review | get | set | setEstimator

Objects
mpc

SS

Convert unconstrained MPC controller to state-space linear system form

Syntax

```
kss = ss(mpcobj)
kssFull = ss(mpcobj,signals)
kssFullPv = ss(mpcobj,signals,refPreview,mdPreview)
[kss,ut] = ss(mpcobj)
```

Description

Use the Model Predictive Control Toolbox `ss` function to convert an unconstrained MPC controller with defined sample time to state space form (see `mpc` for background). The returned controller is equivalent to the original MPC controller `mpcobj` when no constraints are active. You can then use Control System Toolbox™ software for sensitivity analysis and other diagnostic calculations.

To create or convert a generic LTI dynamical system to state space form, see `ss` and "Dynamic System Models".

`kss = ss(mpcobj)` returns the linear discrete-time dynamic controller `kss`, in state-space form. `kss` is equivalent to the MPC controller `mpcobj` when no constraint is active.

`kssFull = ss(mpcobj,signals)` returns the linear discrete-time dynamic controller `kss`, in full state-space form, and allows you to specify the signals that you want to include as inputs for `kssFull`.

`kssFullPv = ss(mpcobj,signals,refPreview,mdPreview)` specifies whether the returned controller has preview action, that is if it uses the whole reference and measured disturbance sequences as input signals.

`[kss,ut] = ss(mpcobj)` also returns the input target values for the full form of the controller.

Examples

Convert Unconstrained MPC Controller to State-Space Model

Create the plant model.

```
plant = rss(5,2,3);
plant.D = 0;
plant = setmpcsignals(plant,'mv',1,'md',2,'ud',3,'mo',1,'uo',2);
```

Configure the MPC controller with a sample time of 0.1 seconds and nonzero nominal values, weights, and input targets.

```
mpcobj = mpc(plant,0.1);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
   for output(s) y1 and zero weight for output(s) y2
```

```
mpcobj.Model.Nominal.U = [0.7 0.8 0];
mpcobj.Model.Nominal.Y = [0.5 0.6];
mpcobj.Model.Nominal.DX = rand(5,1);

mpcobj.Weights.MV = 2;
mpcobj.Weights.OV = [3 4];

mpcobj.MV.Target = [0.1 0.2 0.3];
```

Specifying `mpcobj.Model.Nominal.DX` as nonzero means that the nominal values are not at steady state `mpcobj.MV.Target` specifies three preview steps.

Convert `mpcobj` to a state-space model.

```
Kss = ss(mpcobj)
```

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property is empty:
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
Kss =

  A =
                xMPC1       xMPC2       xMPC3       xMPC4       xMPC5       xMPC6     prev.MV1
     xMPC1        0.76      0.2216   -0.006469      0.3508     0.02977    -0.04783    3.019e-05
     xMPC2      -0.231       1.135     -0.4594      0.5914     0.01656     -0.1007    7.773e-08
     xMPC3    -0.03007      0.1185      0.5467      0.1386     0.01752       -0.11    0.0001122
     xMPC4     -0.1927     -0.2434     -0.1331      0.2121    -0.02496     -0.1596    0.0001298
     xMPC5     -0.1399     -0.2179     0.09667     -0.3748      0.5926      0.1385   -2.941e-05
     xMPC6    -0.09826           0      0.1393     -0.1346    -0.09578           1            0
   prev.MV1     -2.511     0.02726      -2.984        -1.2    -0.08729      -2.102     0.001735

  B =
                MO1
     xMPC1     0.05903
     xMPC2      0.1294
     xMPC3     0.03619
     xMPC4     -0.0589
     xMPC5    -0.08955
     xMPC6    -0.09022
   prev.MV1    0.09002

  C =
           xMPC1       xMPC2       xMPC3       xMPC4       xMPC5       xMPC6   prev.MV1
     MV1    -2.511     0.02726      -2.984        -1.2    -0.08729      -2.102   0.001735

  D =
            MO1
     MV1   0.09002
```

```
Sample time: 0.1 seconds
Discrete-time state-space model.
```

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)`.

Set `mpcobj` to use custom state estimation.

```
setEstimator(mpcobj,'custom');
```

Convert `mpcobj` to a static gain matrix for state-feedback.

```
K = ss(mpcobj)
```

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property is empty:
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

K =

  D =
          xMPC1     xMPC2     xMPC3     xMPC4     xMPC5     xMPC6  prev.MV1
    MV1   -2.609   0.02726   -2.845    -1.334   -0.1829   -2.102  0.001735

Static gain.
```

As expected this is a row vector with seven elements.

Reset `mpcobj` to use the default estimator.

```
setEstimator(mpcobj,'default');
```

You can use `'rv'` as a second argument to return a system with 3 additional inputs (reference of measured plant output, reference of unmeasured plant output and measured disturbance).

```
Kss = ss(mpcobj, 'rv')
```

```
-->Converting model to discrete time.
-->The "Model.Disturbance" property is empty:
   Assuming unmeasured input disturbance #3 is integrated white noise.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.

Kss =

  A =
                xMPC1      xMPC2      xMPC3      xMPC4      xMPC5      xMPC6    prev.MV1
    xMPC1        0.76     0.2216  -0.006469     0.3508    0.02977   -0.04783   3.019e-05
    xMPC2      -0.231      1.135    -0.4594     0.5914    0.01656    -0.1007   7.773e-08
    xMPC3    -0.03007     0.1185     0.5467     0.1386    0.01752      -0.11   0.0001122
    xMPC4     -0.1927    -0.2434    -0.1331     0.2121   -0.02496    -0.1596   0.0001298
    xMPC5     -0.1399    -0.2179    0.09667    -0.3748     0.5926     0.1385  -2.941e-05
    xMPC6    -0.09826          0     0.1393    -0.1346   -0.09578          1           0
    prev.MV1   -2.511    0.02726     -2.984       -1.2   -0.08729     -2.102    0.001735
```

```
B =
                  MO1      ref.MO1      ref.UO1          MD1
  xMPC1         0.05903    -0.003162    -0.03311      -0.06524
  xMPC2          0.1294   -8.142e-06   -8.526e-05    -0.0008242
  xMPC3         0.03619     -0.01175      -0.123       0.08255
  xMPC4         -0.0589     -0.01359      -0.1423      0.03386
  xMPC5        -0.08955     0.003081      0.03226    -0.005793
  xMPC6        -0.09022            0            0            0
  prev.MV1      0.09002      -0.1817       -1.903        1.503

C =
           xMPC1      xMPC2      xMPC3      xMPC4      xMPC5      xMPC6   prev.MV1
  MV1      -2.511    0.02726     -2.984       -1.2   -0.08729     -2.102   0.001735

D =
          MO1   ref.MO1   ref.UO1       MD1
  MV1   0.09002   -0.1817    -1.903     1.503

Sample time: 0.1 seconds
Discrete-time state-space model.
```

Use `'t'` as a second argument to return a system with 3 additional inputs (input target channels corresponding to three preview steps).

```
Kss = ss(mpcobj, 't')

Kss =

  A =
                  xMPC1       xMPC2       xMPC3       xMPC4       xMPC5       xMPC6     prev.MV1
  xMPC1            0.76      0.2216    -0.006469      0.3508     0.02977    -0.04783    3.019e-05
  xMPC2          -0.231       1.135     -0.4594       0.5914     0.01656     -0.1007    7.773e-08
  xMPC3        -0.03007      0.1185      0.5467       0.1386     0.01752       -0.11    0.0001122
  xMPC4         -0.1927     -0.2434     -0.1331       0.2121    -0.02496     -0.1596    0.0001298
  xMPC5         -0.1399     -0.2179     0.09667      -0.3748      0.5926      0.1385   -2.941e-05
  xMPC6        -0.09826            0      0.1393     -0.1346    -0.09578           1            0
  prev.MV1       -2.511     0.02726      -2.984        -1.2    -0.08729      -2.102     0.001735

  B =
                  MO1  MV1.target(0  MV1.target(1  MV1.target(2
  xMPC1         0.05903       0.01207    -0.001223     -0.009782
  xMPC2          0.1294     3.109e-05    -3.149e-06    -2.519e-05
  xMPC3         0.03619       0.04487    -0.004544      -0.03635
  xMPC4         -0.0589       0.05191    -0.005257      -0.04206
  xMPC5        -0.08955      -0.01176     0.001191       0.00953
  xMPC6        -0.09022             0            0             0
  prev.MV1      0.09002        0.6938     -0.07026       -0.5621

  C =
           xMPC1      xMPC2      xMPC3      xMPC4      xMPC5      xMPC6   prev.MV1
  MV1      -2.511    0.02726     -2.984       -1.2   -0.08729     -2.102   0.001735

  D =
                  MO1  MV1.target(0  MV1.target(1  MV1.target(2
  MV1         0.09002        0.6938     -0.07026       -0.5621

Sample time: 0.1 seconds
Discrete-time state-space model.
```

You can use the second output argument to return the vector of manipulated variables target values. This vector corresponds to `C.MV.Target - C.Model.Nominal.U(1)`.

```
[Kss, ut] = ss(mpcobj);
ut
```

```
ut = 3×1

    -0.6000
    -0.5000
    -0.4000
```

Input Arguments

`mpcobj` — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

`signals` — Signal selection
`' '` (default) | character array | string

Specify `signals` as a character vector or string with any combination that contains one or more of the following characters:

- `'r'` — Output references

- `'v'` — Measured disturbances

- `'o'` — Offset terms

- `'t'` — Input targets

For example, to obtain a controller that maps [y_m; r; v] to u, use:

```
kss = ss(mpcobj,'rv');
```

Example: `'r'`

`refPreview` — use whole reference sequence as input
`'off'` (default) | `'on'`

If this flag is `'on'`, then the input matrices of the returned controller have a larger size to multiply the whole reference sequence.

Example: `'on'`

`mdPreview` — use whole measured disturbance sequence as input
`'off'` (default) | `'on'`

If this flag is `'on'`, then the input matrices of the returned controller have a larger size to multiply the whole disturbance sequence.

Example: `'on'`

Output Arguments

kss — state space form of the unconstrained MPC controller
ss object

The discrete-time state space form of the unconstrained MPC controller has the following structure:

$x(k + 1) = Ax(k) + By_m(k)$

$u(k) = Cx(k) + Dy_m(k)$

where *A*, *B*, *C*, and *D* are the matrices forming a state space realization of the controller kss, y_m is the vector of measured outputs of the plant, and *u* is the vector of manipulated variables. The sampling time of controller kss is mpcobj.Ts.

Note Vector *x* includes the states of the observer (plant + disturbance + noise model states) and the previous manipulated variable *u*(*k*-1).

Note Only the following fields of mpcobj are used when computing the state-space model: Model, PredictionHorizon, ControlHorizon, Ts, Weights.

Note If mpcobj is set to use custom state estimation, then ss returns a static gain feedback matrix from the states of the augmented discrete-time plant, including previous values of the manipulated variables, to the manipulated variables.

kssFull — full state space form of the unconstrained MPC controller
ss object

The full discrete-time state space form of the unconstrained 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:

- *A*, *B*, *C*, and *D* are the matrices forming a state space realization of the controller from measured plant output to manipulated variables
- *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.

In the general case of nonzero offsets, y_m, *r*, *v*, and u_{target} must be interpreted as the difference between the vector and the corresponding offset. Offsets can be nonzero is mpcobj.Model.Nominal.Y or mpcobj.Model.Nominal.U are nonzero.

Vectors B_{off} and D_{off} are constant terms. They are nonzero if and only if mpcobj.Model.Nominal.DX is nonzero (continuous-time prediction models), or mpcobj.Model.Nominal.Dx-mpcobj.Model.Nominal.X is nonzero (discrete-time prediction models). In other words, when Nominal.X represents an equilibrium state, B_{off}, D_{off} are zero.

kssFullPv — full state space form of the unconstrained MPC controller
ss object

If the flag refPreview = '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 mdPreview='on', then matrices B_v and D_v multiply the whole measured disturbance sequence:

$x(k + 1) = Ax(k) + ... + B_v[v(k); v(k + 1); ...; v(k + p)] + ...$

$u(k) = Cx(k) + ... + D_v[v(k); v(k + 1); ...; v(k + p)] + ...$

ut — target values
column vector

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

Version History
Introduced before R2006a

See Also

Functions
tf | zpk | size | d2d | trim | cloffset | review | sensitivity | get | set | setEstimator | trim

Objects
mpc

Topics
"Model Predictive Control of Multi-Input Single-Output Plant"
"What is Model Predictive Control?"
"Review Model Predictive Controller for Stability and Robustness Issues"

tf

Convert unconstrained MPC controller to linear transfer function form

Syntax

```
ktf = tf(mpcobj)
```

Description

Use the Model Predictive Control Toolbox `tf` function to convert an unconstrained MPC controller with defined sample time to transfer function form (see `mpc` for background). The returned controller is equivalent to the original MPC controller `mpcobj` when no constraints are active. You can then use Control System Toolbox software for sensitivity analysis and other diagnostic calculations.

To create or convert a generic LTI dynamical system to zero/pole/gain form instead, see `tf` and "Dynamic System Models".

`ktf = tf(mpcobj)` returns the linear discrete-time dynamic controller `ktf`, in transfer function form. `ktf` is equivalent to the MPC controller `mpcobj` when no constraint is active.

Examples

Convert Unconstrained MPC Controller to Transfer Function Form

Create a plant, a corresponding MPC object, and convert it to transfer function form.

```
mpcverbosity off;              % turn off mpc messaging
plant=tf(1,[1 1],0.2);         % create plant (0.2 seconds sampling time)
mpcobj=mpc(plant,0.2);         % create mpc object (0.2 second sampling time)

ktf=tf(mpcobj)                 % convert mpc to transfer function

ktf =

  From input "MO1" to output "MV1":
    0.452 z^3 - 0.6781 z^2 - 1.506e-16 z
  ------------------------------------------
  z^3 - 1.001 z^2 + 0.0002642 z + 0.0006399

Sample time: 0.2 seconds
Discrete-time transfer function.
```

Plot Response to Unmeasured Disturbance

Plot the response to a step in the unmeasured disturbance input using both the `sim` command and the controller transfer function form.

```
% create a plant and the corresponding mpc object
mpcverbosity off;                      % turn off mpc messaging
```

```
plant=tf({1,1},{[1 1],[1 1]},0.2); % create plant (0.2 seconds sampling time)
plant=setmpcsignals(plant,'UD',2); % second input is a disturbance entering at mv
mpcobj=mpc(plant,0.2);              % create mpc object (0.2 second sampling time)

% set input and output disturbance models (remove integrators)
setindist(mpcobj,'model',tf(1))    % set input disturbance model to 1
setoutdist(mpcobj,'model',tf(1))   % set output disturbance model to 1

% closed loop output sensitivity
cloffset(mpcobj)
ans =
    1.4472

% convert the controller and calculate closed loop transfer function
ktf=tf(mpcobj);                     % convert mpc to transfer function
Muy=feedback(plant(:,1),ktf,1);    % closed loop transfer function from mv to y

% closed loop output sensitivity using transfer function
1+dcgain(Muy*ktf)
ans =
    1.4472

% plot closed loop response to a step on the measured input
step(Muy,10);                                    % simulate using step

% create option object to inject disturbance in simulation
SimOptions = mpcsimopt;                          % create object
SimOptions.UnmeasuredDisturbance = ones(50,1);   % specify unmeasured input disturbance

% simulate closed loop for 50 steps with sim and step and plot the response
[y,t,u,xp]=sim(mpcobj,50,0,[],SimOptions);       % simulate using sim

% overlap the sim results on the plot
hold on
stairs(t,y,'r')
hold off
```

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use mpc.

Output Arguments

ktf — transfer function form of the unconstrained MPC controller
tf object

Transfer function form of the MPC controller mpcobj when no constraint is active. This is also equivalent to tf(ss(mpcobj))

Note If `mpcobj` is set to use custom state estimation, then `tf` returns a static gain feedback matrix from the states of the augmented discrete-time plant, including previous values of the manipulated variables, to the manipulated variables.

Version History
Introduced before R2006a

See Also

Functions
ss | zpk | size | d2d | trim | cloffset | review | sensitivity | mpcprops | get | set | setEstimator

Objects
mpc

Topics
"Model Predictive Control of Multi-Input Single-Output Plant"

trim

Compute steady-state value of MPC controller plant model state for given inputs and outputs

Syntax

```
x = trim(mpcobj,y,u)
```

Description

Use the Model Predictive Control Toolbox `trim` function to calculate steady state values of LTI discrete-time plants controlled by an MPC controller (see `mpc` for background).

To find operating points of dynamic systems instead, see `trim` (Simulink) and "Compute Steady-State Operating Points" (Simulink Control Design).

`x = trim(mpcobj,y,u)` returns a steady-state value for the plant state or the best approximation in a least squares sense such that:

$$x - x_{off} = A(x - x_{off}) + B(u - u_{off})$$
$$y - y_{off} = C(x - x_{off}) + D(u - u_{off})$$

Here, *A*,*B*, *C*, and *D* are the state space realization matrices of the discrete-time plant model used within `mpcobj`, x_{off}, u_{off}, and y_{off} are the nominal values of the extended state *x*, input *u*, and output *y* respectively.

Examples

Calculate the steady state value of the plant model state

Create a plant, a corresponding MPC object, and calculate the steady state value of the plant model state.

```
mpcverbosity off;                              % turn off mpc messaging
plant=c2d(ss(zpk([],[-1 -10],20)),1);          % create plant (note the steady state gain)
mpcobj=mpc(plant,1);                           % create mpc object

x=trim(mpcobj,2,1)                             % caclulate trim point
MPCSTATE object with fields
         Plant: [0.4000 0.4000]
    Disturbance: 0
         Noise: [1×0 double]
       LastMove: 1
      Covariance: [3×3 double]

% check whether the calculated value is actually an equilibrium point
mpcobj.Model.Plant.A*x.Plant+mpcobj.Model.Plant.B*1-x.Plant
ans =
    1.0e-15 *
    0.1110
    0.0555
```

```
mpcobj.Model.Plant.C*x.Plant+mpcobj.Model.Plant.D*1-2
ans =
  -2.2204e-16
```

The resulting state value is an equilibrium point because for the given output and input values, the state at the next time step is equal to the current state (except some numerical errors).

Input Arguments

mpcobj — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use mpc.

y — steady state plant output
mpcobj.Model.Nominal.Y (default) | column vector | scalar

This is the plant output (including both measured and unmeasured signals) for which you want to find a stationary value of the extended plant state. If the plant has a finite steady state gain matrix G0 and y is equal to G0*u then the plant has a stationary state with output y and input u.

Example: [1 1]'

u — steady state plant input
mpcobj.Model.Nominal.U (default) | column vector | scalar

This is the plant input (including manipulated variables, measured disturbances, and unmeasured disturbances) for which you want to find a stationary value of the extended plant state. If unmeasured input disturbance variables exist, their value must be 0.

Example: [0 1]'

Output Arguments

x — steady state extended plant state
mpcstate object

This is the best approximation, in a least squares sense, of the steady-state value for the plant state corresponding to the given input and output values.

Version History
Introduced before R2006a

See Also

Functions
cloffset | sensitivity | ss | tf | zpk | size | d2d | review | set | get | sim

Objects
mpc | mpcstate | mpcsimopt

Blocks
MPC Controller

Topics
"Linearize Simulink Models"
"Linearize Simulink Models Using MPC Designer"
"What is Model Predictive Control?"
"Review Model Predictive Controller for Stability and Robustness Issues"

validateFcns

Examine prediction model and custom functions of `nlmpc` or `nlmpcMultistage` objects for potential problems

Syntax

```
validateFcns(nlmpcobj,x,mv)
validateFcns(nlmpcobj,x,mv,md)
validateFcns(nlmpcobj,x,mv,md,parameters)
validateFcns(nlmpcobj,x,mv,md,parameters,ref)
validateFcns(nlmpcobj,x,mv,md,parameters,ref,mvtarget)

validateFcns(nlmpcMSobj,x,mv)
validateFcns(nlmpcMSobj,x,mv,simdata)
```

Description

`validateFunctions` tests the prediction model, custom cost, custom constraint, and Jacobian functions of a nonlinear MPC controller for potential problems such as whether information is missing, whether input and output arguments of any user supplied functions are incompatible with object settings or whether user supplied analytical gradient/Jacobian functions are numerically accurate. When you first design your nonlinear MPC controller, or when you make significant changes to an existing controller, it is best practice to validate your controller functions.

Nonlinear MPC

`validateFcns(nlmpcobj,x,mv)` tests the functions of nonlinear MPC controller `nlmpcobj` for potential problems. The functions are tested using specified state and manipulated variable values, `x` and `mv`, respectively. These values can represent nominal conditions or an arbitrary operating point. Use this syntax if your controller has no measured disturbances and no parameters.

`validateFcns(nlmpcobj,x,mv,md)` specifies measured disturbance values. If your controller has measured disturbance channels, you must specify `md`. These values can represent nominal conditions or an arbitrary operating point.

`validateFcns(nlmpcobj,x,mv,md,parameters)` specifies parameter values. If your controller has parameters, you must specify `parameters`.

`validateFcns(nlmpcobj,x,mv,md,parameters,ref)` specifies output references at nominal conditions or for an arbitrary operating point.

`validateFcns(nlmpcobj,x,mv,md,parameters,ref,mvtarget)` specifies manipulated variable targets at nominal conditions or for an arbitrary operating point.

Multistage Nonlinear MPC

`validateFcns(nlmpcMSobj,x,mv)` tests the functions of multistage nonlinear MPC controller `nlmpcMSobj` for potential problems. The functions are tested using the specified state and manipulated variable values, `x` and `mv`, respectively. These values can represent nominal conditions or an arbitrary operating point. Use this syntax if your controller has no measured disturbances and no parameters.

validateFcns(nlmpcMSobj,x,mv,simdata) specifies the additional simdata structure. If parameters are needed for the state and stage functions, you need to provide them in simdata.

Examples

Validate Nonlinear MPC Prediction Model and Custom Functions

Create nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Validate the prediction model and custom functions at the initial states (x0) and initial inputs (u0) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Nonlinear MPC Controller with Discrete-Time Prediction Model

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nx = 4;
ny = 2;
nu = 1;
nlobj = nlmpc(nx,ny,nu);
```

```
Zero weights are applied to one or more OVs because there are fewer MVs than OVs.
```

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The discrete-time state function uses an optional parameter, the sample time `Ts`, to integrate the continuous-time model. Therefore, you must specify the number of optional parameters as 1.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function for the controller. In this case, define the first and third states as outputs. Even though this output function does not use the optional sample time parameter, you must specify the parameter as an input argument (`Ts`).

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Validate the prediction model functions for nominal states `x0` and nominal inputs `u0`. Since the prediction model uses a custom parameter, you must pass this parameter to `validateFcns`.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Nonlinear MPC Controller with Measured and Unmeasured Disturbances

Create a nonlinear MPC controller with three states, one output, and four inputs. The first two inputs are measured disturbances, the third input is the manipulated variable, and the fourth input is an unmeasured disturbance.

```
nlobj = nlmpc(3,1,'MV',3,'MD',[1 2],'UD',4);
```

To view the controller state, output, and input dimensions and indices, use the `Dimensions` property of the controller.

```
nlobj.Dimensions
```

```
ans = struct with fields:
     NumberOfStates: 3
    NumberOfOutputs: 1
     NumberOfInputs: 4
            MVIndex: 3
            MDIndex: [1 2]
            UDIndex: 4
```

Specify the controller sample time and horizons.

```
nlobj.Ts = 0.5;
nlobj.PredictionHorizon = 6;
nlobj.ControlHorizon = 3;
```

Specify the prediction model state function, which is in the file `exocstrStateFcnCT.m`.

```
nlobj.Model.StateFcn = 'exocstrStateFcnCT';
```

Specify the prediction model output function, which is in the file `exocstrOutputFcn.m`.

```
nlobj.Model.OutputFcn = 'exocstrOutputFcn';
```

Validate the prediction model functions using the initial operating point as the nominal condition for testing and setting the unmeasured disturbance state, `x0(3)`, to `0`. Since the model has measured disturbances, you must pass them to `validateFcns`.

```
x0 = [311.2639; 8.5698; 0];
u0 = [10; 298.15; 298.15];
validateFcns(nlobj,x0,u0(3),u0(1:2)');
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Input Arguments

`nlmpcobj` — Nonlinear MPC controller
`nlmpc` object

Nonlinear MPC controller, specified as an `nlmpc` object.

`nlmpcMSobj` — Nonlinear Multistage MPC controller
`nlmpcMultistage` object

Multistage nonlinear MPC controller, specified as an `nlmpcMultistage` object.

`x` — State values
vector

State values, specified as a vector of length N_x, where N_x is equal to
`nlmpcobj.Dimensions.NumberOfStates`, or `nlmpcMSobj.Dimensions.NumberOfStates`. The
state values can represent nominal conditions or an arbitrary operating point.

`mv` — Manipulated variable values
vector

Manipulated variable values, specified as a vector of length N_{mv}, where N_{mv} is equal to the length of
`nlmpcobj.Dimensions.MVIndex` or `nlmpcMSobj.Dimensions.MVIndex`. The manipulated
variable values can represent nominal conditions or an arbitrary operating point.

`md` — Measured disturbance values
`[]` (default) | vector

Measured disturbance values, specified as a vector of length N_{md}, where N_{md} is equal to the length of
`nlmpcobj.Dimensions.MDIndex`. The measured disturbance values can represent nominal
conditions or an arbitrary operating point.

If your controller has measured disturbance channels, you must specify `md`. If your controller does not
have measured disturbance channels, specify `md` as `[]`.

`parameters` — Parameter values
`[]` (default) | cell array

Parameter values used by the prediction model, custom cost function, and custom constraints,
specified as a cell array of length N_p, where N_p is equal to `nlmpcobj.Model.NumberOfParameters`
or `nlmpcMSobj.Model.NumberOfParameters`. The order of the parameters must match the order
specified in the model functions, and each parameter must be a numeric parameter with the correct
dimensions.

If your controller has parameters, you must specify `parameters`. If your controller does not have
parameters, specify `parameters` as `[]`.

`ref` — Output reference values
`[]` (default) | vector

Output reference values, specified as a vector of length N_y, where Ny is equal to
`nlmpcobj.Dimensions.NumberOfOutputs` or `nlmpcMSobj.Model.NumberOfOutputs`. `ref` is
passed to the custom cost and constraint function. The output reference values can represent
nominal conditions or an arbitrary operating point.

If you do not specify `ref`, the controller passes a vector of zeros to the custom functions.

`mvtarget` — Manipulated variable targets
`[]` (default) | vector

Manipulated variable targets, specified as a vector of length N_{mv}, where N_{mv} is equal to the length of
`nlmpcobj.Dimensions.MVIndex` or `nlmpcMSobj.Dimensions.MVIndex`. The manipulated
variable target values can represent nominal conditions or an arbitrary operating point.

`mvtarget` is passed to the custom cost and constraint function. If you do not specify `mvtarget`, the
controller passes a vector of zeros to the custom functions.

`simdata` — Run-time simulation data structure
structure

Run-time simulation data, initially created by `getSimulationData`, and specified as structure with fields described in detail in `nlmpcmove`. For `validateFcns`, only the following fields are relevant.

MeasuredDisturbance — Measured disturbance values
[ ] (default) | row vector | array

Measured disturbance values, specified as a row vector of length N_{md} or an array with N_{md} columns, where N_{md} is the number of measured disturbances. If your multistage MPC object has any measured disturbance channel defined, you must specify `MeasuredDisturbance`. If your controller has no measured disturbances, this field does not exist in the structure returned by `getSimulationData`.

StateFcnParameters — State function parameter values
[ ] (default) | vector

State function parameter values, specified as a vector with length equal to the value of the `Model.ParameterLength` property of the multistage controller object. If `Model.StateFcn` needs a parameter vector, you must provide its value at runtime using this field. If your state function has no parameter, this field does not exist in the structure returned by `getSimulationData`.

StageFcnParameters — Stage functions parameter values
[ ] (default) | vector

Stage functions parameter values, specified as a vector with length equal to the sum of all the values in the `Stages(i).ParameterLength` properties of the multistage controller object. If any cost or constraint function defined in the `Stages` property needs a parameter vector, you must provide all the parameter vectors at runtime (stacked in a single column) using this field. If none of your stage functions needs any parameter, this field does not exist in the structure returned by `getSimulationData`.

Tips

*   When you provide your own analytical Jacobian functions, it is especially important that these functions return valid Jacobian values. If `validateFunctions` detects large differences between the values returned by your user-defined Jacobian functions and the finite-difference approximation, verify the code in your Jacobian implementations.

Algorithms

For each controller function, `validateFunctions` checks whether the function:

*   Exists on the MATLAB path
*   Has the required number of input arguments
*   Can be executed successfully without errors
*   Returns the output arguments with the correct size and dimensions
*   Returns valid numerical data; that is, it does not return `Inf` or `NaN` values

For Jacobian functions, `validateFunctions` checks whether the returned values are comparable to a finite-difference approximation of the Jacobian values. These finite-difference values are computed using numerical perturbation.

Version History
Introduced in R2018b

See Also

Functions
generateJacobianFunction | nlmpcmove | nlmpcmoveCodeGeneration | getSimulationData | getCodeGenerationData

Objects
nlmpc | nlmpcMultistage

Blocks
Multistage Nonlinear MPC Controller

Topics
"Landing a Vehicle Using Multistage Nonlinear MPC"
"Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC"
"Nonlinear MPC"

zpk

Convert unconstrained MPC controller to zero/pole/gain form

Syntax

```
kzpk = zpk(mpcobj)
```

Description

Use the Model Predictive Control Toolbox `zpk` function to convert an unconstrained MPC controller with defined sample time to zero/pole/gain form (see `mpc` for background). The returned controller is equivalent to the original MPC controller `mpcobj` when no constraints are active. You can then use Control System Toolbox software for sensitivity analysis and other diagnostic calculations.

To create or convert a generic LTI dynamical system to zero/pole/gain form instead, see `zpk` and "Dynamic System Models".

`kzpk = zpk(mpcobj)` returns the linear discrete-time dynamic controller `kzpk`, in zero/pole/gain form. `kzpk` is equivalent to the MPC controller `mpcobj` when no constraint is active.

Examples

Convert Unconstrained MPC Controller to Zero/Pole/Gain Form

Create a plant, a corresponding MPC object, and convert it to zero/pole/gain form.

```
mpcverbosity off;            % turn off mpc messaging
plant=tf(1,[1 1],0.2);       % create plant (0.2 seconds sampling time)
mpcobj=mpc(plant,0.2);       % create mpc object (0.2 second sampling time)

kzpk=zpk(mpcobj)             % convert to zpk form show the controller's poles and zeroes

kzpk =

  From input "MO1" to output "MV1":
      0.45205 z^2 (z-1.5)
  -----------------------------
  (z-1) (z-0.02575) (z+0.02485)

Sample time: 0.2 seconds
Discrete-time zero/pole/gain model.
```

The poles are all inside the unit circle, except the one in `z=1`. The position of this pole , which is due to the fact that the default noise model is an integrator, causes the controller static gain to approach infinity, in turn allowing near perfect tracking of the output reference signal.

Input Arguments

`mpcobj` — Model predictive controller
mpc object

Model predictive controller, specified as an MPC controller object. To create an MPC controller, use `mpc`.

Output Arguments

kzpk — zero/pole/gain form of the unconstrained MPC controller
zpk object

Zero/pole/gain form of the MPC controller `mpcobj` when no constraint is active. This is also equivalent to `zpk(ss(mpcobj))`

Note If `mpcobj` is set to use custom state estimation, then `tf` returns a static gain feedback matrix from the states of the augmented discrete-time plant, including previous values of the manipulated variables, to the manipulated variables.

Version History

Introduced before R2006a

See Also

Functions
ss | tf | size | d2d | trim | cloffset | review | sensitivity | mpcprops | get | set | trim | setEstimator

Objects
mpc

Topics
"Model Predictive Control of Multi-Input Single-Output Plant"

Objects

explicitMPC

Explicit model predictive controller

Description

Explicit model predictive control uses offline computations to determine all operating regions in which the optimal control moves are determined by evaluating a linear function. Explicit MPC controllers require fewer run-time computations than traditional (implicit) model predictive controllers and are therefore useful for applications that require small sample times.

To implement explicit MPC, first design a traditional (implicit) model predictive controller for your application, and then use this controller to generate an explicit MPC controller for use in real-time control. For more information, see "Design Workflow for Explicit MPC".

Creation

To create an `explicitMPC` object:

1 Create an implicit MPC controller using an `mpc` object.

2 Define the operating range for the explicit MPC controller by creating a range structure using the `generateExplicitRange` function and specifying the bounds using dot notation.

3 Define the optimization options for converting the implicit controller into an explicit controller using the `generateExplicitOptions` function.

4 Create the explicit MPC controller based on the implicit controller, operating range, and optimization options using the `generateExplicitMPC` function.

Properties

MPC — Implicit MPC controller
`mpc` object

Implicit MPC controller, specified as an `mpc` object.

range — Parameter bounds
structure

Parameter bounds that define the controller operating range, specified as a structure with the following fields.

Field	Description
`State`	Bounds on controller state values
`Reference`	Bounds on controller reference signal values
`MeasuredDisturbance`	Bounds on measured disturbance values
`ManipulatedVariable`	Bounds on manipulated variable values

Define this property using the `range` input argument to the `generateExplicitMPC` function, which you create using the `generateExplicitRange` function and modify using dot notation. For detailed descriptions of the range parameters, see `generateExplicitRange`.

`OptimizationOptions` — Optimization options
structure

Optimization options for the conversion computation, specified as a structure with the following fields.

Field	Description
`zerotol`	Zero-detection tolerance
`removetol`	Redundant-inequality-constraint detection tolerance
`flattol`	Flat region detection tolerance
`normalizetol`	Constraint normalization tolerance
`maxiterNNLS`	Maximum number of NNLS solver iterations
`maxiterQP`	Maximum number of QP solver iterations
`maxiterBS`	Maximum number of bisection method iterations
`polyreduction`	Method for removing redundant inequalities

Define this property using the `opt` input argument to the `generateExplicitMPC` function, which you create using the `generateExplicitOptions` function. For detailed descriptions of these options, see `generateExplicitOptions`.

`PiecewiseAffineSolution` — Piecewise affine solution
structure array

Piecewise affine solution for the different operating regions, specified as a structure array with N_r elements, where N_r is the number of operating regions.

Each structure element contains fields defining the inequality constraints and control law for each region. For more information on the control law and constraints, see "Design Workflow for Explicit MPC".

Field	Dimensions
F	Row vector of length N_x-by-N_{mv}.
G	Column vector of length N_{mv}
H	N_c-by-N_x array
K	Column vector of length N_c

Here:

- N_x is the number of independent variables.
- N_{mv} is the number of manipulated variables.
- N_c is the number of inequality constraints for the region.

`IsSimplified` — Flag indicating whether the explicit control law has been simplified
false (default) | true

Flag indicating whether the explicit control law has been simplified using the `simplify` command. If the control law is simplified, it approximates the implicit MPC controller behavior. If the control law is not simplified, it should reproduce the implicit controller behavior exactly, provided both operate within the bounds described by the `range` property.

Object Functions

generatePlotParameters	Parameters for plotSection
getCodeGenerationData	Create data structures for mpcmoveCodeGeneration
mpcmoveExplicit	Compute optimal control using explicit MPC
plotSection	Visualize explicit MPC control law as 2-D sectional plot
sim	Simulate an MPC controller in closed loop with a linear plant
simplify	Reduce explicit MPC controller complexity and memory requirements
size	Size and order of MPC Controller

Examples

Generate Explicit MPC Controller

Generate an explicit MPC controller based upon a traditional MPC controller for a double-integrator plant.

Define the double-integrator plant.

```
plant = tf(1,[1 0 0]);
```

Create a traditional (implicit) MPC controller for this plant, with sample time 0.1, a prediction horizon of 10, and a control horizon of 3.

```
Ts = 0.1;
p = 10;
m = 3;
mpcobj = mpc(plant,Ts,p,m);
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

To generate an explicit MPC controller, you must specify the ranges of parameters such as state values and manipulated variables. To do so, generate a range structure. Then, modify values within the structure to the desired parameter ranges.

```
range = generateExplicitRange(mpcobj);
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
range.State.Min(:) = [-10;-10];
range.State.Max(:) = [10;10];
range.Reference.Min = -2;
range.Reference.Max = 2;
range.ManipulatedVariable.Min = -1.1;
range.ManipulatedVariable.Max = 1.1;
```

Use the more robust reduction method for the computation. Use `generateExplicitOptions` to create a default options set, and then modify the `polyreduction` option.

```
opt = generateExplicitOptions(mpcobj);
opt.polyreduction = 1;
```

Generate the explicit MPC controller.

```
empcobj = generateExplicitMPC(mpcobj,range,opt)


Explicit MPC Controller
---------------------------------------------
Controller sample time:   0.1 (seconds)
Polyhedral regions:       1
Number of parameters:     4
Is solution simplified:   No
State Estimation:         Default Kalman gain
---------------------------------------------
Type 'empcobj.MPC' for the original implicit MPC design.
Type 'empcobj.Range' for the valid range of parameters.
Type 'empcobj.OptimizationOptions' for the options used in multi-parametric QP computation.
Type 'empcobj.PiecewiseAffineSolution' for regions and gain in each solution.
```

Version History
Introduced in R2014b

See Also

Functions
generateExplicitMPC | generateExplicitRange | generateExplicitOptions | simplify | generatePlotParameters | plotSection | mpcmoveExplicit | sim

Objects
mpc | mpcstate

Blocks
Explicit MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC Control of an Aircraft with Unstable Poles"
"Explicit MPC Control of DC Servomotor with Constraint on Unmeasured Output"
"Explicit MPC"
"Design Workflow for Explicit MPC"
"What is Model Predictive Control?"

mpc

Model predictive controller

Description

A model predictive controller uses linear plant, disturbance, and noise models to estimate the controller state and predict future plant outputs. Using the predicted plant outputs, the controller solves a quadratic programming optimization problem to determine control moves.

For more information on the structure of model predictive controllers, see "MPC Prediction Models".

Creation

Syntax

```
mpcobj = mpc(plant)
mpcobj = mpc(plant,ts)
mpcobj = mpc(plant,ts,P,M,W,MV,OV,DV)

mpcobj = mpc(model)
mpcobj = mpc(model,ts)
mpcobj = mpc(model,ts,P,M,W,MV,OV,DV)
```

Description

`mpcobj = mpc(plant)` creates a model predictive controller object based on the discrete-time prediction model `plant`. The controller, `mpcobj`, inherits its control interval from `plant.Ts`, and its time unit from `plant.TimeUnit`. All other controller properties are default values. After you create the MPC controller, you can set its properties using dot notation.

If `plant.Ts = -1`, you must set the `Ts` property of the controller to a positive value before designing and simulating your controller.

`mpcobj = mpc(plant,ts)` creates a model predictive controller based on the specified plant model and sets the `Ts` property of the controller. If `plant` is:

- A continuous-time model, then the controller discretizes the model for prediction using sample time `ts`
- A discrete-time model with a specified sample time, the controller resamples the plant for prediction using sample time `ts`
- A discrete-time model with an unspecified sample time (`plant.Ts = −1`), it inherits the sample time `ts` when used for predictions.

`mpcobj = mpc(plant,ts,P,M,W,MV,OV,DV)` specifies the following controller properties. If any of these values are omitted or empty, the default values apply.

- `P` sets the `PredictionHorizon` property.

- M sets the `ControlHorizon` property.
- W sets the `Weights` property.
- MV sets the `ManipulatedVariables` property.
- OV sets the `OutputVariables` property.
- DV sets the `DisturbanceVariables` property.

`mpcobj = mpc(model)` creates a model predictive controller object based on the specified prediction model set, which includes the plant, input disturbance, and measurement noise models along with the nominal conditions at which the models were obtained. When you do not specify a sample time, the plant model, `model.Plant`, must be a discrete-time model. This syntax sets the `Model` property of the controller.

`mpcobj = mpc(model,ts)` creates a model predictive controller based on the specified plant model and sets the `Ts` property of the controller to `ts`. If `model.Plant` is a discrete-time LTI model with an unspecified sample time (`model.Plant.Ts = −1`), it inherits the sample time `ts` when used for predictions.

`mpcobj = mpc(model,ts,P,M,W,MV,OV,DV)` specifies additional controller properties. If any of these values are omitted or empty, the default values apply.

Input Arguments

`plant` — Plant prediction model
LTI model | identified linear model

Plant prediction model, specified as either an LTI model or a linear System Identification Toolbox model. The specified plant corresponds to the `Model.Plant` property of the controller.

If you do not specify a sample time when creating your controller, `plant` must be a discrete-time model. For prediction, `plant` is discretized or resampled if needed using `mpcobj.Ts` as sample time. Delays, if present, are incorporated in the resulting discrete time model.

For more information on MPC prediction models, see "MPC Prediction Models".

Note Direct feedthrough from manipulated variables to any output in `plant` is not supported.

Example: `tf(10,[1 10])`

`model` — Prediction model
structure

Prediction model, specified as a structure with the same format as the `Model` property of the controller. If you do not specify a sample time when creating your controller, `model.Plant` must be a discrete-time model.

For more information on MPC prediction models, see "MPC Prediction Models".

Properties

`Ts` — Controller sample time
positive scalar

Controller sample time, specified as a positive finite scalar. The controller uses a discrete-time model with sample time `Ts` for prediction. The time unit is inherited from `plant.TimeUnit`.

Example: `mpcobj.Ts = 0.1`

PredictionHorizon — Prediction horizon
10 + delay-dependent steps (default) | positive integer

Prediction horizon steps, specified as a positive integer. The product of `PredictionHorizon` and `Ts` is the prediction time; that is, how far the controller looks into the future.

If you specify a prediction horizon that is not larger than the number of steps necessary to cover the longest delay among all the input-output channels, `Nd = max(mpcobj.Model.Plant.IODelay(:))/mpcobj.Ts`, then the software automatically increases the prediction horizon by `floor(Nd)`.

Example: `mpcobj.PredictionHorizon = 15`

ControlHorizon — Control horizon
2 (default) | positive integer | vector of positive integers

Control horizon, specified as one of the following:

- Positive integer, m, between 1 and p, inclusive, where p is equal to `PredictionHorizon`. In this case, the controller computes m free control moves occurring at times k through $k+m$-1, and holds the controller output constant for the remaining prediction horizon steps from $k+m$ through $k+p$-1. Here, k is the current control interval.

- Vector of positive integers $[m_1, m_2, ...]$, specifying the lengths of blocking intervals. By default the controller computes M blocks of free moves, where M is the number of blocking intervals. The first free move applies to times k through $k+m_1$-1, the second free move applies from time $k+m_1$ through $k+m_1+m_2$-1, and so on. Using block moves can improve the robustness of your controller. The sum of the values in `ControlHorizon` must match the prediction horizon p. If you specify a vector whose sum is:

  - Less than the prediction horizon, then the controller adds a blocking interval. The length of this interval is such that the sum of the interval lengths is p. For example, if $p=10$ and you specify a control horizon of `ControlHorizon=[1 2 3]`, then the controller uses four intervals with lengths `[1 2 3 4]`.

  - Greater than the prediction horizon, then the intervals are truncated until the sum of the interval lengths is equal to p. For example, if $p=10$ and you specify a control horizon of `ControlHorizon= [1 2 3 6 7]`, then the controller uses four intervals with lengths `[1 2 3 4]`.

  For more information on manipulated variable blocking, see "Manipulated Variable Blocking".

Example: `mpcobj.ControlHorizon = 3`

Model — Prediction model and nominal conditions
structure

Prediction model and nominal conditions, specified as a structure with the following fields. For more information on the MPC prediction model, see "MPC Prediction Models" and "Controller State Estimation".

Plant — Plant prediction model
LTI model | identified linear model

Plant prediction model, specified as either an LTI model or a linear System Identification Toolbox model.

Note Direct feedthrough from manipulated variables to any output in `plant` is not supported.

Example: `mpcobj.Model.Plant = ss(-1,1,1,0)`

Disturbance — Model describing expected unmeasured disturbances
LTI model

Model describing expected unmeasured disturbances, specified as an LTI model. This model is required only when the plant has unmeasured disturbances. You can set this disturbance model directly using dot notation or using the `setindist` function.

By default, input disturbances are expected to be integrated white noise. To model the signal, an integrator with dimensionless unity gain is added for each unmeasured input disturbance, unless the addition causes the controller to lose state observability. In that case, the disturbance is expected to be white noise, and so, a dimensionless unity gain is added to that channel instead.

Example: `mpcobj.Model.Disturbance = tf(5,[1 5])`

Noise — Model describing expected output measurement noise
LTI model

Model describing expected output measurement noise, specified as an LTI model.

By default, measurement noise is expected to be white noise with unit variance. To model the signal, a dimensionless unity gain is added for each measured channel.

Example: `mpcobj.Model.Noise = zpk(0,-1,1)`

Nominal — Nominal operating point at which plant model is linearized
structure

Nominal operating point at which plant model is linearized, specified as a structure with the following fields.

Field	Description	Default
X	Plant state at operating point, specified as a column vector with length equal to the number of states in `Model.Plant`.	zero vector
U	Plant input at operating point, including manipulated variables and measured and unmeasured disturbances, specified as a column vector with length equal to the number of inputs in `Model.Plant`.	zero vector
Y	Plant output at operating point, including measured and unmeasured outputs, specified as a column vector with length equal to the number of outputs in `Model.Plant`.	zero vector

Field	Description	Default
DX	For continuous-time models, DX is the state derivative at operating point: DX=f(X,U). For discrete-time models, DX=x(k+1)-x(k)=f(X,U)-X. Specify DX as a column vector with length equal to the number of states in `Model.Plant`.	zero vector

`ManipulatedVariables` — Manipulated variable information, bounds, and scale factors
structure array

Manipulated Variable (MV) information, bounds, and scale factors, specified as a structure array with N_{mv} elements, where N_{mv} is the number of manipulated variables. To access this property, you can use the alias MV instead of `ManipulatedVariables`.

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 $du/dt \cong \Delta u(k)/T_s$.

Each structure element has the following fields.

`Min` — MV lower bound
`-Inf` (default) | scalar | vector

Lower bound for a given manipulated variable, specified as a scalar or vector. By default, this lower bound is `-Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(1).Min = -5`

`Max` — MV upper bound
`Inf` (default) | scalar | vector

Upper bound for a given manipulated variable, specified as a scalar or vector. By default, this upper bound is `Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(1).Max = 5`

`MinECR` — MV lower bound softness
`0` (default) | nonnegative scalar | vector

Softness of the lower bound for a given manipulated variable. A larger equal concern for relaxation (ECR) value indicates a softer constraint, specified as a nonnegative scalar or vector. By default, MV lower bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(1).MinECR = 0.01`

MaxECR — MV upper bound
0 (default) | nonnegative scalar | vector

Softness of the upper bound for a given manipulated variable. A larger equal concern for relaxation (ECR) value indicates a softer constraint, specified as a nonnegative scalar or vector. By default, MV upper bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(1).MaxECR = 0.01`

RateMin — MV rate of change lower bound
-Inf (default) | nonpositive scalar | vector

Lower bound for the rate of change of a given manipulated variable, specified as a nonpositive scalar or vector. The MV rate of change is defined as $MV(k)$ - $MV(k$-1), where k is the current time. By default, this lower bound is -Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(1).RateMin = -2`

RateMax — MV rate of change upper bound
Inf (default) | nonnegative scalar | vector

Upper bound for the rate of change of a given manipulated variable, specified as a nonnegative scalar or vector. The MV rate of change is defined as $MV(k)$ - $MV(k$-1), where k is the current time. By default, this lower bound is Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(1).RateMax = 2`

RateMinECR — MV rate of change lower bound softness
0 (default) | nonnegative finite scalar | vector

Softness of the lower bound for the rate of change of a given manipulated variable. A larger equal concern for relaxation (ECR) value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, MV rate of change lower bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR values over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR values are used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(2).RateMinECR = 0.01`

RateMaxECR — MV rate of change upper bound softness
0 (default) | nonnegative finite scalar | vector

Softness of the upper bound for the rate of change of a given manipulated variable. A larger equal concern for relaxation (ECR) value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, MV rate of change upper bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR values over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR values are used for the remaining steps of the prediction horizon.

Example: `mpcobj.ManipulatedVariables(2).RateMaxECR = 0.01`

Target — MV targets
`'nominal'` (default) | scalar | vector

Targets for a given manipulated variable, specified as a scalar, vector, or as `'nominal'` (default). When `Target` is `'nominal'`, then the manipulated variable targets correspond to `mpcobj.Model.Nominal.U`.

To use the same target across the prediction horizon, specify a scalar value.

To vary the target over the prediction horizon from time k to time $k+p$-1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final value is used for the remaining steps of the prediction horizon.

You might want to set target values for some manipulated variables, along with corresponding nonzero cost function weights, for economic or operational reasons, whenever you have more manipulated variables than plant outputs. For more information, see "Design MPC Controller for Nonsquare Plants" and "Setting Targets for Manipulated Variables".

Example: `mpcobj.ManipulatedVariables(2).Target = [0.3 0.2]`

Name — MV name
string | character vector

Name of a given manipulated variable, specified as a string or character vector. This is a read-only property. To modify the names of manipulated variables, use `mpcobj.Model.Plant.InputName`.

Example: `mpcobj.ManipulatedVariables(2).Name`

Units — MV units
`""` (default) | string | character vector

Units of a given manipulated variable, specified as a string or character vector. This is a read-only property. To modify the units of manipulated variables, use `mpcobj.Model.Plant.InputUnit`.

Example: `mpcobj.ManipulatedVariables(2).Units`

ScaleFactor — MV scale factor
1 (default) | positive finite scalar

Scale factor of a given manipulated variable, specified as a positive finite scalar. Specifying the proper scale factor can improve numerical conditioning for optimization. In general, use the amplitude of the operating range of the manipulated variable. For more information, see "Specify Scale Factors".

Example: `mpcobj.ManipulatedVariables(1).ScaleFactor = 10`

Type — MV type
`'continuous'` (default) | `'integer'` | `'binary'` | vector

Type of a given manipulated variable, specified as:

- `'continuous'` — This indicates that the manipulated variable is continuous.
- `'binary'` — This restricts the manipulated variable to be either 0 or 1.
- `'integer'` — This restricts the manipulated variable to be an integer.
- A vector containing all the possible values — This restricts the manipulated variable to the specified values, for example `mpcobj.MV(1).Type = [-1,0,0.5,1,2];`.

By default, the type is set to `'continuous'`.

For more information, see "Finite Control Set MPC".

Example: `mpcobj.ManipulatedVariables(1).Type = 'binary'`

OutputVariables — Output variable information, bounds, and scale factors
structure array

Output variable (OV) information, bounds, and scale factors, specified as a structure array with N_y elements, where N_y is the number of output variables. To access this property, you can use the alias `OV` instead of `OutputVariables`.

Each structure element has the following fields.

Min — OV lower bound
`-Inf` (default) | scalar | vector

Lower bound for a given output variable, specified as a scalar or vector. By default, this lower bound is `-Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `mpcobj.OutputVariables(1).Min = -10`

Max — OV upper bound
Inf (default) | scalar | vector

Upper bound for a given output variable, specified as a scalar or vector. By default, this upper bound is Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: mpcobj.OutputVariables(1).Max = 10

MinECR — OV lower bound softness
1 (default) | nonnegative finite scalar | vector

Softness of the lower bound for a given output variable. A larger equal concern for relaxation (ECR) value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, OV upper bounds are soft constraints.

To avoid creating an infeasible optimization problem at run time, it is best practice to use soft OV bounds.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: mpcobj.OutputVariables(1).MinECR = 5

MaxECR — OV upper bound softness
1 (default) | nonnegative finite scalar | vector

Softness of the upper bound for a given output variable. A larger equal concern for relaxation (ECR) value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, OV lower bounds are soft constraints.

To avoid creating an infeasible optimization problem at run time, it is best practice to use soft OV bounds.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: mpcobj.OutputVariables(1).MaxECR = 10

Name — OV name
string | character vector

Name of a given output variable, specified as a string or character vector. This is a read-only property. To modify the names of output variables, use mpcobj.Model.Plant.OutputName

Example: mpcobj.OutputVariables(2).Name

Units — OV units
"" (default) | string | character vector

Units of a given output variable, specified as a string or character vector. This is a read-only property. To modify the names of output variables, use `mpcobj.Model.Plant.OutputUnit`.

Example: `mpcobj.OutputVariables(2).Units`

ScaleFactor — OV scale factor
1 (default) | positive finite scalar

Scale factor of a given output variable, specified as a positive finite scalar. Specifying the proper scale factor can improve numerical conditioning for optimization. In general, use the operating range of the output variable. For more information, see "Specify Scale Factors".

Example: `mpcobj.OutputVariables(1).ScaleFactor = 20`

DisturbanceVariables — Input disturbance variable information and scale factors
structure array

Disturbance variable (DV) information and scale factors, specified as a structure array with N_d elements, where N_d is the total number of measured and unmeasured disturbance inputs. The order of the disturbance signals within `DisturbanceVariables` is the following: the first N_{md} entries relate to measured input disturbances, the last N_{ud} entries relate to unmeasured input disturbances.

To access this property, you can use the alias `DV` instead of `DisturbanceVariables`.

Each structure element has the following fields.

Name — DV name
string | character vector

DV name, specified as a string or character vector. This is a read-only property. To modify the names of disturbance variables, use `mpcobj.Plant.Inputname`.

Example: `mpcobj.DisturbanceVariables(1).Name`

Units — OV units
"" (default) | string | character vector

OV units, specified as a string or character vector. This is a read-only property. To modify the units of disturbance variables, use `mpcobj.Model.Plant.InputUnit`.

Example: `mpcobj.DisturbanceVariables(1).Units`

ScaleFactor — DV scale factor
1 (default) | positive finite scalar

DV scale factor, specified as a positive finite scalar. Specifying the proper scale factor can improve numerical conditioning for optimization. For more information, see "Specify Scale Factors".

Example: `mpcobj.DisturbanceVariables(1).ScaleFactor = 15`

Weights — Standard cost function tuning weights
structure

Standard cost function tuning weights, specified as a structure. The controller applies these weights to the scaled variables. Therefore, the tuning weights are dimensionless values.

The format of `OutputWeights` must match the format of the `Weights.OutputVariables` property of the controller object. For example, you cannot specify constant weights across the prediction horizon in the controller object, and then specify time-varying weights using `mpcmoveopt`.

`Weights` has the following fields. The values of these fields depend on whether you use the standard or alternative cost function. For more information on these cost functions, see "Optimization Problem".

ManipulatedVariables — Manipulated variable tuning weights
row vector | array

Manipulated variable tuning weights, which penalize deviations from MV targets, specified as a row vector or array of nonnegative values. The default weight for all manipulated variables is `0`.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

If you use the alternative cost function, specify `Weights.ManipulatedVariables` as a cell array that contains the N_{mv}-by-N_{mv} R_u matrix. For example, `mpcobj.Weights.ManipulatedVariables = {Ru}`. R_u must be a positive semidefinite matrix. Varying the R_u matrix across the prediction horizon Is not supported. For more information, see "Alternative Cost Function".

Example: `mpcobj.Weights.ManipulatedVariables = [0.1 0.2]`

ManipulatedVariablesRate — Manipulated variable rate tuning weights
row vector | array | cell array

Manipulated variable rate tuning weights, which penalize large changes in control moves, specified as a row vector or array of nonnegative values. The default weight for all manipulated variable rates is `0.1`.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable rate tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

Note It is best practice to use nonzero manipulated variable rate weights. If all manipulated variable rate weights are strictly positive, the resulting QP problem is strictly convex. If some weights are zero, the QP Hessian could be positive semidefinite. To keep the QP problem strictly convex, when the condition number of the Hessian matrix $K_{\Delta U}$ is larger than 10^{12}, the quantity `10*sqrt(eps)` is added to each diagonal term. See "Cost Function".

If you use the alternative cost function, specify `Weights.ManipulatedVariablesRate` as a cell array that contains the N_{mv}-by-N_{mv} $R_{\Delta u}$ matrix. For example,

mpcobj.Weights.ManipulatedVariablesRate = {Rdu}. $R_{\Delta u}$ must be a positive semidefinite matrix. Varying the $R_{\Delta u}$ matrix across the prediction horizon Is not supported. For more information, see "Alternative Cost Function".

Example: `mpcobj.Weights.ManipulatedVariablesRate = [0.1 0.1]`

OutputVariables — Output variable tuning weights
vector | array

Output variable tuning weights, which penalize deviation from output references, specified as a row vector or array of nonnegative values. The default weight for all output variables is 1.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

If you use the alternative cost function, specify `Weights.OutputVariables` as a cell array that contains the N_y-by-N_y Q matrix. For example, `mpcobj.Weights.OutputVariables = {Q}`. Q must be a positive semidefinite matrix. Varying the Q matrix across the prediction horizon Is not supported. For more information, see "Alternative Cost Function".

Example: `mpcobj.Weights.OutputVariables = [1 1]`

ECR — Slack variable tuning weight
1e5 (default) | positive scalar

Slack variable tuning weight, specified as a positive scalar. Increase or decrease the equal concern for relaxation (ECR) weight to make all soft constraints harder or softer, respectively.

Example: `mpcobj.Weights.ECR = 1e4`

Optimizer — QP optimization parameters
structure

QP optimization parameters, specified as a structure with the following fields. The first four fields, `Algorithm`, `ActiveSetOptions`, `InteriorPointOptions` and `MixedIntegerOptions`, are related to the built in solvers. If you chose to use a custom solver for simulation (by setting `CustomSolver` to `true`) these four fields are ignored for simulation. Likewise, if you chose to use a custom solver for code generation (by setting `CustomSolverCodeGen` to `true`) these four fields are ignored for code generation.

For more information on the supported QP solvers, see "QP Solvers".

Algorithm — QP solver algorithm
`'active-set'` (default) | `'interior-point'`

QP solver algorithm, specified as one of the following:

- `'active-set'` — Solve the QP problem using the KWIK active-set algorithm.
- `'interior-point'` — Solve the QP problem using a primal-dual interior-point algorithm with Mehrotra predictor-corrector.

For applications that require solving QP problems, you can also access the active-set and interior-point algorithms using the `mpcActiveSetSolver` and `mpcInteriorPointSolver` functions, respectively.

Example: `mpcobj.Optimizer.Algorithm = 'interior-point'`

ActiveSetOptions — Active-set QP solver settings
structure

Active-set QP solver settings, specified as a structure. These settings apply only when `Algorithm` is `'active-set'`, and the `type` property of all manipulated variables is `'continuous'`.

You can specify the following active-set optimizer settings.

MaxIterations — Maximum number of iterations
`'default'` (default) | positive integer

Maximum number of iterations allowed when computing the QP solution, specified as one of the following:

- `'default'` — The MPC controller automatically computes the maximum number of QP solver iterations as $4(n_c + n_v)$, where:

  - n_c is the total number of constraints across the prediction horizon.
  - n_v is the total number of optimization variables across the control horizon.

  The default `MaxIterations` value has a lower bound of `120`.

- Positive integer — The QP solver stops after the specified number of iterations. If the solver fails to converge in the final iteration, the controller:

  - Freezes the controller movement if `UseSuboptimalSolution` is `false`.
  - Applies the suboptimal solution reached after the final iteration if `UseSuboptimalSolution` is `true`.

Note The default `MaxIterations` value can be very large for some controller configurations, such as those with large prediction and control horizons. When simulating such controllers, if the QP solver cannot find a feasible solution, the simulation can appear to stop responding, since the solver continues searching for a solution until the number of iterations reach `MaxIterations`.

Example: `mpcobj.Optimizer.ActiveSetOptions.MaxIterations = 2000`

ConstraintTolerance — Tolerance used to verify that inequality constraints are satisfied
`1e-6` (default) | positive scalar

Tolerance used to verify that inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger `ConstraintTolerance` value allows for larger constraint violations.

Example: `mpcobj.Optimizer.ActiveSetOptions.ConstraintTolerance = 1e-5`

UseWarmStart — Option indicating whether to warm start each QP solver iteration
`true` (default) | `false`

Option indicating whether to *warm start* each QP solver iteration by passing in a list of active inequalities from the previous iteration, specified as a logical value. Inequalities are active when their equal portion is true.

Example: `mpcobj.Optimizer.ActiveSetOptions.UseWarmStart = true`

InteriorPointOptions — Interior-point QP solver settings
structure

Interior-point QP solver settings, specified as a structure. These settings apply only when `Algorithm` is `'interior-point'`, and the `type` property of all manipulated variables is `'continuous'`.

You can specify the following interior-point optimizer settings.

MaxIterations — Maximum number of iterations
50 (default) | positive integer

Maximum number of iterations allowed when computing the QP solution, specified as a positive integer. The QP solver stops after the specified number of iterations. If the solver fails to converge in the final iteration, the controller:

- Freezes the controller movement if `UseSuboptimalSolution` is `false`.
- Applies the suboptimal solution reached after the final iteration if `UseSuboptimalSolution` is `true`.

Example: `mpcobj.Optimizer.InteriorPointOptions.MaxIterations = 30`

ConstraintTolerance — Tolerance used to verify that equality and inequality constraints are satisfied
1e-6 (default) | positive scalar

Tolerance used to verify that equality and inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger `ConstraintTolerance` value allows for larger constraint violations.

Example: `mpcobj.Optimizer.InteriorPointOptions.ConstraintTolerance = 1e-5`

OptimalityTolerance — Termination tolerance for first-order optimality (KKT dual residual)
1e-6 (default) | positive scalar

Termination tolerance for first-order optimality (KKT dual residual), specified as a positive scalar.

Example: `mpcobj.Optimizer.InteriorPointOptions.OptimalityTolerance = 1e-5`

ComplementarityTolerance — Termination tolerance for first-order optimality (KKT average complementarity residual)
1e-8 (default) | positive scalar

Termination tolerance for first-order optimality (KKT average complementarity residual), specified as a positive scalar. Increasing this value improves robustness, while decreasing this value increases accuracy.

Example: `mpcobj.Optimizer.InteriorPointOptions.ComplementarityTolerance = 1e-6`

StepTolerance — Termination tolerance for decision variables
1e-8 (default) | positive scalar

Termination tolerance for decision variables, specified as a positive scalar.

Example: `mpcobj.Optimizer.InteriorPointOptions.StepTolerance = 1e-7`

MixedIntegerOptions — Mixed-integer QP solver settings
structure

Mixed-integer QP solver settings, specified as a structure. This setting apply when any manipulated variable has a `type` property which is not `'continuous'`. In this case, a built it mixed-integer KWIK algorithm that implements a branch and bound method is used.

You can specify the following mixed-integer QP optimizer settings.

MaxIterations — Maximum number of iterations
1000 (default) | positive integer

Maximum number of iterations allowed when computing the mixed-integer QP solution, specified as a positive integer. The mixed-integer QP solver stops after the specified number of iterations. If the solver fails to converge in the final iteration, the controller:

- Freezes the controller movement if `UseSuboptimalSolution` is `false`.
- Applies the suboptimal solution reached after the final iteration if `UseSuboptimalSolution` is `true`.

Example: `mpcobj.Optimizer.MixedIntegerOptions.MaxIterations = 500`

ConstraintTolerance — Tolerance used to verify that equality and inequality constraints are satisfied
1e-6 (default) | positive scalar

Tolerance used to verify that equality and inequality constraints are satisfied by the optimal solution, specified as a positive scalar. A larger `ConstraintTolerance` value allows for larger constraint violations.

Example: `mpcobj.Optimizer.MixedIntegerOptions.ConstraintTolerance = 1e-5`

DiscreteConstraintTolerance — Tolerance used to verify that constraints on the discrete manipulated variables are satisfied
1e-6 (default) | positive scalar

Tolerance used to verify that constraints in the discrete manipulated variables are satisfied by the optimal solution, specified as a positive scalar. A larger `DiscreteConstraintTolerance` value allows for larger constraint violations.

Example: `mpcobj.Optimizer.MixedIntegerOptions.DiscreteConstraintTolerance = 1e-5`

RoundingAtRootNode — Option to round the solution at the root node
1 (default) | 0

Option to round the solution at the root node, specified as a boolean. When `RoundingAtRootNode=1`, the solver rounds the solution of the relaxed QP problem solved at the root node of the search tree, so that discrete constraints are satisfied. Then, an additional QP is solved with respect to the remaining (continuous) variables. If such a QP has a feasible solution, the corresponding cost is used as a valid upper-bound on the optimal solution of the original mixed-integer problem. Having such an upper-bound may eliminate entire subtrees in the rest of the

execution of the solver and accelerate the solution of the following QP relaxations. Unless the number of iterations `MaxIterations` is small, it is worth setting `RoundingAtRootNode=1`. Otherwise, setting `RoundingAtRootNode=0` avoids solving the additional QP.

Example: `mpcobj.Optimizer.MixedIntegerOptions.RoundingAtRootNode = true`

`MaxPendingNodes` — Maximum number of pending nodes
1000 (default) | positive scalar

This is the maximum number of pending QP relaxations that can be stored. It determines the memory allocated to store all pending QP relaxations, which is proportional to (2*m +3*Nd)*`MaxPendingNodes`, where m is the number of inequality constraints, and Nd is the number of discrete variables. If the number of pending relaxations exceeds `MaxPendingNodes` then the solver is stopped with status code `-3`, `-4` or `-5`.

Example: `mpcobj.Optimizer.MixedIntegerOptions.MaxPendingNodes = 2000`

`MinOutputECR` — Minimum value allowed for output constraint ECR values
0 (default) | nonnegative scalar

Minimum value allowed for output constraint equal concern for relaxation (ECR) values, specified as a nonnegative scalar. A value of `0` indicates that hard output constraints are allowed. If either of the `OutputVariables.MinECR` or `OutputVariables.MaxECR` properties of an MPC controller are less than `MinOutputECR`, a warning is displayed and the value is raised to `MinOutputECR` during computation.

Example: `mpcobj.Optimizer.MinOutputECR = 1e-10`

`UseSuboptimalSolution` — Option indicating whether a suboptimal solution is acceptable
false (default) | true

Option indicating whether a suboptimal solution is acceptable, specified as a logical value. When the QP solver reaches the maximum number of iterations without finding a solution (the exit Option is `0`), the controller:

- Freezes the MV values if `UseSuboptimalSolution` is `false`
- Applies the suboptimal solution found by the solver after the final iteration if `UseSuboptimalSolution` is `true`

To specify the maximum number of iterations, depending on the value of `Algorithm`, use either `ActiveSetOptions.MaxIterations` or `InteriorPointOptions.MaxIterations`.

Example: `mpcobj.Optimizer.UseSuboptimalSolution = true`

`CustomSolver` — Option indicating whether to use a custom QP solver for simulation
false (default) | true

Option indicating whether to use a custom QP solver for simulation, specified as a logical value. If `CustomSolver` is `true`, the user must provide an `mpcCustomSolver` function on the MATLAB path.

This custom solver is not used for code generation. To generate code for a controller with a custom solver, use `CustomSolverCodeGen`.

If `CustomSolver` is `true`, the controller does not require the custom solver to honor the settings in either `ActiveSetOptions` or `InteriorPointOptions`.

You can also use the function `setCustomSolver` to automatically configure `mpcobj` to use the active-set algorithm of `quadprog` as a custom QP solver for both simulation and code generation.

For more information on using a custom QP solver see, "QP Solvers".

Example: `mpcobj.Optimizer.CustomSolver = true`

`CustomSolverCodeGen` — Option indicating whether to use a custom QP solver for code generation
`false` (default) | `true`

Option indicating whether to use a custom QP solver for code generation, specified as a logical value. If `CustomSolverCodeGen` is `true`, the user must provide an `mpcCustomSolverCodeGen` function on the MATLAB path.

This custom solver is not used for simulation. To simulate a controller with a custom solver, use `CustomSolver`.

You can also use the function `setCustomSolver` to automatically configure `mpcobj` to use the active-set algorithm of `quadprog` as a custom QP solver for both simulation and code generation.

For more information on using a custom QP solver see, "QP Solvers".

Example: `mpcobj.Optimizer.CustomSolverCodeGen = true`

`Notes` — User notes
`{}` (default) | cell array of character vectors

User notes associated with the MPC controller, specified as a cell array of character vectors.

Example: `mpcobj.Notes = {'Longitudinal Controller'; 'Version 2.1'}`

`UserData` — User data
`[]` (default) | any MATLAB data

User data associated with the MPC controller, specified as any MATLAB data, such as a cell array or structure.

Example: `mpcobj.UserData = {'Parameters',0.2,[3 4]'}`

`History` — Controller creation date and time
vector

This property is read-only.

Controller creation date and time, specified as a vector with the following elements:

- `History(1)` — Year
- `History(2)` — Month
- `History(3)` — Day
- `History(4)` — Hours
- `History(5)` — Minutes
- `History(6)` — Seconds

Use `datestr(mpcobj.History)` to display the controller creation date as a character vector.

Example: `mpcobj.History = datevec(now)`

Object Functions

buildMEX	Build MEX file that solves an MPC control problem
cloffset	Compute closed-loop DC gain from output disturbances to measured outputs assuming constraints are inactive at steady state
compare	Compare two MPC objects
generateExplicitMPC	Convert implicit MPC controller to explicit MPC controller
generatePlotParameters	Parameters for plotSection
generateExplicitOptions	Optimization options for explicit MPC generation
generateExplicitRange	Bounds on explicit MPC control law parameters
get	Get property values from MPC object
getCodeGenerationData	Create data structures for mpcmoveCodeGeneration
getEstimator	Obtain Kalman gains and model for estimator design
getconstraint	Obtain mixed input/output constraints from model predictive controller
getindist	Retrieve unmeasured input disturbance model
getname	Retrieve I/O signal names from MPC plant model
getoutdist	Retrieve unmeasured output disturbance model
mpcmove	Compute optimal control action and update controller states
mpcmoveAdaptive	Compute optimal control with prediction model updating
plot	Plot responses generated by MPC simulations
review	Examine MPC controller for design errors and stability problems at run time
sensitivity	Calculate the value of a performance metric and its sensitivity to the diagonal weights of an MPC controller
set	Set or modify MPC object properties
setCustomSolver	Configures an MPC object to use the QP solver from Optimization Toolbox as a custom solver
setEstimator	Modify a model predictive controller's state estimator
setconstraint	Set mixed input/output constraints for model predictive controller
setindist	Modify unmeasured input disturbance model
setname	Set I/O signal names in MPC plant model
setoutdist	Modify unmeasured output disturbance model
setterminal	Terminal weights and constraints
sim	Simulate an MPC controller in closed loop with a linear plant
size	Size and order of MPC Controller
ss	Convert unconstrained MPC controller to state-space linear system form
tf	Convert unconstrained MPC controller to linear transfer function form
trim	Compute steady-state value of MPC controller plant model state for given inputs and outputs

Examples

Create MPC Controller with Specified Prediction and Control Horizons

Create a plant model with the transfer function $(s + 1)/(s^2 + 2s)$.

```
Plant = tf([1 1],[1 2 0]);
```

The plant is SISO, so its input must be a manipulated variable and its output must be measured. In general, it is good practice to designate all plant signal types using either the `setmpcsignals` command, or the LTI `InputGroup` and `OutputGroup` properties.

Specify a sample time for the controller.

```
Ts = 0.1;
```

Define bounds on the manipulated variable, u, such that $-1 \le u \le 1$.

```
MV = struct(Min=-1,Max=1);
```

MV contains only the upper and lower bounds on the manipulated variable. In general, you can specify additional MV properties. When you do not specify other properties, their default values apply.

Specify a 20-interval prediction horizon and a 3-interval control horizon.

```
p = 20;
m = 3;
```

Create an MPC controller using the specified values. The fifth input argument is empty, so default tuning weights apply.

```
mpcobj = mpc(Plant,Ts,p,m,[],MV);
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Algorithms

To minimize computational overhead, model predictive controller creation occurs in two phases. The first happens at creation when you use the `mpc` function, or when you change a controller property. Creation includes basic validity and consistency checks, such as signal dimensions and nonnegativity of weights.

The second phase is initialization, which occurs when you use the object for the first time in a simulation or analytical procedure. Initialization computes all constant properties required for efficient numerical performance, such as matrices defining the optimal control problem and state estimator gains. Additional, diagnostic checks occur during initialization, such as verification that the controller states are observable.

By default, both phases display informative messages in the command window. You can turn these messages on or off using the `mpcverbosity` function.

Alternative Functionality

You can also create model predictive controllers using the **MPC Designer** app.

Version History
Introduced before R2006a

R2018b: Support for implementing economic MPC using a linear MPC controller has been removed
Errors starting in R2018b

Support for implementing economic MPC using a linear MPC controller has been removed. Implement economic MPC using a nonlinear MPC controller instead. For more information on nonlinear MPC controllers, see "Nonlinear MPC".

Update Code

If you previously saved a linear MPC object configured with custom cost or constraint functions, the software generates a warning when the object is loaded and an error if it is simulated. To suppress the error and warning messages and continue using your linear MPC controller, `mpcobj`, without the custom costs and constraints, set the `IsEconomicMPC` option to `false`.

```
mpcobj.IsEconomicMPC = false;
```

To implement your economic MPC controller using a nonlinear MPC object:

1 Create an `nlmpc` object.

2 Convert your custom cost function to the format required for nonlinear MPC. For more information on nonlinear MPC cost functions, see "Specify Cost Function for Nonlinear MPC".

3 Convert your custom constraint function to the format required for nonlinear MPC. For more information on nonlinear MPC constraints, see "Specify Constraints for Generic Nonlinear MPC".

4 Implement your linear prediction model using state and output functions. For more information on nonlinear MPC prediction models, see "Specify Prediction Model for Nonlinear MPC".

See Also

Apps
MPC Designer

Functions
setmpcsignals | mpcprops | mpcverbosity | review | cloffset | getEstimator | setEstimator | buildMEX | mpcmove | mpcmoveCodeGeneration | mpcmoveAdaptive | mpcmoveMultiple | sim

Objects
mpcstate | mpcsimopt | mpcmoveopt | explicitMPC

Blocks
MPC Controller | Adaptive MPC Controller | Multiple MPC Controllers

Topics
"Design MPC Controller at the Command Line"
"Model Predictive Control of Multi-Input Single-Output Plant"
"Choose Sample Time and Horizons"
"Tune Weights"
"Specify Constraints"
"MPC Signal Types"
"MPC Prediction Models"
"What is Model Predictive Control?"

mpcmoveopt

Option set for `mpcmove` function

Description

To specify options for the `mpcmove`, `mpcmoveAdaptive`, and `mpcmoveMultiple` functions, use an `mpcmoveopt` object.

Using this object, you can specify run-time values for a subset of controller properties, such as tuning weights and constraints. If you do not specify a value for one of the `mpcmoveopt` properties, the value of the corresponding controller option is used instead.

Creation

Syntax

```
options = mpcmoveopt
```

Description

`options = mpcmoveopt` creates a default set of options for the `mpcmove` function. To modify the property values, use dot notation.

Properties

`OutputWeights` — Output variable tuning weights
[] (default) | vector | array

Output variable tuning weights that replace the `Weights.OutputVariables` property of the controller at run time, specified as a vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

The format of `OutputWeights` must match the format of the `Weights.OutputVariables` property of the controller object. For example, you cannot specify constant weights across the prediction horizon in the controller object, and then specify time-varying weights using `mpcmoveopt`.

`MVWeights` — Manipulated variable tuning weights
[] (default) | vector | array

Manipulated variable tuning weights that replace the `Weights.ManipulatedVariables` property of the controller at run time, specified as a vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

The format of `MVWeights` must match the format of the `Weights.ManipulatedVariables` property of the controller object. For example, you cannot specify constant weights across the prediction horizon in the controller object, and then specify time-varying weights using `mpcmoveopt`.

MVRateWeights — Manipulated variable rate tuning weights
[ ] (default) | vector | array

Manipulated variable rate tuning weights that replace the `Weights.ManipulatedVariablesRate` property of the controller at run time, specified as a vector or array of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable rate tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

The format of `MVRateWeights` must match the format of the `Weights.ManipulatedVariablesRate` property of the controller object. For example, you cannot specify constant weights across the prediction horizon in the controller object, and then specify time-varying weights using `mpcmoveopt`.

ECRWeight — Slack variable tuning weight
[ ] (default) | positive scalar

Slack variable tuning weight that replaces the `Weights.ECR` property of the controller at run time, specified as a positive scalar.

OutputMin — Output variable lower bounds
[ ] (default) | row vector | matrix

Output variable lower bounds, specified as a row vector of length N_y or as a matrix with N_y columns, where N_y is the number of output variables.

If you did not specify the `OutputVariables(i).Min` property of the `mpc` object, then specifying `OutputMin` results in an error when you execute `mpcmove`.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row are used for the remaining steps of the prediction horizon.

OutputMin(:,i) replaces the OutputVariables(i).Min property of the mpc object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar OutputVariables(i).Min in the mpc object (a constant bound for the ith plant output to be applied to all prediction steps)

OutputMin Dimension	Replacement Behavior
Scalar OutputMin (single output, constant bound)	OutputMin replaces the constant bound defined in OutputVariables(i).Min
Column vector OutputMin (single output, time-varying bound)	OutputMin replaces the constant bound defined in OutputVariables(i).Min with a time-varying bo
Row vector OutputMin (multiple outputs, constant bounds)	OutputMin(i) replaces the constant bound defined OutputVariables(i).Min
Matrix OutputMin (multiple outputs, time-varying bounds)	OutputMin(:,i) replaces the constant bound defi OutputVariables(i).Min with a time-varying bo

Vector OutputVariables(i).Min in the mpc object (a time-varying bound for the ith plant output with different values at different prediction steps)

OutputMin Dimension	Replacement Behavior
Scalar OutputMin (single output, constant bound)	OutputMin replaces the first finite entry in OutputVariables.Min and the remaining entries OutputVariables.Min shift up or down with the sa amount of displacement to retain the profile defined original OutputVariables.Min vector.
Column vector OutputMin (single output, time-varying bound)	OutputMin replaces the time-varying bound defined OutputVariables(i).Min, and the original bound discarded.
Row vector OutputMin (multiple outputs, constant bounds)	OutputMin(i) replaces the first finite entry in OutputVariables(i).Min and the remaining en OutputVariables(i).Min shift up or down with th amount of displacement to retain the profile defined original OutputVariables(i).Min vector.
Matrix OutputMin (multiple outputs, time-varying bounds).	OutputMin(:,i) replaces the time-varying bound d OutputVariables(i).Min, and the original bound discarded.

OutputMax — Output variable upper bounds
[] (default) | row vector | matrix

Output variable upper bounds, specified as a row vector of length N_y or as a matrix with N_y columns, where N_y is the number of output variables.

If you did not specify the OutputVariables(i).Max property of the mpc object, then specifying OutputMax results in an error when you execute mpcmove.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row are used for the remaining steps of the prediction horizon.

OutputMax(:,i) replaces the OutputVariables(i).Max property of the mpc object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar OutputVariables(i).Max in the mpc object (a constant bound for the ith plant output to be applied to all prediction steps)

OutputMax Dimension	Replacement Behavior
Scalar OutputMax (single output, constant bound)	OutputMax replaces the constant bound defined in OutputVariables(i).Max
Column vector OutputMax (single output, time-varying bound)	OutputMax replaces the constant bound defined in OutputVariables(i).Max with a time-varying boun
Row vector OutputMax (multiple outputs, constant bounds)	OutputMax(i) replaces the constant bound defined i OutputVariables(i).Max
Matrix OutputMax (multiple outputs, time-varying bounds)	OutputMax(:,i) replaces the constant bound define OutputVariables(i).Max with a time-varying boun

Vector OutputVariables(i).Max in the mpc object (a time-varying bound for the ith plant output with different values at different prediction steps)

OutputMax Dimension	Replacement Behavior
Scalar OutputMax (single output, constant bound)	OutputMax replaces the first finite entry in OutputVariables.Max and the remaining entries OutputVariables.Max shift up or down with the sa amount of displacement to retain the profile defined b original OutputVariables.Max vector.
Column vector OutputMax (single output, time-varying bound)	OutputMax replaces the time-varying bound defined i OutputVariables(i).Max, and the original bound discarded.
Row vector OutputMax (multiple outputs, constant bounds)	OutputMax(i) replaces the first finite entry in OutputVariables(i).Max and the remaining ent OutputVariables(i).Max shift up or down with the amount of displacement to retain the profile defined b original OutputVariables(i).Max vector.
Matrix OutputMax (multiple outputs, time-varying bounds).	OutputMax(:,i) replaces the time-varying bound de OutputVariables(i).Max, and the original bound discarded.

MVMin — Manipulated variable lower bounds
[] (default) | row vector | matrix

Manipulated variable lower bounds, specified as a row vector of length N_{mv} or as a matrix with N_{mv} columns, where N_{mv} is the number of output variables.

If you did not specify the ManipulatedVariables(i).Min property of the mpc object, then specifying MVMin results in an error when you execute mpcmove.

To change the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row are used for the remaining steps of the prediction horizon.

MVMin(:,i) replaces the ManipulatedVariables(i).Min property of the mpc object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar ManipulatedVariables(i).Min in the mpc object (a constant bound for the ith manipulated variable to be applied to all prediction steps)

MVMin Dimension	Replacement Behavior
Scalar MVMin (single output, constant bound)	MVMin replaces the constant bound defined in ManipulatedVariables(i).Min
Column vector MVMin (single output, time-varying bound)	MVMin replaces the constant bound defined in ManipulatedVariables(i).Min with a time-varying bo
Row vector MVMin (multiple outputs, constant bounds)	MVMin(i) replaces the constant bound defined in ManipulatedVariables(i).Min
Matrix MVMin (multiple outputs, time-varying bounds)	MVMin(:,i) replaces the constant bound defined in ManipulatedVariables(i).Min with a time-varying bo

Vector ManipulatedVariables(i).Min in the mpc object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

MVMin Dimension	Replacement Behavior
Scalar MVMin (single output, constant bound)	MVMin replaces the first finite entry in ManipulatedVariables.Min and the remaining entrie ManipulatedVariables.Min shift up or down with the sa amount of displacement to retain the profile defined by the original ManipulatedVariables.Min vector.
Column vector MVMin (single output, time-varying bound)	MVMin replaces the time-varying bound defined in ManipulatedVariables(i).Min, and the original bound profile is discarded.
Row vector MVMin (multiple outputs, constant bounds)	MVMin(i) replaces the first finite entry in ManipulatedVariables(i).Min and the remaining en in ManipulatedVariables(i).Min shift up or down with same amount of displacement to retain the profile defined b the original ManipulatedVariables(i).Min vector.
Matrix MVMin (multiple outputs, time-varying bounds).	MVMin(:,i) replaces the time-varying bound defined in ManipulatedVariables(i).Min, and the original bound profile is discarded.

MVMax — Manipulated variable upper bounds
[] (default) | row vector | matrix

Manipulated variable upper bounds, specified as a row vector of length N_{mv} or as a matrix with N_{mv} columns, where N_{mv} is the number of output variables.

If you did not specify the ManipulatedVariables(i).Max property of the mpc object, then specifying MVMax results in an error when you execute mpcmove.

To change the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row are used for the remaining steps of the prediction horizon.

MVMax(:,i) replaces the ManipulatedVariables(i).Max property of the mpc object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar ManipulatedVariables(i).Max in the mpc object (a constant bound for the ith manipulated variable to be applied to all prediction steps)

MVMax Dimension	Replacement Behavior
Scalar MVMax (single output, constant bound)	MVMax replaces the constant bound defined in ManipulatedVariables(i).Max
Column vector MVMax (single output, time-varying bound)	MVMax replaces the constant bound defined in ManipulatedVariables(i).Max with a time-varying b
Row vector MVMax (multiple outputs, constant bounds)	MVMax(i) replaces the constant bound defined in ManipulatedVariables(i).Max
Matrix MVMax (multiple outputs, time-varying bounds)	MVMax(:,i) replaces the constant bound defined in ManipulatedVariables(i).Max with a time-varying b

Vector ManipulatedVariables(i).Max in the mpc object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

MVMax Dimension	Replacement Behavior
Scalar MVMax (single output, constant bound)	MVMax replaces the first finite entry in ManipulatedVariables.Max and the remaining en ManipulatedVariables.Max shift up or down with th amount of displacement to retain the profile defined by t original ManipulatedVariables.Max vector.
Column vector MVMax (single output, time-varying bound)	MVMax replaces the time-varying bound defined in ManipulatedVariables(i).Max, and the original bou profile is discarded.
Row vector MVMax (multiple outputs, constant bounds)	MVMax(i) replaces the first finite entry in ManipulatedVariables(i).Max and the remaining entries in ManipulatedVariables(i).Max shift up or with the same amount of displacement to retain the prof defined by the original ManipulatedVariables(i).M vector.
Matrix MVMax (multiple outputs, time-varying bounds).	MVMax(:,i) replaces the time-varying bound defined in ManipulatedVariables(i).Max, and the original bou profile is discarded.

CustomConstraint — Custom mixed input/output constraints
[] (default) | structure

Custom mixed input/output constraints, specified as a structure with the following fields. These constraints replace the mixed input/output constraints previously set using setconstraint.

E — Manipulated variable constraint constant
array of zeros (default) | N_c-by-N_{mv} array

Manipulated variable constraint constant, specified as an N_c-by-N_{mv} array, where N_c is the number of constraints, and N_{mv} is the number of manipulated variables.

F — Controlled output constraint constant
array of zeros (default) | N_c-by-N_y array

Controlled output constraint constant, specified as an N_c-by-N_y array, where N_y is the number of controlled outputs (measured and unmeasured).

G — Mixed input/output constraint constant
column vector of zeros (default) | column vector of length N_c

Mixed input/output constraint constant, specified as a column vector of length N_c.

S — Measured disturbance constraint constant
array of zeros (default) | N_c-by-N_{md} array

Measured disturbance constraint constant, specified as an N_c-by-N_{md} array, where N_{md} is the number of measured disturbances.

OnlyComputeCost — Option to return only the value of the cost function in the solution details structure
0 (default) | 1

Option to return only the value of the cost function in the info solution details output structure returned by mpcmove. Specify it as one of the following:

- 0 — mpcmove returns all the detailed solution information in its second output argument.
- 1 — mpcmove returns only the predicted value of the cost function. This saves some computational effort as the predicted optimal plant state and output sequences do not need to be calculated. Note that if mpcmove is not called with a second output argument these sequences are not calculated either.

MVused — Manipulated variable values used in the plant during the previous control interval
[] (default) | row vector

Manipulated variable values used in the plant during the previous control interval, specified as a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables. If you do not specify MVused, the mpvmove uses the LastMove property of its current controller state input argument, x.

MVTarget — Manipulated variable targets
[] (default) | row vector

Manipulated variable targets, specified as a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables. MVTarget(i) replaces the ManipulatedVariables(i).Target property of the controller at run time.

PredictionHorizon — Prediction horizon
[] (default) | positive integer

Prediction horizon, which replaces the PredictionHorizon property of the controller at run time, specified as a positive integer. If you specify PredictionHorizon, you must also specify ControlHorizon.

Specifying PredictionHorizon changes the:

- Number of rows in the optimal sequences returned by the `mpcmove` and `mpcmoveAdaptive` functions
- Maximum dimensions of the `Plant` and `Nominal` input arguments of `mpcmoveAdaptive`

This parameter is ignored by the `mpcmoveMultiple` function.

`ControlHorizon` — Control horizon
[ ] (default) | positive integer | vector of positive integers

Control horizon, which replaces the `ControlHorizon` property of the controller at run time, specified as one of the following:

- Positive integer, *m*, between 1 and *p*, inclusive, where *p* is equal to `PredictionHorizon`. In this case, the controller computes *m* free control moves occurring at times *k* through *k+m*-1, and holds the controller output constant for the remaining prediction horizon steps from *k+m* through *k+p*-1. Here, *k* is the current control interval. For optimal trajectory planning set *m* equal to *p*.
- Vector of positive integers, [m_1, m_2, ...], where the sum of the integers equals the prediction horizon, *p*. In this case, the controller computes *M* blocks of free moves, where *M* is the length of the `ControlHorizon` vector. The first free move applies to times *k* through $k+m_1$-1, the second free move applies from time $k+m_1$ through $k+m_1+m_2$-1, and so on. Using block moves can improve the robustness of your controller compared to the default case.

If you specify `ControlHorizon`, you must also specify `PredictionHorizon`.

This parameter is ignored by the `mpcmoveMultiple` function.

Object Functions

mpcmove   Compute optimal control action and update controller states
mpcmoveAdaptive   Compute optimal control with prediction model updating
mpcmoveMultiple   Compute gain-scheduling MPC control action at a single time instant

Examples

Simulation with Varying Controller Property

Vary a manipulated variable upper bound during a simulation.

Define the plant, which includes a 4-second input delay. Convert to a delay-free, discrete, state-space model using a 2-second control interval. Create the corresponding default controller, and specify MV bounds at +/-2.

```
Ts = 2;
Plant = absorbDelay(c2d(ss(tf(0.8,[5 1],InputDelay=4)),Ts));
mpcobj = mpc(Plant,Ts);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

```
mpcobj.MV(1).Min = -2;
mpcobj.MV(1).Max = 2;
```

Create an empty `mpcmoveopt` object. During simulation, you can set properties of the object to specify controller parameters.

```
options = mpcmoveopt;
```

Pre-allocate storage and initialize the controller state.

```
v = [];
t = [0:Ts:20];
N = length(t);
y = zeros(N,1);
u = zeros(N,1);
x = mpcstate(mpcobj);
```

```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Use `mpcmove` to simulate the following:

- Reference (setpoint) step change from initial condition $r = 0$ to $r = 1$ (servo response)
- MV upper bound step decrease from 2 to 1, occurring at $t = 10$

```
r = 1;
for i = 1:N
    y(i) = Plant.C*x.Plant;
    if t(i) >= 10
        options.MVMax = 1;
    end
    [u(i),Info] = mpcmove(mpcobj,x,y(i),r,v,options);
end
```

As the loop executes, the value of `options.MVMax` is reset to 1 for all iterations that occur after $t = 10$. Prior to that iteration, `options.MVMax` is empty. Therefore, the controller's value for `MVMax` is used, `mpcobj.MV(1).Max = 2`.

Plot the results of the simulation.

```
[Ts,us] = stairs(t,u);
plot(Ts,us,'b-',t,y,'r-')
legend('MV','OV')
xlabel(sprintf('Time, %s',Plant.TimeUnit))
```
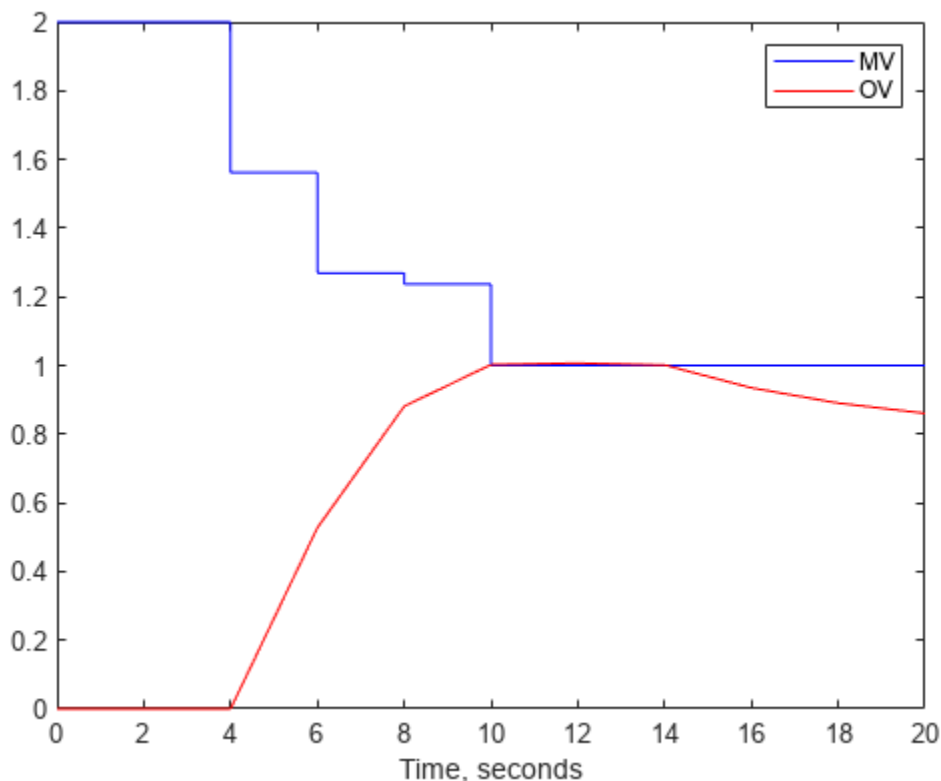
From the plot, you can observe that the original MV upper bound is active until $t = 4$. After the input delay of 4 seconds, the output variable (OV) moves smoothly to its new target of $r = 1$. reaching the target at $t = 10$. The new MV bound imposed at $t = 10$ becomes active immediately. This forces the OV below its target, after the input delay elapses.

Now assume that you want to impose an OV upper bound at a specified location relative to the OV target. Consider the following constraint design command:

```
mpcobj.OV(1).Max = [Inf,Inf,0.4,0.3,0.2];
```

This is a horizon-varying constraint. The known input delay makes it impossible for the controller to satisfy an OV constraint prior to the third prediction-horizon step. Therefore, a finite constraint during the first two steps would be poor practice. For illustrative purposes, the previous constraint also decreases from 0.4 at step 3 to 0.2 at step 5 and thereafter.

The following commands produce the same results shown in the previous plot. The OV constraint is never active because it is being varied in concert with the setpoint, r.

```
x = mpcstate(mpcobj);
```

```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

```
OPTobj = mpcmoveopt;
for i = 1:N
    y(i) = Plant.C*x.Plant;
    if t(i) >= 10
```

```
        OPTobj.MVMax = 1;
    end
    OPTobj.OutputMax = r + 0.4;
    [u(i),Info] = mpcmove(mpcobj,x,y(i),r,v,OPTobj);
end
```

The scalar value *r* + 0.4 replaces the first finite value in the `mpcobj.OV(1).Max` vector, and the remaining finite values adjust to maintain the original profile, that is, the numerical difference between these values is unchanged. *r* = 1 for the simulation, so the previous use of the `mpcmoveopt` object is equivalent to the command

```
mpcobj.OV(1).Max = [Inf, Inf, 1.4, 1.3, 1.2];
```

However, using the `mpcmoveopt` object involves much less computational overhead.

Tips

- If a variable is unconstrained in the initial controller design, you cannot constrain it using `mpcmoveopt`. The controller ignores any such specifications.
- You cannot remove a constraint from a variable that is constrained in the initial controller design. However, you can change it to a large (or small) value such that it is unlikely to become active.

Version History
Introduced in R2018b

See Also

Functions
mpcmove | mpcmoveAdaptive | mpcmoveMultiple | setconstraint | setterminal

Objects
mpc

Topics
"Switching Controllers Based on Optimal Costs"
"Optimization Problem"
"Design MPC Controller at the Command Line"
"Model Predictive Control of Multi-Input Single-Output Plant"

mpcsimopt

MPC simulation options

Description

When simulating an implicit or explicit MPC controller using the `sim` function, you can specify additional simulation options using an `mpcsimopt` object.

Creation

Syntax

```
options = mpcsimopt;
```

Description

`options = mpcsimopt;` creates a default set of options for specifying additional parameters for simulating an MPC controller with the `sim` function. To specify nondefault values for the properties on page 3-37, use dot notation.

Properties

`PlantInitialState` — Simulation plant model initial state
[ ] (default) | vector

Simulation plant model initial state, specified as a vector with length equal to the number states in the plant model used for the simulation. To use the default nominal state of the simulation plant model, set `PlantInitialState` to [ ].

If you do not specify the `Model` option, then the plant model used for the simulation is the internal plant model from the controller. In this case, the default initial controller state is equal to `mpcobj.Model.Nominal.X`.

If you specify the `Model` option, then the plant model used for simulation is `Model.Plant`. In this case, the default initial controller state is equal to `Model.Nominal.X`.

`ControllerInitialState` — MPC controller initial condition
[ ] (default) | `mpcstate` object

MPC controller initial condition, specified as an `mpcstate` object. Setting `ControllerInitialState = []` is equivalent to setting `ControllerInitialState = mpcstate(mpcobj)`.

`UnmeasuredDisturbance` — Unmeasured disturbance signal
[ ] (default) | array

Unmeasured disturbance signal for simulating disturbances occurring at the unmeasured disturbance inputs of the simulation plant model, specified as an array with N_{ud} columns and up to N_t rows, where

N_{ud} is the number of unmeasured disturbances, and N_t is the number of simulation steps. If you specify fewer than N_t rows, then the values in the final row of the array are extended to the end of the simulation.

InputNoise — Manipulated variable noise signal
[ ] (default) | array

Manipulated variable noise signal for simulating load disturbances occurring at the manipulated variable inputs to the simulation plant model, specified as an array with N_{mv} columns and up to N_t rows, where N_{mv} is the number of manipulated variables, and N_t is the number of simulation steps. If you specify fewer than N_t rows, then the values in the final row of the array are extended to the end of the simulation.

OutputNoise — Measured output noise signal
[ ] (default) | array

Measured output noise signal for simulating disturbances occurring at the measured output of the simulation plant model, specified as an array with N_y columns and up to N_t rows, where N_y is the number of measured outputs, and N_t is the number of simulation steps. If you specify fewer than N_t rows, then the values in the final row of the array are extended to the end of the simulation.

RefLookAhead — Option to use reference previewing
'off' (default) | 'on'

Option to use reference previewing during simulation, specified as one of the following:

- 'off' — Do not use reference previewing.
- 'on' — Use reference previewing.

When simulating an explicit MPC controller, you must set RefLookAhead to 'off'.

MDLookAhead — Option to use measured disturbance previewing
'off' (default) | 'on'

Option to use measured disturbance previewing during simulation, specified as one of the following:

- 'off' — Do not use measured disturbance previewing.
- 'on' — Use measured disturbance previewing.

When simulating an explicit MPC controller, you must set MDLookAhead to 'off'.

Constraints — Enable constraints
'on' (default) | 'off'

Option to enable constraints during simulation, specified as one of the following:

- 'on' — Use the constraints defined in the controller during simulation.
- 'off' — Simulate the controller without any constraints.

Model — Plant model to use for simulation
[ ] (default) | LTI system object | structure

Plant model to use for simulation, specified as one of the following:

- [] — Simulate the controller against its internal plant model (mpcobj.Model). In this case, there is no plant-model mismatch.
- LTI system object — Simulate the controller against the specified LTI plant. The specified plant must have the same input and output group configuration as mpcobj.Model.Plant. To set this configuration, use setmpcsignals.
- Structure with fields Plant and Nominal — Simulate the controller using the specified plant (Plant) and nominal conditions (Nominal).

Model sets the actual plant (not the internal prediction model of the controller) to be used in closed-loop or open-loop simulations. To test the controller robustness against a plant model mismatch, use this option to specify a plant that differs from the controller internal plant model.

If you do not specify nominal conditions, Model.Nominal.U and Model.Nominal.Y are inherited from mpcobj.Model.Nominal. Model.Nominal.X and Model.Nominal.DX are only inherited if both plants are state-space objects with the same state dimension.

To specify the initial state of this simulation plant model, use the PlantInitialState option.

StatusBar — Display the simulation status bar
'off' (default) | 'on'

Option to display the simulation status bar, specified as one of the following:

- 'off' — Do not display the status bar.
- 'on' — Display the status bar.

MVSignal — Sequence of manipulated variables
[] (default) | array

Sequence of manipulated variables to use during open-loop simulation, specified as an array with N_{mv} columns and up to N_t rows, where N_{mv} is the number of manipulated variables, and N_t is the number of simulation steps. If you specify fewer than N_t rows, then the values in the final row of the array are extended to the end of the simulation. The specified manipulated variable signals must include any manipulated variable offsets. If MVSignal is an empty array (default) then a value of 0 is assumed for all the manipulated variables.

MVSignal is ignored if OpenLoop is set to 'off'.

OpenLoop — Run an open-loop simulation
'off' (default) | 'on'

Option to run an open-loop simulation, specified as one of the following:

- 'off' — Perform a closed-loop simulation.
- 'on' — Perform an open-loop simulation.

ShowOutputNoise — Include output noise in measured output plots
'off' (default) | 'on'

Option to include output noise in measured output plots, specified as one of the following:

- 'off' — Do not include output noise in plots
- 'on' — Include output noise in plots

Enable this option to view the output disturbance rejection behavior of the controller during closed-loop simulations.

Object Functions

sim   Simulate an MPC controller in closed loop with a linear plant

Examples

Simulate MPC Control with Plant Model Mismatch

Simulate the MPC control of a multi-input, multi-output (MIMO) system with a mismatch between the predicted and actual plant models. The system has two manipulated variables, two unmeasured disturbances, and two measured outputs.

Define the predicted plant model.

```
p1 = tf(1,[1 2 1])*[1 1;0 1];
plantPredict = ss([p1 p1]);
plantPredict.InputName = {'mv1','mv2','umd3','umd4'};
```

Specify the MPC signal types.

```
plantPredict = setmpcsignals(plantPredict,'MV',[1 2],'UD',[3 4]);
```

Create the MPC controller.

```
mpcobj = mpc(plantPredict,1,40,2);
```

```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Define the unmeasured input disturbance model used by the controller.

```
distModel = eye(2,2)*ss(-0.5,1,1,0);
mpcobj.Model.Disturbance = distModel;
```

Define an actual plant model which differs from the predicted model and has unforeseen unmeasured disturbance inputs.

```
p2 = tf(1.5,[0.1 1 2 1])*[1 1;0 1];
plantActual = ss([p2 p2 tf(1,[1 1])*[0;1]]);
plantActual = setmpcsignals(plantActual,'MV',[1 2],'UD',[3 4 5]);
```

Configure the unmeasured disturbance and output reference trajectories.

```
dist = ones(1,3);
refs = [1 2];
```

Create and configure a simulation option set.

```
options = mpcsimopt(mpcobj);
options.UnmeasuredDisturbance = dist;
options.Model = plantActual;
```

Simulate the system.

```
sim(mpcobj,20,refs,options)
```

```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
-->Converting model to discrete time.
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```
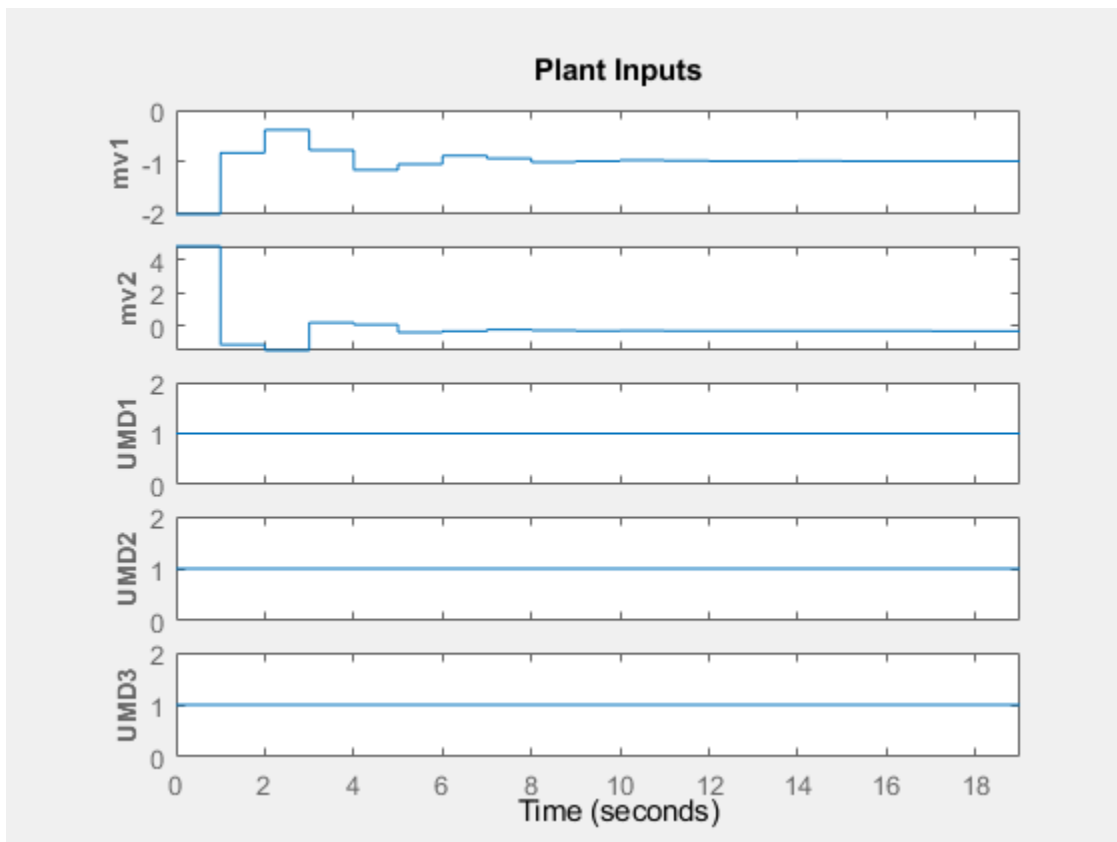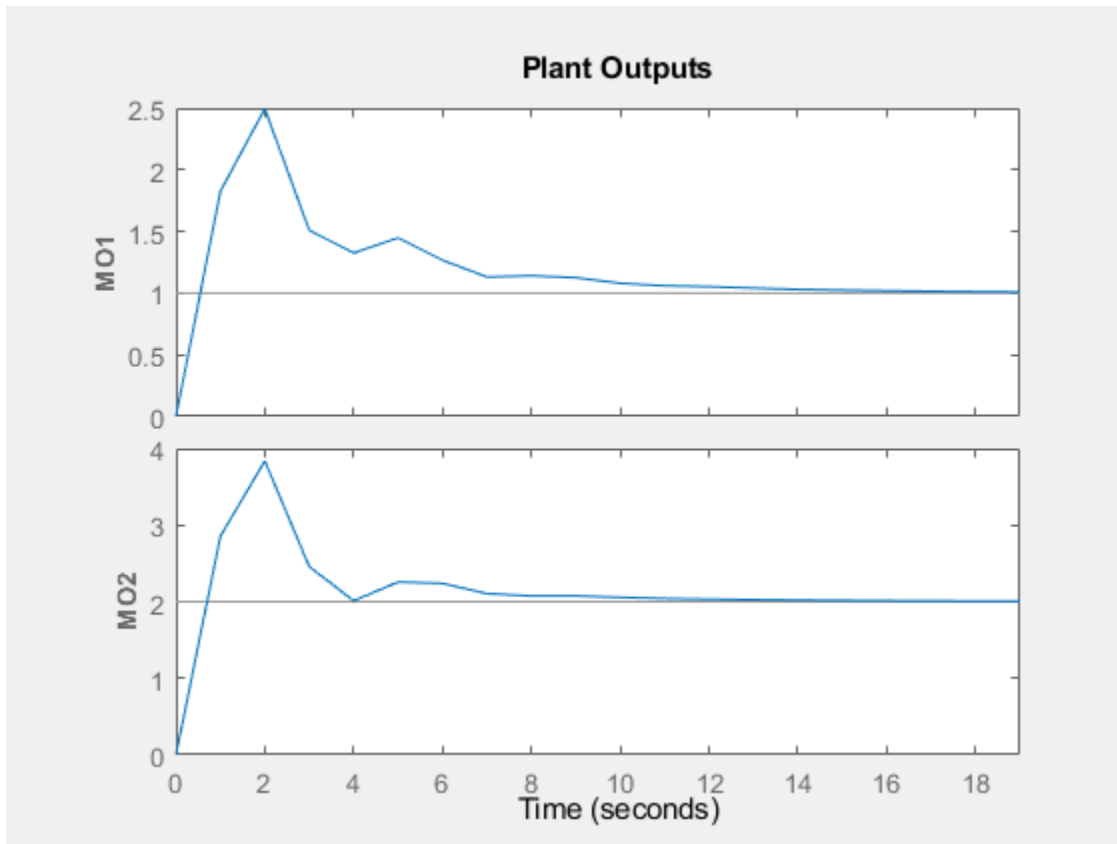
Version History

Introduced before R2006a

See Also

Functions
sim | plot

Objects
mpc

Topics
"Simulate Linear MPC Controller with Nonlinear Plant Using Successive Linearizations"

mpcstate

MPC controller state

Description

An `mpcstate` object is a handle object pointing to the current state of an implicit or explicit MPC controller. Use an `mpcstate` object to initialize and hold the current controller state within a MATLAB simulation.

The controller state includes the:

- States of the plant, disturbance, and noise models. These are the states of the built-in state estimator. For more information, see "Controller State Estimation".
- Manipulated variables used in the previous control interval.
- State covariance matrix of the state estimator.

Before simulation, use `mpcstate` to obtain a handle object and, if needed, set the initial condition of the controller state. By default the initial controller state is set to the nominal values specified in the related `mpc` object

During simulation, the controller move function (`mpcmove`, `mpcmoveExplicit`, `mpcmoveAdaptive` or `mpcmoveMultiple`), also updates the controller state when calculating a new control move. If you use built-in state estimation, then you do not need to modify the controller state at all. If you use custom state estimation then you must use the handle object to provide current states for the plant, disturbance, and noise models, before calling the controller move function.

After the simulation, the handle object reflects the state of the controller at the end of the simulation. For more information about handle objects, see "Handle Object Behavior".

Creation

Syntax

```
xc = mpcstate(mpcobj)
xc = mpcstate(mpcobj,plant,disturbance,noise,lastMove,covariance)
```

Description

`xc = mpcstate(mpcobj)` returns a handle object pointing to the state of `mpcobj` and sets the state object properties to their default values.

`xc = mpcstate(mpcobj,plant,disturbance,noise,lastMove,covariance)` sets the Properties on page 3-44 of the state object to specified nondefault values. To leave a specific property to its default value, set the corresponding input argument to `[]`.

Input Arguments

`mpcobj` — MPC controller object
mpc object | explicitMPC object

MPC controller object, specified as either an `mpc` or `explicitMPC` object.

Output Arguments

`xc` — Handle object pointing to current controller state
mpcstate object

Handle object pointing to current controller state, returned as an `mpcstate` object, with the following "Properties" on page 3-44.

Note Since `xc` is a handle object, if you copy it to a new variable, the new variable still points to the current state of the same `mpc` object. If this state changes, this is reflected in both `xc` and its copy. To preserve the original state for later use, you can save `xc` to a MAT-file. During simulation, the controller move function also updates `xc` when a new control move is calculated. For more information, see `mpcmove`. For more information about handle objects, see "Handle Object Behavior".

Properties

`Plant` — Plant model state estimates
vector

Plant model state estimates, specified as a vector. The plant state estimate values are in engineering units and are absolute; that is, they include state offsets. By default, the `Plant` property is equal to the `Model.Nominal.X` property of the controller used to create the `mpcstate` object.

If the controller plant model includes delays, the `Plant` property includes states that model the delays. Therefore the number of elements in `Plant` is greater than the order of the nondelayed controller plant model.

During a simulation, if you are using custom state estimation, prior to calling the MPC controller move function (`mpcmove`, `mpcmoveExplicit`, `mpcmoveAdaptive` or `mpcmoveMultiple`), you must set `xc.Plant` to its best estimate. If you are using built-in state estimation, the MPC controller move function uses a Kalman filter to calculate this estimate and to set `xc.Plant`, and therefore you do not need to update this property at all.

`Disturbance` — Disturbance model state estimates
vector

Disturbance model state estimates, specified as a vector. The disturbance state estimates include the states of the input disturbance model followed by the states of the output disturbance model. By default, the `Disturbance` property is a zero vector if the controller has disturbance model states and empty otherwise.

To view the input and output disturbance models of your controller, use the `getindist` and `getoutdist` functions, respectively.

During a simulation, if you are using custom state estimation, prior to calling the function that calculates the MPC controller move, you must set `xc.Disturbance` to its best estimate. If you are

using built-in state estimation, the MPC controller move function uses a Kalman filter to calculate this estimate and to set `xc.Disturbance`, and therefore you do not need to update this property at all.

`Noise` — Output measurement noise model state estimates
vector

Output measurement noise model state estimates, specified as a vector. By default, the `Noise` property is a zero vector if the controller has noise model states and empty otherwise.

During a simulation, if you are using custom state estimation, prior to calling the function that calculates the MPC controller move, you must set `xc.Noise` to its best estimate. If you are using built-in state estimation, the MPC controller move function uses a Kalman filter to calculate this estimate and to set `xc.Noise`, and therefore you do not need to update this property at all.

`LastMove` — Optimal manipulated variable control move from previous control interval
vector

Optimal manipulated variable control move from previous control interval, specified as a vector with length equal to the number of manipulated variables. By default, the `LastMove` property is equal to the nominal values of the manipulated variables.

During a simulation, this property is automatically set by the function that calculates the MPC controller move; therefore you do not need to update it.

When the actual control signals sent to the plant in the previous control interval do not match the calculated optimal value, do not use `LastMove` to specify the actual control signals. Instead, do so using the `MVused` property of an `mpcmoveopt` object. The `LastMove` property of the `mpcstate` object is used to store the optimal control move at the previous control interval, and the controller uses the difference between `LastMove` and `MVused` to correct the internal state estimation.

`Covariance` — Covariance matrix for controller state estimates
symmetrical matrix

Covariance matrix for controller state estimates, specified as an N_s-by-N_s symmetric matrix, where N_s is the sum of the number states contained in the `Plant`, `Disturbance`, and `Noise` fields.

If the controller is employing default state estimation the default covariance matrix is the steady-state covariance computed according to the assumptions in "Controller State Estimation". This matrix is static when the controller is time-invariant. For more information, see the description of the `P` output argument of the `kalmd` function.

During a MATLAB simulation, do not modify `xc.Covariance`. If the controller uses custom state estimation, this property is empty and it is not used. Otherwise, `xc.Covariance` is automatically updated by the function that calculates the MPC controller move.

Object Functions

mpcmove	Compute optimal control action and update controller states
mpcmoveAdaptive	Compute optimal control with prediction model updating
mpcmoveMultiple	Compute gain-scheduling MPC control action at a single time instant
mpcmoveExplicit	Compute optimal control using explicit MPC

Examples

Get Handle to Controller State Object

Create a model predictive controller for a single-input-single-output (SISO) plant. For this example, the plant includes an input delay of `0.4` time units, and the control sample time is `0.2` time units.

```
P = tf(1,[10 1],'InputDelay',0.4);
mpcobj = mpc(P,0.2);
```

```
-->"PredictionHorizon" is empty. Assuming default 10.
-->"ControlHorizon" is empty. Assuming default 2.
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Obtain a handle object pointing to the current controller state, initializing all controller states to their default values.

```
xmpc = mpcstate(mpcobj)
```

```
-->Converting the "Model.Plant" property to state-space.
-->Converting model to discrete time.
-->Converting delays to states.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
MPCSTATE object with fields
          Plant: [0 0 0]
    Disturbance: 0
          Noise: [1x0 double]
       LastMove: 0
     Covariance: [4x4 double]
```

The controller state, which is an `mpcstate` object, contains the controller states only at the current time. Specifically:

- `xmpc.Plant` is the current value of the estimated plant states.
- `xmpc.Disturbance` is the current value of the disturbance model states.
- `xmpc.Noise` is the current value of the noise model states.
- `xmpc.LastMove` is the last value of the manipulated variable.
- `xmpc.Covariance` is the current value of the estimator covariance matrix.

In this example, the `Plant` property of the `mpcstate` object contains the state of the first-order, discretized, transfer function P as well as two additional states to model the two delay intervals. By default, the controller contains a first-order output disturbance model (the `Disturbance` property is a scalar) and a static gain noise model (the `Noise` property is empty).

You can access the properties of the controller state object using dot notation. For example, view the default covariance matrix.

```
xmpc.Covariance
```

```
ans = 4×4

    0.0624    0.0000    0.0000   -0.0224
    0.0000    1.0000    0.0000   -0.0000
    0.0000    0.0000    1.0000    0.0000
   -0.0224   -0.0000    0.0000    0.2301
```

Note that `xmpc` is a handle object, which always points to the current state of the controller.

```
isa(xmpc,'handle')
```

```
ans = logical
   1
```

If you copy an `mpcstate` object into a new variable, the new variable still points to the current state of the same `mpc` object.

```
xmpc_copy = xmpc;
xmpc.Disturbance = 0.1;
xmpc.Plant = [0.2 0.3 0.4];
xmpc_copy
```

```
MPCSTATE object with fields
          Plant: [0.2000 0.3000 0.4000]
    Disturbance: 0.1000
          Noise: [1x0 double]
       LastMove: 0
     Covariance: [4x4 double]
```

During simulation, `mpcmove` updates the internal plant state when a new control move is calculated. Therefore, in general, you do not need to update `xmpc`, which always points to the current (hence updated) state.

Compute the controller action and update the internal controller states assuming the output signal is `0` and the reference signal is `1`.

```
u = mpcmove(mpcobj,xmpc,0,1)
```

```
u = 5.6711
```

```
xmpc
```

```
MPCSTATE object with fields
          Plant: [0.2105 0.4000 5.6711]
    Disturbance: 0.0674
          Noise: [1x0 double]
       LastMove: 5.6711
     Covariance: [4x4 double]
```

```
xmpc_copy
```

```
MPCSTATE object with fields
          Plant: [0.2105 0.4000 5.6711]
    Disturbance: 0.0674
          Noise: [1x0 double]
       LastMove: 5.6711
     Covariance: [4x4 double]
```

For more information about handle objects, see "Handle Object Behavior".

Version History
Introduced before R2006a

See Also

Functions
getoutdist | setoutdist | setindist | getindist | getEstimator | setEstimator | mpcmove | mpcmoveAdaptive | mpcmoveMultiple | mpcmoveExplicit

Objects
mpc

nlmpc

Nonlinear model predictive controller

Description

A nonlinear model predictive controller computes optimal control moves across the prediction horizon using a nonlinear prediction model, a nonlinear cost function, and nonlinear constraints. For more information on nonlinear MPC, see "Nonlinear MPC".

Creation

Syntax

```
nlobj = nlmpc(nx,ny,nu)

nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex)
nlobj = nlmpc(nx,ny,'MV',mvIndex,'UD',udIndex)
nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex,'UD',udIndex)
```

Description

`nlobj = nlmpc(nx,ny,nu)` creates an `nlmpc` object whose prediction model has `nx` states, `ny` outputs, and `nu` inputs, where all inputs are manipulated variables. Use this syntax if your model has no measured or unmeasured disturbance inputs.

`nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex)` creates an `nlmpc` object whose prediction model has measured disturbance inputs. Specify the input indices for the manipulated variables, `mvIndex`, and measured disturbances, `mdIndex`.

`nlobj = nlmpc(nx,ny,'MV',mvIndex,'UD',udIndex)` creates an `nlmpc` object whose prediction model has unmeasured disturbance inputs. Specify the input indices for the manipulated variables and unmeasured disturbances, `udIndex`.

`nlobj = nlmpc(nx,ny,'MV',mvIndex,'MD',mdIndex,'UD',udIndex)` creates an `nlmpc` object whose prediction model has both measured and unmeasured disturbance inputs. Specify the input indices for the manipulated variables, measured disturbances, and unmeasured disturbances.

Input Arguments

nx — Number of prediction model states
positive integer

Number of prediction model states, specified as a positive integer. This value is stored in the `Dimensions.NumberOfStates` controller read-only property. You cannot change the number of states after creating the controller object.

Example: 6

ny — Number of prediction model outputs
positive integer

Number of prediction model outputs, specified as a positive integer. This value is stored in the `Dimensions.NumberOfOutputs` controller read-only property. You cannot change the number of outputs after creating the controller object.

Example: 2

nu — Number of prediction model inputs
positive integer

Number of prediction model inputs, which are all set to be manipulated variables, specified as a positive integer. This value is stored in the `Dimensions.NumberOfInputs` controller read-only property. You cannot change the number of manipulated variables after creating the controller object.

Example: 4

mvIndex — Manipulated variable indices
vector of positive integers

Manipulated variable indices, specified as a vector of positive integers. This value is stored in the `Dimensions.MVIndex` controller read-only property. You cannot change these indices after creating the controller object.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from 1 through N_u, where N_u is the number of prediction model inputs.

Example: [1 3]

mdIndex — Measured disturbance indices
vector of positive integers

Measured disturbance indices, specified as a vector of positive integers. This value is stored in the `Dimensions.MDIndex` controller read-only property. You cannot change these indices after creating the controller object.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from 1 through N_u, where N_u is the number of prediction model inputs.

Example: 2

udIndex — Unmeasured disturbance indices
vector of positive integers

Unmeasured disturbance indices, specified as a vector of positive integers. This value is stored in the `Dimensions.UDIndex` controller read-only property. You cannot change these indices after creating the controller object.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from 1 through N_u, where N_u is the number of prediction model inputs.

Example: 4

Properties

`Ts` — Prediction model sample time
1 (default) | positive finite scalar

Prediction model sample time, specified as a positive finite scalar. The controller uses a discrete-time model with a sample time of `Ts` for prediction. If you specify a continuous-time prediction model (`Model.IsContinuousTime` is `true`), then the controller discretizes the model using the built-in implicit trapezoidal rule with a sample time of `Ts`.

Example: `0.1`

`PredictionHorizon` — Prediction horizon
10 (default) | positive integer

Prediction horizon steps, specified as a positive integer. The product of `PredictionHorizon` and `Ts` is the prediction time, that is, how far the controller looks into the future.

Example: 15

`ControlHorizon` — Control horizon
2 (default) | positive integer | vector of positive integers

Control horizon, specified as one of the following:

- Positive integer, m, between 1 and p, inclusive, where p is equal to `PredictionHorizon`. In this case, the controller computes m free control moves occurring at times k through $k+m-1$, and holds the controller output constant for the remaining prediction horizon steps from $k+m$ through $k+p-1$. Here, k is the current control interval.

- Vector of positive integers $[m_1, m_2, ...]$, specifying the lengths of blocking intervals. By default the controller computes M blocks of free moves, where M is the number of blocking intervals. The first free move applies to times k through $k+m_1-1$, the second free move applies from time $k+m_1$ through $k+m_1+m_2-1$, and so on. Using block moves can improve the robustness of your controller. The sum of the values in `ControlHorizon` must match the prediction horizon p. If you specify a vector whose sum is:

  - Less than the prediction horizon, then the controller adds a blocking interval. The length of this interval is such that the sum of the interval lengths is p. For example, if $p=10$ and you specify a control horizon of `ControlHorizon=[1 2 3]`, then the controller uses four intervals with lengths `[1 2 3 4]`.

  - Greater than the prediction horizon, then the intervals are truncated until the sum of the interval lengths is equal to p. For example, if $p=10$ and you specify a control horizon of `ControlHorizon= [1 2 3 6 7]`, then the controller uses four intervals with lengths `[1 2 3 4]`.

Piecewise constant blocking moves are often too restrictive for optimal path planning applications. To produce a less-restrictive, better-conditioned nonlinear programming problem, you can specify piecewise linear manipulated variable blocking intervals. To do so, set the `Optimization.MVInterpolationOrder` property of your `nlmpc` controller object to `1`.

For more information on how manipulated variable blocking works with different interpolation methods, see "Manipulated Variable Blocking".

Example: 3

Dimensions — Prediction model dimensional information
structure

This property is read-only.

Prediction model dimensional information, specified when you create the controller and stored as a structure with the following fields.

NumberOfStates — Number of states
positive integer

Number of states in the prediction model, specified as a positive integer. This value corresponds to nx.

Example: 6

NumberOfOutputs — Number of outputs
positive integer

Number of outputs in the prediction model, specified as a positive integer. This value corresponds to ny.

Example: 1

NumberOfInputs — Number of inputs
positive integer

Number of inputs in the prediction model, specified as a positive integer. This value corresponds to either nu or the sum of the lengths of mvIndex, mdIndex, and udIndex.

Example: 3

MVIndex — Manipulated variable indices
vector of positive integers

Manipulated variable indices for the prediction model, specified as a vector of positive integers. This value corresponds to mvIndex.

Example: [1 2]

MDIndex — Measured disturbance indices
vector of positive integers

Measured disturbance indices for the prediction model, specified as a vector of positive integers. This value corresponds to mdIndex.

Example: 4

UDIndex — Unmeasured disturbance indices
vector of positive integers

Unmeasured disturbance indices for the prediction model, specified as a vector of positive integers. This value corresponds to udIndex.

Example: 3

Model — Prediction model
structure

Prediction model, specified as a structure with the following fields.

StateFcn — State function
string | character vector | function handle

State function, specified as a string, character vector, or function handle. For a continuous-time prediction model, `StateFcn` is the state derivative function. For a discrete-time prediction model, `StateFcn` is the state update function.

If your state function is continuous-time, the controller automatically discretizes the model using the implicit trapezoidal rule. This method can handle moderately stiff models, and its prediction accuracy depends on the controller sample time `Ts`; that is, a large sample time leads to inaccurate prediction.

If the default discretization method does not provide satisfactory prediction for your application, you can specify your own discrete-time prediction model that uses a different method, such as the multistep forward Euler rule.

You can specify your state function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myStateFunction";
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myStateFunction;
  ```

  For more information on local functions, see "Local Functions".

- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myStateFunction(x,u,params)
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

For more information, see "Specify Prediction Model for Nonlinear MPC".

Example: `"@transFcn"`

OutputFcn — Output function
`[]` (default) | string | character vector | function handle

Output function, specified as a string, character vector, or function handle. If the number of states and outputs of the prediction model are the same, you can omit `OutputFcn`, which implies that all states are measurable; that is, each output corresponds to one state.

Note Your output function cannot have direct feedthrough from any manipulated variable to any output at any time.

You can specify your output function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  Model.OutputFcn = "myOutputFunction";

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  Model.OutputFcn = @myOutputFunction;

  For more information on local functions, see "Local Functions".

- Anonymous function

  Model.OutputFcn = @(x,u,params) myOutputFunction(x,u,params)

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

For more information, see "Specify Prediction Model for Nonlinear MPC".

Example: "@outFcn"

IsContinuousTime — Option to indicate prediction model time domain
true (default) | false

Option to indicate prediction model time domain, specified as one of the following:

- true — Continuous-time prediction model. In this case, the controller automatically discretizes the model during prediction using Ts.
- false — Discrete-time prediction model. In this case, Ts is the sample time of the model.

Note IsContinuousTime must be consistent with the functions specified in Model.StateFcn and Model.OutputFcn.

If IsContinuousTime is true, StateFcn must return the derivative of the state with respect to time, at the current time. Otherwise StateFcn must return the state at the next control interval.

Example: true

NumberOfParameters — Number of optional model parameters
0 (default) | nonnegative integer

Number of optional model parameters used by the prediction model, custom cost function, custom constraints, passivity functions, specified as a nonnegative integer. The number of parameters includes all the parameters used by these functions. For example, if the state function uses only parameter p1, the constraint functions use only parameter p2, and the cost function uses only parameter p3, then NumberOfParameters is 3.

Example: 1

`States` — State information, bounds, and scale factors
structure array

State information, bounds, and scale factors, specified as a structure array with N_x elements, where N_x is the number of states. Each structure element has the following fields.

`Min` — State lower bound
-Inf (default) | scalar | vector

State lower bound, specified as a scalar or vector. By default, this lower bound is -Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

State bounds are always hard constraints.

Example: [-20 -18 -15]

`Max` — State upper bound
Inf (default) | scalar | vector

State upper bound, specified as a scalar or vector. By default, this upper bound is +Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

State bounds are always hard constraints.

Example: [20 15]

`Name` — State name
string | character vector

State name, specified as a string or character vector. The default state name is "x#", where # is its state index.

Example: "speed"

`Units` — State units
"" (default) | string | character vector

State units, specified as a string or character vector.

Example: "m/s"

`ScaleFactor` — State scale factor
1 (default) | positive finite scalar

State scale factor, specified as a positive finite scalar. In general, use the operating range of the state. Specifying the proper scale factor can improve numerical conditioning for optimization.

Example: 10

`OutputVariables` — Output variable information, bounds, and scale factors
structure array

Output variable (OV) information, bounds, and scale factors, specified as a structure array with N_y elements, where N_y is the number of output variables. To access this property, you can use the alias `OV` instead of `OutputVariables`.

Each structure element has the following fields.

`Min` — OV lower bound
-Inf (default) | scalar | vector

OV lower bound, specified as a scalar or vector. By default, this lower bound is `-Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: [-10 -8]

`Max` — OV upper bound
Inf (default) | scalar | vector

OV upper bound, specified as a scalar or vector. By default, this upper bound is `+Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: [12 10 8]

`MinECR` — OV lower bound softness
1 (default) | nonnegative finite scalar | vector

OV lower bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, OV upper bounds are soft constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: [2 1 0.5]

`MaxECR` — OV upper bound softness
1 (default) | nonnegative finite scalar | vector

OV upper bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, OV lower bounds are soft constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: `[5 2 1]`

Name — OV name
string | character vector

OV name, specified as a string or character vector. The default OV name is `"y#"`, where `#` is its output index.

Example: `"attack angle"`

Units — OV units
`""` (default) | string | character vector

OV units, specified as a string or character vector.

Example: `"degrees"`

ScaleFactor — OV scale factor
1 (default) | positive finite scalar

OV scale factor, specified as a positive finite scalar. In general, use the operating range of the output variable. Specifying the proper scale factor can improve numerical conditioning for optimization.

Example: `90`

ManipulatedVariables — Manipulated variable information, bounds, and scale factors
structure array

Manipulated Variable (MV) information, bounds, and scale factors, specified as a structure array with N_{mv} elements, where N_{mv} is the number of manipulated variables. To access this property, you can use the alias MV instead of `ManipulatedVariables`.

Each structure element has the following fields.

Min — MV lower bound
`-Inf` (default) | scalar | vector

MV lower bound, specified as a scalar or vector. By default, this lower bound is `-Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `[-1.1 -1]`

Max — MV upper bound
`Inf` (default) | scalar | vector

MV upper bound, specified as a scalar or vector. By default, this upper bound is `+Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: [1.2 1]

MinECR — MV lower bound softness
0 (default) | nonnegative scalar | vector

MV lower bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative scalar or vector. By default, MV lower bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: [0.1 0]

MaxECR — MV upper bound
0 (default) | nonnegative scalar | vector

MV upper bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative scalar or vector. By default, MV upper bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR value over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR value is used for the remaining steps of the prediction horizon.

Example: [0.5 0.2]

RateMin — MV rate of change lower bound
-Inf (default) | nonpositive scalar | vector

MV rate of change lower bound, specified as a nonpositive scalar or vector. The MV rate of change is defined as $MV(k)$ - $MV(k$–1), where k is the current time. By default, this lower bound is -Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: [-50 -20]

RateMax — MV rate of change upper bound
Inf (default) | nonnegative scalar | vector

MV rate of change upper bound, specified as a nonnegative scalar or vector. The MV rate of change is defined as $MV(k)$ - $MV(k$–1), where k is the current time. By default, this upper bound is +Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

Example: `[50 20]`

RateMinECR — MV rate of change lower bound softness
`0` (default) | nonnegative finite scalar | vector

MV rate of change lower bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, MV rate of change lower bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR values over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR values are used for the remaining steps of the prediction horizon.

Example: `[0.1 0]`

RateMaxECR — MV rate of change upper bound softness
`0` (default) | nonnegative finite scalar | vector

MV rate of change upper bound softness, where a larger ECR value indicates a softer constraint, specified as a nonnegative finite scalar or vector. By default, MV rate of change upper bounds are hard constraints.

To use the same ECR value across the prediction horizon, specify a scalar value.

To vary the ECR values over the prediction horizon from time k to time $k+p$–1, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final ECR values are used for the remaining steps of the prediction horizon.

Example: `[1 0.5 0.2]`

Name — MV name
string | character vector

MV name, specified as a string or character vector. The default MV name is `"u#"`, where # is its input index.

Example: `"rudder angle"`

Units — MV units
`""` (default) | string | character vector

MV units, specified as a string or character vector.

Example: `"degrees"`

ScaleFactor — MV scale factor
`1` (default) | positive finite scalar

MV scale factor, specified as a positive finite scalar. In general, use the operating range of the manipulated variable. Specifying the proper scale factor can improve numerical conditioning for optimization.

Example: 60

`MeasuredDisturbances` — Measured disturbance information and scale factors
structure array

Measured disturbance (MD) information and scale factors, specified as a structure array with N_{md} elements, where N_{md} is the number of measured disturbances. If your model does not have measured disturbances, then `MeasuredDisturbances` is `[]`. To access this property, you can use the alias `MD` instead of `MeasuredDisturbances`.

Each structure element has the following fields.

`Name` — MD name
string | character vector

MD name, specified as a string or character vector. The default MD name is `"u#"`, where `#` is its input index.

Example: `"wind speed"`

`Units` — MD units
`""` (default) | string | character vector

MD units, specified as a string or character vector.

Example: `"m/s"`

`ScaleFactor` — MD scale factor
1 (default) | positive finite scalar

MD scale factor, specified as a positive finite scalar. In general, use the operating range of the disturbance. Specifying the proper scale factor can improve numerical conditioning for optimization.

Example: 10

`Weights` — Standard cost function tuning weights
structure

Standard cost function tuning weights, specified as a structure. The controller applies these weights to the scaled variables. Therefore, the tuning weights are dimensionless values.

Note If you define a custom cost function using `Optimization.CustomCostFcn` and set `Optimization.ReplaceStandardCost` to `true`, then the controller ignores the standard cost function tuning weights in `Weights`.

`Weights` has the following fields.

`ManipulatedVariables` — Manipulated variable tuning weights
row vector | array

Manipulated variable tuning weights, which penalize deviations from MV targets, specified as a row vector or array of nonnegative values. The default weight for all manipulated variables is `0`.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

To specify MV targets at run time, in Simulink, pass the target values to the Nonlinear MPC Controller block. In MATLAB, pass the target values to a simulation function (such as `nlmpcmove`, using the `MVTarget` property of an `nlmpcmoveopt` object).

Example: `[0.1 0.2]`

ManipulatedVariablesRate — Manipulated variable rate tuning weights
row vector | array

Manipulated variable rate tuning weights, which penalize large changes in control moves, specified as a row vector or array of nonnegative values. The default weight for all manipulated variable rates is `0.1`.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable rate tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

Example: `[0.1 0.1]`

OutputVariables — Output variable tuning weights
vector | array

Output variable tuning weights, which penalize deviation from output references, specified as a row vector or array of nonnegative values. The default weight for all output variables is `1`.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

Example: `[0.1 0.1]`

ECR — Slack variable tuning weight
1e5 (default) | positive scalar

Slack variable tuning weight, specified as a positive scalar.

Example: `1e4`

Optimization — Custom optimization functions and solver
structure

Custom optimization functions and solver, specified as a structure with the following fields.

CustomCostFcn — Custom cost function
[ ] | string | character vector | function handle

Custom cost function, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  Optimization.CustomCostFcn = "myCostFunction";

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  Optimization.CustomCostFcn = @myCostFunction;

  For more information on local functions, see "Local Functions".

- Anonymous function

  Optimization.CustomCostFcn = @(X,U,e,data,params) myCostFunction(X,U,e,data,params);

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Your cost function must have the signature:

function J = myCostFunction(X,U,e,data,params)

For more information, see "Specify Cost Function for Nonlinear MPC".

Example: @costFcn

ReplaceStandardCost — Option to replace the standard cost function
true (default) | false

Option to replace the standard cost function with the custom cost function, specified as one of the following:

- true — The controller uses the custom cost alone as the objective function during optimization. In this case, the Weights property of the controller is ignored.
- false — The controller uses the sum of the standard cost and custom cost as the objective function during optimization.

If you do not specify a custom cost function using CustomCostFcn, then the controller ignores RepalceStandardCost.

For more information, see "Specify Cost Function for Nonlinear MPC".

Example: true

CustomEqConFcn — Custom equality constraint function
[ ] (default) | string | character vector | function handle

Custom equality constraint function, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomEqConFcn = "myEqConFunction";
  ```
- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Optimization.CustomEqConFcn = @myEqConFunction;
  ```

  For more information on local functions, see "Local Functions".
- Anonymous function

  ```
  Optimization.CustomEqConFcn = @(X,U,data,params) myEqConFunction(X,U,data,params);
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Your equality constraint function must have the signature:

```
function ceq = myEqConFunction(X,U,data,params)
```

For more information, see "Specify Constraints for Generic Nonlinear MPC".

Example: @eqFcn

CustomIneqConFcn — Custom inequality constraint function
[ ] (default) | string | character vector | function handle

Custom inequality constraint function, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomIneqConFcn = "myIneqConFunction";
  ```
- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Optimization.CustomIneqConFcn = @myIneqConFunction;
  ```

  For more information on local functions, see "Local Functions".
- Anonymous function

  ```
  Optimization.CustomIneqConFcn = @(X,U,e,data,params) myIneqConFunction(X,U,e,data,params);
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Your equality constraint function must have the signature:

```
function cineq = myIneqConFunction(X,U,e,data,params)
```

For more information, see "Specify Constraints for Generic Nonlinear MPC".

Example: @ineqFcn

CustomSolverFcn — Custom nonlinear programming solver
[] (default) | string | character vector | function handle

Custom nonlinear programming solver function, specified as a string, character vector, or function handle. If you do not have Optimization Toolbox software, you must specify your own custom nonlinear programming solver. You can specify your custom solver function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Optimization.CustomSolverFcn = "myNLPSolver";
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Optimization.CustomSolverFcn = @myNLPSolver;
  ```

For more information, see "Configure Optimization Solver for Nonlinear MPC".

Example: @mySolver

SolverOptions — Solver options
options object for fmincon | []

Solver options, specified as an options object for fmincon or [].

If you have Optimization Toolbox software, SolverOptions contains an options object for the fmincon solver.

If you do not have Optimization Toolbox, SolverOptions is [].

For more information, see "Configure Optimization Solver for Nonlinear MPC".

RunAsLinearMPC — Option to simulate as a linear controller
"off" (default) | "Adaptive" | "TimeVarying"

Option to simulate as a linear controller, specified as one of the following:

- "off" — Simulate the controller as a nonlinear controller with a nonlinear prediction model.
- "Adaptive" — For each control interval, a linear model is obtained from the specified nonlinear state and output functions at the current operating point and used across the prediction horizon. To determine if an adaptive MPC controller provides comparable performance to the nonlinear controller, use this option. For more information on adaptive MPC, see "Adaptive MPC".
- "TimeVarying" — For each control interval, p linear models are obtained from the specified nonlinear state and output functions at the p operating points predicted from the previous interval, one for each prediction horizon step. To determine if a linear time-varying MPC controller provides comparable performance to the nonlinear controller, use this option. For more information on time-varying MPC, see "Time-Varying MPC".

To use the either the `"Adaptive"` or `"TimeVarying"` option, your controller must have no custom constraints and no custom cost function.

For an example that simulates a nonlinear MPC controller as a linear controller, see "Optimization and Control of a Fed-Batch Reactor Using Nonlinear MPC".

Example: `"Adaptive"`

UseSuboptimalSolution — Option to accept a suboptimal solution
`false` (default) | `true`

Option to accept a suboptimal solution, specified as a logical value. When the nonlinear programming solver reaches the maximum number of iterations without finding a solution (the exit flag is `0`), the controller:

- Freezes the MV values if `UseSuboptimalSolution` is `false`
- Applies the suboptimal solution found by the solver after the final iteration if `UseSuboptimalSolution` is `true`

To specify the maximum number of iterations, use `Optimization.SolverOptions.MaxIter`.

Example: `true`

MVInterpolationOrder — Linear interpolation order used for block moves
`0` (default) | `1`

Linear interpolation order used by block moves, specified as one of the following:

- `0` — Use piecewise constant manipulated variable intervals.
- `1` — Use piecewise linear manipulated variable intervals.

If the control horizon is a scalar, then the controller ignores `MVInterpolationOrder`.

For more information on manipulated variable blocking, see "Manipulated Variable Blocking".

Example: `1`

Jacobian — Jacobians of model functions, and custom cost and constraint functions
structure

Jacobians of model functions, and custom cost and constraint functions, specified as a structure. As a best practice, use Jacobians whenever they are available, since they improve optimization efficiency. If you do not specify a Jacobian for a given function, the nonlinear programming solver must numerically compute the Jacobian.

The `Jacobian` structure contains the following fields.

StateFcn — Jacobian of state function
`[]` (default) | string | character vector | function handle

Jacobian of state function z from `Model.StateFcn`, specified as one of the following

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myStateJacobian";
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myStateJacobian;
  ```

  For more information on local functions, see "Local Functions".

- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myStateJacobian(x,u,params)
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

For more information, see "Specify Prediction Model for Nonlinear MPC".

Example: @Afcn

`OutputFcn` — Jacobian of output function
[] (default) | string | character vector | function handle

Jacobian of output function y from `Model.OutputFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateFcn = "myOutputJacobian";
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Model.StateFcn = @myOutputJacobian;
  ```

  For more information on local functions, see "Local Functions".

- Anonymous function

  ```
  Model.StateFcn = @(x,u,params) myOutputJacobian(x,u,params)
  ```

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

For more information, see "Specify Prediction Model for Nonlinear MPC".

Example: @Cfcn

`CustomCostFcn` — Jacobian of custom cost function
[] | string | character vector | function handle

Jacobian of custom cost function J from `Optimization.CustomCostFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Jacobian.CustomCostFcn = "myCostJacobian";`

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  `Jacobian.CustomCostFcn = @myCostJacobian;`

  For more information on local functions, see "Local Functions".

- Anonymous function

  `Jacobian.CustomCostFcn = @(X,U,e,data,params) myCostJacobian(X,U,e,data,params)`

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Your cost Jacobian function must have the signature:

`function [G,Gmv,Ge] = myCostJacobian(X,U,e,data,params)`

For more information, see "Specify Cost Function for Nonlinear MPC".

Example: `@costJacFcn`

CustomEqConFcn — Jacobian of custom equality constraints
[ ] (default) | string | character vector | function handle

Jacobian of custom equality constraints `ceq` from `Optimization.CustomEqConFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Jacobian.CustomEqConFcn = "myEqConJacobian";`

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  `Jacobian.CustomEqConFcn = @myEqConJacobian;`

  For more information on local functions, see "Local Functions".

- Anonymous function

  `Jacobian.CustomEqConFcn = @(X,U,data,params) myEqConJacobian(X,U,data,params);`

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Your equality constraint Jacobian function must have the signature:

```
function [G,Gmv] = myEqConJacobian(X,U,data,params)
```

For more information, see "Specify Constraints for Generic Nonlinear MPC".

Example: `@eqJacFcn`

`CustomIneqConFcn` — Jacobian of custom inequality constraints
`[ ]` (default) | string | character vector | function handle

Jacobian of custom inequality constraints `c` from `Optimization.CustomIneqConFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Jacobian.CustomEqConFcn = "myIneqConJacobian";
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Jacobian.CustomEqConFcn = @myIneqConJacobian;
  ```

  For more information on local functions, see "Local Functions".

- Anonymous function

  ```
  Jacobian.CustomEqConFcn = @(X,U,data,params) myIneqConJacobian(X,U,data,params);
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Your inequality constraint Jacobian function must have the signature:

```
function [G,Gmv,Ge] = myIneqConJacobian(X,U,data,params)
```

For more information, see "Specify Constraints for Generic Nonlinear MPC".

Example: `@ineqJacFcn`

`Passivity` — Passivity constraints
structure

Passivity constraints, specified as a structure with the following fields.

When your nonlinear MPC controller is configured to use passivity constraints, at each step the optimization algorithm tries to enforce the inequality constraints:

$$y_p(x, u)^T u_p(x, u) + \nu_y\, y_p(x, u)^T y_p(x, u) + \nu_u\, u_p(x, u)^T u_p(x, u) \leq 0.$$

Here, ν_y is the output passivity index, ν_u is the input passivity index, $u_p(x,u)$ is the passivity input function, and $y_p(x,u)$ is the passivity output function. The variables x and u are the current state and input of the prediction model.

Assuming that the plant is already passive with respect to the input-output pair u_p and y_p, if these two inequalities are verified, then (under mild conditions) the resulting closed loop system tends to dissipate energy over time, and therefore has a stable equilibrium. For more information on passivity see "Specify Constraints for Generic Nonlinear MPC" and, in the context of linear systems, "About Passivity and Passivity Indices". For examples, see "Control Quadruple-Tank Using Passivity-Based Nonlinear MPC" and "Control Robot Manipulator Using Passivity-Based Nonlinear MPC".

EnforceConstraint — Option to enforce constraints
`false` (default) | `true`

Option to enforce constraints, specified as one of the following:

- `true` — Passivity constraints are enforced during optimization. In this case, you must specify the `OutputFcn` and `InputFcn` properties.
- `false` — Passivity constraints are not enforced during optimization.

Example: `true`

OutputPassivityIndex — Desired output passivity index for controller
`0.1` (default) | nonnegative scalar

Desired output passivity index for the controller, specified as a nonnegative scalar.

If `Passivity.EnforceConstraint` is `true`, at each step the optimization algorithm tries to enforce the passivity inequality constraint, which involves the passivity index ν_y specified in `Passivity.OututPassivityIndex`.

Example: 1

InputPassivityIndex — Desired input passivity index for controller
`0` (default) | nonnegative scalar

Desired output passivity index for the controller, specified as a nonnegative scalar.

If `Passivity.EnforceConstraint` is `true`, at each step the optimization algorithm tries to enforce the passivity inequality constraint, which involves the passivity index ν_u specified in `Passivity.InputPassivityIndex`.

Example: 1

OutputFcn — Passivity output function
`[]` (default) | string | character vector | function handle

Passivity output function, specified as a string, character vector, or function handle.

If `Passivity.EnforceConstraint` is `true` then at each step the optimization algorithm tries to enforce the input and output inequality constraints, which involve the function $y_p(x,u)$ specified in `Passivity.OutputFcn`.

You can specify your passivity output function as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Passivity.OutputFcn = "myPassivityOutputFcn";`

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Passivity.OutputFcn = @myPassivityOutputFcn;
  ```

  For more information on local functions, see "Local Functions".

- Anonymous function

  ```
  Passivity.OutputFcn = @(x,u,params) myPassivityOutputFcn(x,u,params)
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Here, x and u are the prediction model states and inputs, respectively, and `params` is an optional comma separated list of parameters (for example `p1,p2,p3`) that might be needed by the function you specify. If any of your functions use optional parameters, you must specify the number of parameters using `Model.NumberOfParameters`. At run time, in Simulink, you then pass these parameters to the Nonlinear MPC Controller block. In MATLAB, you pass the parameters to a simulation function (such as `nlmpcmove`, using an `nlmpcmoveopt` option set object).

Example: `@ypFcn`

InputFcn — Passivity input function
string | character vector | function handle

Passivity output function, specified as a string, character vector, or function handle. If `Passivity.EnforceConstraint` is `true` then at each step the optimization algorithm tries to enforce the input and output inequality constraints, which involve the function $u_p(x,u)$ specified in `Passivity.OutputFcn`.

You can specify your passivity input function as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Passivity.InputFcn = "myPassivityInputFcn";
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Passivity.InputFcn = @myPassivityInputFcn;
  ```

  For more information on local functions, see "Local Functions".

- Anonymous function

  ```
  Passivity.InputFcn = @(x,u,params) myPassivityInputFcn(x,u,params)
  ```

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Here, `x` and `u` are the prediction model states and inputs, respectively, and `params` is an optional comma separated list of parameters (for example `p1,p2,p3`) that might be needed by the function you specify. If any of your functions use optional parameters, you must specify the number of parameters using `Model.NumberOfParameters`. At run time, in Simulink, you then pass these parameters to the Nonlinear MPC Controller block. In MATLAB, you pass the parameters to a simulation function (such as `nlmpcmove`, using an `nlmpcmoveopt` option set object).

Example: `@upFcn`

`OutputJacobianFcn` — Jacobian of passivity output function
`[]` (default) | string | character vector | function handle

Jacobian of the passivity output function `Passivity.OutputFcn`, specified as one of the following:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Passivity.OutputJacobianFcn = "myPsvOutJacFcn";`

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  `Passivity.OutputJacobianFcn = @myPsvOutJacFcn;`

  For more information on local functions, see "Local Functions".

- Anonymous function

  `Passivity.OutputJacobianFcn = @(x,u,params) myPsvOutJacFcn(x,u,params)`

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Here, `x` and `u` are the prediction model states and outputs, respectively, and `params` is an optional comma separated list (for example `p1,p2,p3`) of parameters that might be needed by the function you specify. If any of your function use optional parameters, you must specify the number of parameters using `Model.NumberOfParameters`. At run time, in Simulink, you then pass these parameters to the Nonlinear MPC Controller block. In MATLAB, you pass the parameters to a simulation function (such as `nlmpcmove`, using an `nlmpcmoveopt`).

The function specified in `Passivity.OutputJacobianFcn` (if any) must return as a first output argument the Jacobian matrix of the output passivity function with respect to the current state (an N_{y_p} by N_x matrix), and as a second output argument the Jacobian matrix of the output passivity function with respect to the manipulated variables (an N_{y_p} by N_{mv} matrix).

Here, N_x is the number of state variables of the prediction model, N_{mv} is the number of manipulated variables and N_{y_p} is the number of outputs of the passivity output function.

Example: `@ypJac`

InputJacobianFcn — Jacobian of passivity input function
`[]` (default) | string | character vector | function handle

Jacobian of the passivity input function `Passivity.InputFcn`, specified as one of the following

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Passivity.InputJacobianFcn = "myPsvInJacFcn";`

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  `Passivity.InputJacobianFcn = @myPsvInJacFcn;`

  For more information on local functions, see "Local Functions".

- Anonymous function

  `Passivity.InputJacobianFcn = @(x,u,params) myPsvInJacFcn(x,u,params)`

  For more information on anonymous functions, see "Anonymous Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, output, cost, or constraint functions (or their Jacobians) as local or anonymous functions is not recommended.

Here, `x` and `u` are the prediction model states and outputs, respectively, and `params` is an optional comma separated list (for example `p1,p2,p3`) of parameters that might be needed by the function you specify. If any of your function use optional parameters, you must specify the number of parameters using `Model.NumberOfParameters`. At run time, in Simulink, you then pass these parameters to the Nonlinear MPC Controller block. In MATLAB, you pass the parameters to a simulation function (such as `nlmpcmove`, using an `nlmpcmoveopt`).

The function specified in `Passivity.InputJacobianFcn` (if any) must return as a first output argument the Jacobian of the input passivity function with respect to the current state (an N_{u_p} by N_x matrix), and as a second output argument the Jacobian of the input passivity function with respect to the manipulated variables (an N_{u_p} by N_{mv} matrix).

Here, N_x is the number of state variables of the prediction model, N_{mv} is the number of manipulated variables and N_{u_p} is the number of outputs of the passivity input function.

Example: `@upFcn`

UsePredictedState — Option to use predicted or current state
`true` (default) | `false`

Option to use predicted or current state, specified as one of the following:

- `true` — $x[k+1]$ is a decision variable in the optimization problem.
- `false` — $x[k]$ is a decision variable in the optimization problem.

Example: `true`

Object Functions

nlmpcmove	Compute optimal control action for nonlinear MPC controller
validateFcns	Examine prediction model and custom functions of nlmpc or nlmpcMultistage objects for potential problems
convertToMPC	Convert nlmpc object into one or more mpc objects
createParameterBus	Create Simulink bus object and configure Bus Creator block for passing model parameters to Nonlinear MPC Controller block

Examples

Create Nonlinear MPC Controller with Discrete-Time Prediction Model

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nx = 4;
ny = 2;
nu = 1;
nlobj = nlmpc(nx,ny,nu);
```

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The discrete-time state function uses an optional parameter, the sample time `Ts`, to integrate the continuous-time model. Therefore, you must specify the number of optional parameters as `1`.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function for the controller. In this case, define the first and third states as outputs. Even though this output function does not use the optional sample time parameter, you must specify the parameter as an input argument (`Ts`).

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Validate the prediction model functions for nominal states `x0` and nominal inputs `u0`. Since the prediction model uses a custom parameter, you must pass this parameter to `validateFcns`.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Nonlinear MPC Controller with Measured and Unmeasured Disturbances

Create a nonlinear MPC controller with three states, one output, and four inputs. The first two inputs are measured disturbances, the third input is the manipulated variable, and the fourth input is an unmeasured disturbance.

```
nlobj = nlmpc(3,1,'MV',3,'MD',[1 2],'UD',4);
```

To view the controller state, output, and input dimensions and indices, use the `Dimensions` property of the controller.

```
nlobj.Dimensions
```

```
ans = struct with fields:
     NumberOfStates: 3
    NumberOfOutputs: 1
     NumberOfInputs: 4
            MVIndex: 3
            MDIndex: [1 2]
            UDIndex: 4
```

Specify the controller sample time and horizons.

```
nlobj.Ts = 0.5;
nlobj.PredictionHorizon = 6;
nlobj.ControlHorizon = 3;
```

Specify the prediction model state function, which is in the file `exocstrStateFcnCT.m`.

```
nlobj.Model.StateFcn = 'exocstrStateFcnCT';
```

Specify the prediction model output function, which is in the file `exocstrOutputFcn.m`.

```
nlobj.Model.OutputFcn = 'exocstrOutputFcn';
```

Validate the prediction model functions using the initial operating point as the nominal condition for testing and setting the unmeasured disturbance state, `x0(3)`, to `0`. Since the model has measured disturbances, you must pass them to `validateFcns`.

```
x0 = [311.2639; 8.5698; 0];
u0 = [10; 298.15; 298.15];
validateFcns(nlobj,x0,u0(3),u0(1:2)');
```

```
Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Validate Nonlinear MPC Prediction Model and Custom Functions

Create nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
```

```
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

```
Zero weights are applied to one or more OVs because there are fewer MVs than OVs.
```

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Validate the prediction model and custom functions at the initial states (x0) and initial inputs (u0) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Create Linear MPC Controllers from Nonlinear MPC Controller

Create a nonlinear MPC controller with four states, one output variable, one manipulated variable, and one measured disturbance.

```
nlobj = nlmpc(4,1,'MV',1,'MD',2);
```

Specify the controller sample time and horizons.

```
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 3;
```

Specify the state function of the prediction model.

```
nlobj.Model.StateFcn = 'oxidationStateFcn';
```

Specify the prediction model output function and the output variable scale factor.

```
nlobj.Model.OutputFcn = @(x,u) x(3);
nlobj.OutputVariables.ScaleFactor = 0.03;
```

Specify the manipulated variable constraints and scale factor.

```
nlobj.ManipulatedVariables.Min = 0.0704;
nlobj.ManipulatedVariables.Max = 0.7042;
nlobj.ManipulatedVariables.ScaleFactor = 0.6;
```

Specify the measured disturbance scale factor.

```
nlobj.MeasuredDisturbances.ScaleFactor = 0.5;
```

Compute the state and input operating conditions for three linear MPC controllers using the `fsolve` function.

```
options = optimoptions('fsolve','Display','none');

uLow = [0.38 0.5];
xLow = fsolve(@(x) oxidationStateFcn(x,uLow),[1 0.3 0.03 1],options);

uMedium = [0.24 0.5];
xMedium = fsolve(@(x) oxidationStateFcn(x,uMedium),[1 0.3 0.03 1],options);

uHigh = [0.15 0.5];
xHigh = fsolve(@(x) oxidationStateFcn(x,uHigh),[1 0.3 0.03 1],options);
```

Create linear MPC controllers for each of these nominal conditions.

```
mpcobjLow = convertToMPC(nlobj,xLow,uLow);
mpcobjMedium = convertToMPC(nlobj,xMedium,uMedium);
mpcobjHigh = convertToMPC(nlobj,xHigh,uHigh);
```

You can also create multiple controllers using arrays of nominal conditions. The number of rows in the arrays specifies the number controllers to create. The linear controllers are returned as cell array of `mpc` objects.

```
u = [uLow; uMedium; uHigh];
x = [xLow; xMedium; xHigh];
mpcobjs = convertToMPC(nlobj,x,u);
```

View the properties of the `mpcobjLow` controller.

```
mpcobjLow
```

```
MPC object (created on 03-Mar-2023 21:07:32):
---------------------------------------------
Sampling time:      1 (seconds)
Prediction Horizon: 10
Control Horizon:    3

Plant Model:
                              -------------
     1  manipulated variable(s)   -->|  4 states  |
```

```
                                       |              |--> 1 measured output(s)
        1  measured disturbance(s)  -->|  2 inputs    |
                                       |              |--> 0 unmeasured output(s)
        0  unmeasured disturbance(s) -->|  1 outputs  |
                                        --------------
Indices:
  (input vector)     Manipulated variables: [1 ]
                    Measured disturbances: [2 ]
  (output vector)        Measured outputs: [1 ]

Disturbance and Noise Models:
       Output disturbance model: default (type "getoutdist(mpcobjLow)" for details)
        Measurement noise model: default (unity gain after scaling)

Weights:
       ManipulatedVariables: 0
   ManipulatedVariablesRate: 0.1000
           OutputVariables: 1
                       ECR: 100000

State Estimation:  Default Kalman Filter (type "getEstimator(mpcobjLow)" for details)

Constraints:
 0.0704 <= u1 <= 0.7042, u1/rate is unconstrained, y1 is unconstrained

Use built-in "active-set" QP solver with MaxIterations of 120.
```

Plan Optimal Trajectory Using Nonlinear MPC

Create a nonlinear MPC controller with six states, six outputs, and four inputs.

```
nx = 6;
ny = 6;
nu = 4;
nlobj = nlmpc(nx,ny,nu);
```

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the controller sample time and horizons.

```
Ts = 0.4;
p = 30;
c = 4;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = p;
nlobj.ControlHorizon = c;
```

Specify the prediction model state function and the Jacobian of the state function. For this example, use a model of a flying robot.

```
nlobj.Model.StateFcn = "FlyingRobotStateFcn";
nlobj.Jacobian.StateFcn = "FlyingRobotStateJacobianFcn";
```

Specify a custom cost function for the controller that replaces the standard cost function.

```
nlobj.Optimization.CustomCostFcn = @(X,U,e,data) Ts*sum(sum(U(1:p,:)));
nlobj.Optimization.ReplaceStandardCost = true;
```

Specify a custom constraint function for the controller.

```
nlobj.Optimization.CustomEqConFcn = @(X,U,data) X(end,:)';
```

Specify linear constraints on the manipulated variables.

```
for ct = 1:nu
    nlobj.MV(ct).Min = 0;
    nlobj.MV(ct).Max = 1;
end
```

Validate the prediction model and custom functions at the initial states (x0) and initial inputs (u0) of the robot.

```
x0 = [-10;-10;pi/2;0;0;0];
u0 = zeros(nu,1);
validateFcns(nlobj,x0,u0);
```

```
Model.StateFcn is OK.
Jacobian.StateFcn is OK.
No output function specified. Assuming "y = x" in the prediction model.
Optimization.CustomCostFcn is OK.
Optimization.CustomEqConFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Compute the optimal state and manipulated variable trajectories, which are returned in the info.

```
[~,~,info] = nlmpcmove(nlobj,x0,u0);
```

```
Slack variable unused or zero-weighted in your custom cost function.
All constraints will be hard.
```

Plot the optimal trajectories.

```
FlyingRobotPlotPlanning(info,Ts)
```

```
Optimal fuel consumption =   1.884953
```

Optimal Trajectory

Simulate Closed-Loop Control using Nonlinear MPC Controller

Create a nonlinear MPC controller with four states, two outputs, and one input.

```
nlobj = nlmpc(4,2,1);
```

Zero weights are applied to one or more OVs because there are fewer MVs than OVs.

Specify the sample time and horizons of the controller.

```
Ts = 0.1;
nlobj.Ts = Ts;
nlobj.PredictionHorizon = 10;
nlobj.ControlHorizon = 5;
```

Specify the state function for the controller, which is in the file `pendulumDT0.m`. This discrete-time model integrates the continuous time model defined in `pendulumCT0.m` using a multistep forward Euler method.

```
nlobj.Model.StateFcn = "pendulumDT0";
nlobj.Model.IsContinuousTime = false;
```

The prediction model uses an optional parameter, `Ts`, to represent the sample time. Specify the number of parameters.

```
nlobj.Model.NumberOfParameters = 1;
```

Specify the output function of the model, passing the sample time parameter as an input argument.

```
nlobj.Model.OutputFcn = @(x,u,Ts) [x(1); x(3)];
```

Define standard constraints for the controller.

```
nlobj.Weights.OutputVariables = [3 3];
nlobj.Weights.ManipulatedVariablesRate = 0.1;
nlobj.OV(1).Min = -10;
nlobj.OV(1).Max = 10;
nlobj.MV.Min = -100;
nlobj.MV.Max = 100;
```

Validate the prediction model functions.

```
x0 = [0.1;0.2;-pi/2;0.3];
u0 = 0.4;
validateFcns(nlobj, x0, u0, [], {Ts});

Model.StateFcn is OK.
Model.OutputFcn is OK.
Analysis of user-provided model, cost, and constraint functions complete.
```

Only two of the plant states are measurable. Therefore, create an extended Kalman filter for estimating the four plant states. Its state transition function is defined in `pendulumStateFcn.m` and its measurement function is defined in `pendulumMeasurementFcn.m`.

```
EKF = extendedKalmanFilter(@pendulumStateFcn,@pendulumMeasurementFcn);
```

Define initial conditions for the simulation, initialize the extended Kalman filter state, and specify a zero initial manipulated variable value.

```
x = [0;0;-pi;0];
y = [x(1);x(3)];
EKF.State = x;
mv = 0;
```

Specify the output reference value.

```
yref = [0 0];
```

Create an `nlmpcmoveopt` object, and specify the sample time parameter.

```
nloptions = nlmpcmoveopt;
nloptions.Parameters = {Ts};
```

Run the simulation for 10 seconds. During each control interval:

1   Correct the previous prediction using the current measurement.
2   Compute optimal control moves using `nlmpcmove`. This function returns the computed optimal sequences in `nloptions`. Passing the updated options object to `nlmpcmove` in the next control interval provides initial guesses for the optimal sequences.
3   Predict the model states.
4   Apply the first computed optimal control move to the plant, updating the plant states.

5   Generate sensor data with white noise.

6   Save the plant states.

```
Duration = 10;
xHistory = x;
for ct = 1:(Duration/Ts)
    % Correct previous prediction
    xk = correct(EKF,y);
    % Compute optimal control moves
    [mv,nloptions] = nlmpcmove(nlobj,xk,mv,yref,[],nloptions);
    % Predict prediction model states for the next iteration
    predict(EKF,[mv; Ts]);
    % Implement first optimal control move
    x = pendulumDT0(x,mv,Ts);
    % Generate sensor data
    y = x([1 3]) + randn(2,1)*0.01;
    % Save plant states
    xHistory = [xHistory x];
end
```

Plot the resulting state trajectories.

```
figure
subplot(2,2,1)
plot(0:Ts:Duration,xHistory(1,:))
xlabel('time')
ylabel('z')
title('cart position')
subplot(2,2,2)
plot(0:Ts:Duration,xHistory(2,:))
xlabel('time')
ylabel('zdot')
title('cart velocity')
subplot(2,2,3)
plot(0:Ts:Duration,xHistory(3,:))
xlabel('time')
ylabel('theta')
title('pendulum angle')
subplot(2,2,4)
plot(0:Ts:Duration,xHistory(4,:))
xlabel('time')
ylabel('thetadot')
title('pendulum velocity')
```

Version History

Introduced in R2018b

See Also

Functions
validateFcns | nlmpcmove | nlmpcmoveCodeGeneration | getCodeGenerationData

Objects
nlmpcmoveopt | nlmpcMultistage

Blocks
Nonlinear MPC Controller

Topics
"Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC"
"Nonlinear Model Predictive Control of an Exothermic Chemical Reactor"
"Swing-Up Control of a Pendulum Using Nonlinear Model Predictive Control"
"Nonlinear and Gain-Scheduled MPC Control of an Ethylene Oxidation Plant"
"Control Quadruple-Tank Using Passivity-Based Nonlinear MPC"
"Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" (Robotics System Toolbox)

"Nonlinear MPC"
"What is Model Predictive Control?"

nlmpcMultistage

Multistage nonlinear model predictive controller

Description

A multistage nonlinear model predictive controller computes optimal control moves across the prediction horizon *p* using a nonlinear prediction model. Stages include the current time *k* and all prediction steps (from *k*+1 to *k*+*p*). You can specify different cost and constraint functions for each stage. These functions rely only on plant information such as states and inputs available at that stage. For more information on nonlinear MPC, see "Nonlinear MPC".

Creation

Syntax

```
msobj = nlmpcMultistage(p,nx,nu)
```

```
msobj = nlmpcMultistage(p,nx,'MV',mvIndex,'MD',mdIndex)
msobj = nlmpcMultistage(p,nx,'MV',mvIndex,'UD',udIndex)
msobj = nlmpcMultistage(p,nx,'MV',mvIndex,'MD',mdIndex,'UD',udIndex)
```

Description

`msobj = nlmpcMultistage(p,nx,nu)` creates an `nlmpcMultistage` object with a prediction horizon `p`, whose prediction model has `nx` states and `nu` inputs, and where all inputs are manipulated variables. Use this syntax if your model has no measured or unmeasured disturbance inputs.

`msobj = nlmpcMultistage(p,nx,'MV',mvIndex,'MD',mdIndex)` creates an `nlmpcMultistage` object whose prediction model has measured disturbance inputs. Specify the input indices for the manipulated variables, `mvIndex`, and measured disturbances, `mdIndex`.

`msobj = nlmpcMultistage(p,nx,'MV',mvIndex,'UD',udIndex)` creates an `nlmpcMultistage` object whose prediction model has unmeasured disturbance inputs. Specify the input indices for the manipulated variables, `mvIndex`, and unmeasured disturbances, `udIndex`.

`msobj = nlmpcMultistage(p,nx,'MV',mvIndex,'MD',mdIndex,'UD',udIndex)` creates an `nlmpcMultistage` object whose prediction model has both measured and unmeasured disturbance inputs. Specify the input indices for the manipulated variables, measured disturbances, and unmeasured disturbances.

Input Arguments

p — Prediction horizon
positive integer

Prediction horizon number of steps, specified as a positive integer. This syntax sets the read-only property `PredictionHorizon` equal to the input argument `p`. Since this property is read-only, you

cannot change it after creating the controller object. Note that p also determines the number of stages (p+1).

nx — Number of prediction model states
positive integer

Number of prediction model states, specified as a positive integer. This value is stored in the `Dimensions.NumberOfStates` controller read-only property. You cannot change the number of states after creating the controller object.

nu — Number of prediction model inputs
positive integer

Number of prediction model inputs, which are all set to be manipulated variables, specified as a positive integer. This value is stored in the `Dimensions.NumberOfInputs` controller read-only property. You cannot change the number of manipulated variables after creating the controller object.

mvIndex — Manipulated variable indices
vector of positive integers

Manipulated variable indices, specified as a vector of positive integers. You cannot change these indices after creating the controller object. This value is stored in the `Dimensions.MVIndex` controller property.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from 1 through N_u, where N_u is the number of prediction model inputs.

mdIndex — Measured disturbance indices
vector of positive integers

Measured disturbance indices, specified as a vector of positive integers. You cannot change these indices after creating the controller object. This value is stored in the `Dimensions.MDIndex` controller property.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from 1 through N_u, where N_u is the number of prediction model inputs.

udIndex — Unmeasured disturbance indices
vector of positive integers

Unmeasured disturbance indices, specified as a vector of positive integers. You cannot change these indices after creating the controller object. This value is stored in the `Dimensions.UDIndex` controller property.

The combined set of indices from `mvIndex`, `mdIndex`, and `udIndex` must contain all integers from 1 through N_u, where N_u is the number of prediction model inputs.

Properties

Ts — Prediction model sample time
1 (default) | positive finite scalar

Prediction model sample time, specified as a positive finite scalar. The controller uses a discrete-time model with a sample time of `Ts` for prediction. If you specify a continuous-time prediction model

(`Model.IsContinuousTime` is `true`), then the controller discretizes the model using the built-in implicit trapezoidal rule with a sample time of `Ts`.

PredictionHorizon — Prediction horizon
positive integer

This property is read-only.

Prediction horizon steps, specified as a read-only positive integer. The product of `PredictionHorizon` and `Ts` is the prediction time, that is, how far the controller looks into the future.

UseMVRate — MV rate used in MPC problem
`false` (default) | `true`

Flag indicating whether the rate of change of the manipulated variables is used as a decision variable in the problem formulation, specified as a logical value. Set `UseMVRate` to `true` if:

- You need to specify hard upper or lower bounds on the MV rate.
- The MV rate appears as a term in a cost or constraint function at any stage.
- You need to implement block moves (which you can do so by setting the `RateMin` and `RateMax` bounds at the corresponding stages to zero).

By default, the value is `false`, which means that the rate of change of the manipulated variables does not explicitly appear in the formulation of your MPC problem.

Dimensions — Prediction model dimensional information
structure

This property is read-only.

Prediction model dimensional information, specified when you create the controller and stored as a structure with the following fields.

NumberOfStates — Number of states
positive integer

This property is read-only.

Number of states in the prediction model, specified as a positive integer. This value corresponds to `nx`.

NumberOfInputs — Number of inputs
positive integer

This property is read-only.

Number of inputs in the prediction model, specified as a positive integer. This value corresponds to either `nu` or the sum of the lengths of `mvIndex`, `mdIndex`, and `udIndex`.

MVIndex — Manipulated variable indices
vector of positive integers

This property is read-only.

Manipulated variable indices for the prediction model, specified as a vector of positive integers. This value corresponds to `mvIndex`.

MDIndex — Measured disturbance indices
vector of positive integers

This property is read-only.

Measured disturbance indices for the prediction model, specified as a vector of positive integers. This value corresponds to `mdIndex`.

UDIndex — Unmeasured disturbance indices
vector of positive integers

This property is read-only.

Unmeasured disturbance indices for the prediction model, specified as a vector of positive integers. This value corresponds to `udIndex`.

Model — Prediction model
structure

Prediction model, specified as a structure with the following fields.

StateFcn — State function
string | character vector | function handle

State function, specified as a string, character vector, or function handle. For a continuous-time prediction model, `StateFcn` is the state derivative function. For a discrete-time prediction model, `StateFcn` is the state update function.

If your state function is continuous-time, the controller automatically discretizes the model using the implicit trapezoidal rule. This method can handle moderately stiff models, and its prediction accuracy depends on the controller sample time `Ts`; that is, a large sample time leads to inaccurate prediction.

If the default discretization method does not provide satisfactory prediction for your application, you can specify your own discrete-time prediction model that uses a different method, such as the multistep forward Euler rule.

You can specify your state function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector:

  ```
  Model.StateFcn = "myStateFunction";
  ```
- Handle to a local function, or a function defined in the current working folder or on the MATLAB path:

  ```
  Model.StateFcn = @myStateFunction;
  ```

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

The state function must have the following input and outputs.

```
if Model.ParameterLength>0
     out = myStateFunction(x,u,pm);
else
     out = myStateFunction(x,u);
end
```

Here, x is the state vector, u is the input vector, and pm is the model parameter vector. If IsContinuousTime is true then out must be the value of the state derivative with respect to time, otherwise it must be the value of the state in the following time interval.

For more information, see "Specify Prediction Model for Nonlinear MPC".

StateJacFcn — State Jacobian function
[ ] (default) | string | character vector | function handle

State Jacobian function, specified as a string, character vector, or function handle. As a best practice, use Jacobians whenever they are available, since they improve optimization efficiency. If you do not specify a Jacobian for a given function, the nonlinear programming solver must compute the Jacobian numerically for each optimization step.

Note You can automatically generate a Jacobian function using generateJacobianFunction.

You can specify your state Jacobian function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Model.StateJacFcn = "myStateJacFunction";
  ```
- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Model.StateJacFcn = @myStateJacFunction;
  ```

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

The state Jacobian function must have the following input and outputs.

```
if Model.ParameterLength>0
     [A,Bmv] = myStateJacFunction(x,u,pm);
else
     [A,Bmv] = myStateJacFunction(x,u);
end
```

Here, x is the state vector, u is the input vector, and pm is the model parameter vector. A is the Jacobian of the state function (either continuous or discrete time) with respect to the state vector and B is the Jacobian of the state function with respect to the manipulated variable vector. A is a square

matrix with N_x rows and columns, where N_x is the number of states (`Dimensions.NumberOfStates`). Bmv must have N_x rows and N_{mv} columns, where N_{mv} is the number of manipulated variables.

For more information, see "Specify Prediction Model for Nonlinear MPC".

`IsContinuousTime` — Flag indicating prediction model time domain
`true` (default) | `false`

Flag indicating the prediction model time domain, specified as one of the following:

- `true` — Continuous-time prediction model. In this case, the controller automatically discretizes the model during prediction using `Ts`.
- `false` — Discrete-time prediction model. In this case, `Ts` is the sample time of the model.

Note If `IsContinuousTime` is `true`, `StateFcn` must return the derivative of the state with respect to time, at the current time. Otherwise `StateFcn` must return the state at the next control interval.

`ParameterLength` — Length of the parameter vector
`0` (default) | nonnegative integer

Length of the parameter vector used by the prediction model, specified as a nonnegative integer. If the model state function or its Jacobian require external parameters, set this value to the number of scalar parameters needed. At runtime you must then provide a numeric parameter vector, across the whole prediction horizon, to the controller.

`TerminalState` — Terminal state
`[]` (default) | vector

Terminal state, specified as a column vector with as many elements as the number of states. The terminal state is the desired state at the last prediction step. If any states in the vector do not have terminal values, specify `inf` at the corresponding locations to leave their terminal values free.

The default value of this property is `[]`, meaning that no terminal state constraint is enforced.

`States` — State information and bounds
structure array

State information and hard bounds, specified as a structure array with N_x elements, where N_x is the number of states. Each structure element has the following fields.

`Min` — State hard lower bound
`-Inf` (default) | scalar | vector

State hard lower bound, specified as a scalar or vector. By default, this lower bound is `-Inf`.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

State bounds are always hard constraints. Use stage inequality constraints to implement soft bounds (see Stages).

Max — State hard upper bound
Inf (default) | scalar | vector

State hard upper bound, specified as a scalar or vector. By default, this upper bound is Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time $k+1$ to time $k+p$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

State bounds are always hard constraints. Use stage inequality constraints to implement soft bounds (see Stages).

Name — State name
string | character vector

State name, specified as a string or character vector. The default state name is "x#", where # is its state index.

Units — State units
"" (default) | string | character vector

State units, specified as a string or character vector.

ManipulatedVariables — Manipulated variable information and hard bounds
structure array

Manipulated Variable (MV) information and hard bounds, specified as a structure array with N_{mv} elements, where N_{mv} is the number of manipulated variables. To access this property, you can use the alias MV instead of ManipulatedVariables.

Each structure element has the following fields.

Min — MV hard lower bound
-Inf (default) | scalar | vector

MV hard lower bound, specified as a scalar or vector. By default, this lower bound is -Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time k to time $k+p–1$, specify a vector of up to p values. Here, k is the current time and p is the prediction horizon. If you specify fewer than p values, the final bound is used for the remaining steps of the prediction horizon.

MV bounds are always hard constraints. Use stage inequality constraints to implement soft bounds (see Stages).

Max — MV hard upper bound
Inf (default) | scalar | vector

MV hard upper bound, specified as a scalar or vector. By default, this upper bound is +Inf.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time *k* to time *k*+*p*–1, specify a vector of up to *p* values. Here, *k* is the current time and *p* is the prediction horizon. If you specify fewer than *p* values, the final bound is used for the remaining steps of the prediction horizon.

MV bounds are always hard constraints. Use stage inequality constraints to implement soft bounds (see Stages).

RateMin — MV rate of change hard lower bound
`-Inf` (default) | nonpositive scalar | vector

MV rate of change hard lower bound, specified as a nonpositive scalar or vector. The MV rate of change at stage *i* is defined as *MV*(*i*) - *MV*(*i*–1). By default, this lower bound is `-Inf`. If UseMVRate is `false` this value is ignored.

To use the same bound across the prediction horizon, specify a scalar value.

To vary the bound over the prediction horizon from time *k* to time *k*+*p*–1, specify a vector of up to *p* values. Here, *k* is the current time and *p* is the prediction horizon. If you specify fewer than *p* values, the final bound is used for the remaining steps of the prediction horizon.

MV rate bounds are always hard constraints. Use stage inequality constraints to implement soft bounds (see Stages).

RateMax — MV rate of change hard upper bound
`Inf` (default) | nonnegative scalar | vector

MV rate of change hard upper bound, specified as a nonnegative scalar or vector. The MV rate of change at stage *i* is defined as *MV*(*i*) - *MV*(*i*–1). By default, this upper bound is `+Inf`.

To use the same bound across the prediction horizon, specify a scalar value. If UseMVRate is `false` this value is ignored.

To vary the bound over the prediction horizon from time *k* to time *k*+*p*–1, specify a vector of up to *p* values. Here, *k* is the current time and *p* is the prediction horizon. If you specify fewer than *p* values, the final bound is used for the remaining steps of the prediction horizon.

MV Rate bounds are always hard constraints. Use stage inequality constraint to implement soft bounds (see Stages).

Name — MV name
string | character vector

MV name, specified as a string or character vector. The default MV name is `"u#"`, where # is its input index.

Units — MV units
`""` (default) | string | character vector

MV units, specified as a string or character vector.

MeasuredDisturbances — Measured disturbance information
structure array

Measured disturbance (MD) information, specified as a structure array with N_{md} elements, where N_{md} is the number of measured disturbances. If your model does not have measured disturbances, then

`MeasuredDisturbances` is `[]`. To access this property, you can use the alias `MD` instead of `MeasuredDisturbances`.

Each structure element has the following fields.

Name — MD name
string | character vector

MD name, specified as a string or character vector. The default MD name is `"u#"`, where `#` is its input index.

Units — MD units
`""` (default) | string | character vector

MD units, specified as a string or character vector.

Stages — Stage cost and constraint functions
structure

Stage cost and constraint functions, specified as an array of `p+1` structures (where `p` is the prediction horizon), each one with the following fields.

CostFcn — Cost function at stage i
string | character vector | function handle

Cost function at stage `i` (where `i` ranges from `1` to `p+1`), specified as a string, character vector, or function handle. The overall cost function of the nonlinear MPC problem is the sum of the cost functions at each stage.

You can specify your stage cost function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Stages(i).CostFcn = 'myCostFunction';
  ```

- Handle to a local function or a function defined in the current working folder or on the MATLAB path

  ```
  Stages(i).CostFcn = @myCostFunction;
  ```

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

In the most general case in which UseMVRate is `true`, and both `Stages(i).ParameterLength` and `Stages(i).SlackVariableLength` are greater than `0`, the cost function must have the following inputs and outputs.

```
Ji = myCostFunction(i,x,u,dmv,e,pv);
```

Here:

- `Ji` is a double scalar expressing the cost for stage `i`.
- `i` is the stage number from `1` (current control interval) to `p+1` (end of the prediction horizon).
- `x` is the state vector.
- `u` is the input vector.
- `dmv` is the manipulated variable rate vector (change with respect to previous control interval).
- `e` is the stage slack variable vector.
- `pv` is the stage parameter vector.

If `UseMVRate` is `false`, omit the `dmv` input.

If `Stages(i).SlackVariableLength` is `0`, omit the `e` input.

If `Stages(i).ParameterLength` is `0`, omit the `pv` input.

In summary:

```
if UseMVRate is true
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            Ji = myCostFunction(i,x,u,dmv,e,pv);
        else
            Ji = myCostFunction(i,x,u,dmv,e);
        end
    else
        if Stages(i).ParameterLength>0
            Ji = myCostFunction(i,x,u,dmv,pv);
        else
            Ji = myCostFunction(i,x,u,dmv);
        end
    end
else
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            Ji = myCostFunction(i,x,u,e,pv);
        else
            Ji = myCostFunction(i,x,u,e);
        end
    else
        if Stages(i).ParameterLength>0
            Ji = myCostFunction(i,x,u,pv);
        else
            Ji = myCostFunction(i,x,u);
        end
    end
end
```

Note that you can also write separate functions for separate stages as long as their name is specified in `Stages(i).CostFcn` and all functions have the required number of inputs and outputs, in the required order.

For more information, see "Specify Prediction Model for Nonlinear MPC".

CostJacFcn — Gradient of the cost function at stage i
string | character vector | function handle

Gradient of the cost function at stage i (where i ranges from 1 to p+1), specified as a string, character vector, or function handle. It is best practice to use Jacobians (in this case, gradients) whenever they are available, since they improve optimization efficiency. If you do not specify a Jacobian for a given function, the nonlinear programming solver must numerically compute the Jacobian at each optimization step.

Note You can automatically generate a Jacobian function using `generateJacobianFunction`.

You can specify your stage cost Jacobian function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Stages(i).CostJacFcn = 'myCostJacFunction';
  ```
- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Stages(i).CostJacFcn = @myCostJacFunction;
  ```

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

In the most general case in which UseMVRate is `true`, and both `Stages(i).ParameterLength` and `Stages(i).SlackVariableLength` are greater than `0`, the stage cost gradient function must have the following inputs and outputs.

```
[Gx,Gmv,Gdmv,Ge] = myCostJacFunction(i,x,u,dmv,e,pv);
```

where

- `Gx` is the gradient of cost function for stage i with respect to the state vector x. It must be a column vector with N_x elements, where N_x is the number of states.
- `Gmv` is the gradient of the cost function for stage i with respect to the manipulated variable vector mv. It must be a column vector with N_{mv} elements, where N_{mv} is the number of manipulated variables.
- `Gdmv` is the gradient of the cost function for stage i with respect to the manipulated variable vector change dmv. It must be a column vector with N_{mv} elements, where N_{mv} is the number of manipulated variables.
- `Ge` is the gradient of the cost function for stage i with respect to the stage slack variable vector e. It must be a column vector with N_e elements, where N_e is the number of stage slack variables.
- `i` is the stage number from 1 (current control interval) to p+1 (end of the prediction horizon).
- `x` is the state vector.
- `u` is the input vector.
- `dmv` is the manipulated variable rate vector (change with respect to the previous control interval).
- `e` is the stage slack variable vector.

- pv is the stage parameter vector.

If UseMVRate is false, omit the dmv input and the Gdmv output.

If Stages(i).SlackVariableLength is 0, omit the e input and the Ge output.

If Stages(i).ParameterLength is 0, omit the pv input.

In summary:

```
if UseMVRate is true
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            [Gx,Gmv,Gdmv,Ge] = myCostJacFunction(i,x,u,dmv,e,pv);
        else
            [Gx,Gmv,Gdmv,Ge] = myCostJacFunction(i,x,u,dmv,e);
        end
    else
        if Stages(i).ParameterLength>0
            [Gx,Gmv,Gdmv] = myCostJacFunction(i,x,u,dmv,pv);
        else
            [Gx,Gmv,Gdmv] = myCostJacFunction(i,x,u,dmv);
        end
    end
else
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            [Gx,Gmv,Ge] = myCostJacFunction(i,x,u,e,pv);
        else
            [Gx,Gmv,Ge] = myCostJacFunction(i,x,u,e);
        end
    else
        if Stages(i).ParameterLength>0
            [Gx,Gmv] = myCostJacFunction(i,x,u,pv);
        else
            [Gx,Gmv] = myCostJacFunction(i,x,u);
        end
    end
end
```

Note that you can also write separate functions for separate stages as long as their name is specified in Stages(i).CostJacFcn and all functions have the required number of inputs and outputs, in the required order.

For more information, see "Specify Prediction Model for Nonlinear MPC".

EqConFcn — Equality constraint function at stage i
string | character vector | function handle

Equality constraint function at stage i (where i ranges from 1 to p), specified as a string, character vector, or function handle. Note that specifying an equality constraint for the last stage (p+1) is not supported. Use the TerminalState field of the Model property instead.

You can specify your stage equality constraint function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

```
Stages(i).EqConFcn = 'myEqConFunction';
```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

```
Stages(i).EqConFcn = @myEqConFunction;
```

For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

In the most general case in which UseMVRate is `true`, and `Stages(i).ParameterLength` is greater than `0`, the equality constraint function must have the following inputs and outputs.

```
Ceq = myEqConFunction(i,x,u,dmv,pv);
```

where

- `Ceq` is a vector expressing the equality constraints for stage `i`. At any feasible solution of the MPC problem the returned `Ceq` must be equal to `0`. Note that the number of elements in `Ceq` must be less than the number of manipulated variables otherwise the problem is overspecified and generally infeasible.
- `i` is the stage number from `1` (current control interval) to `p+1` (end of the prediction horizon).
- `x` is the state vector.
- `u` is the input vector.
- `dmv` is the manipulated variable rate vector (change with respect to previous control interval).
- `pv` is the stage parameter vector.

If `UseMVRate` is `false`, omit the `dmv` input.

If `Stages(i).ParameterLength` is `0`, omit the `pv` input.

In summary:

```
if UseMVRate is true
        if Stages(i).ParameterLength>0
            Ceq = myEqConFunction(i,x,u,dmv,pv);
        else
            Ceq = myEqConFunction(i,x,u,dmv);
        end
else
        if Stages(i).ParameterLength>0
            Ceq = myEqConFunction(i,x,u,pv);
        else
            Ceq = myEqConFunction(i,x,u);
        end
end
```

Note that you can also write separate functions for separate stages as long as their name is specified in `Stages(i).EqConFcn` and all functions have the required number of inputs and outputs, in the required order.

For more information, see "Specify Prediction Model for Nonlinear MPC".

EqConJacFcn — Jacobian of equality constraint function at stage i
string | character vector | function handle

Jacobian of the equality constraint function at stage i (where i ranges from 1 to p), specified as a string, character vector, or function handle. Note that specifying an equality constraint (and hence its Jacobian function) for the last stage (p+1) is not supported.

It is best practice to use Jacobians whenever they are available, since they improve optimization efficiency. If you do not specify a Jacobian for a given function, the nonlinear programming solver must numerically compute the Jacobian.

You can specify your stage equality constraint Jacobian function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Stages(i).EqConJacFcn = 'myEqConJacFunction';
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Stages(i).EqConJacFcn = @myEqConJacFunction;
  ```

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

In the most general case in which UseMVRate is true, and Stages(i).ParameterLength is greater than 0, the equality constraint Jacobian function must have the following inputs and outputs.

```
[Ceqx,Ceqmv,Ceqdmv] = myEqConJacFunction(i,x,u,dmv,pv);
```

where

- Ceqx is the Jacobian of the equality constraint function for stage i, with respect to the state vector x. It must be a matrix with N_{Ceq} rows and N_x columns, where N_{Ceq} is the number of stage equality constraints and N_x the number of states. Note that N_{Ceq} has to be less than N_{Cmv} otherwise the problem is overdetermined and generally infeasible.
- Ceqmv is the Jacobian of the equality constraint function for stage i, with respect to the manipulated variable vector mv. It must be a matrix with N_{Ceq} rows and N_{mv} columns, where N_{Ceq} is the number of stage equality constraints and N_{mv} the number of manipulated variables.
- Ceqdmv is the Jacobian of the equality constraint function for stage i, with respect to the manipulated variable vector change (rate) dmv. It must be a matrix with N_{Ceq} rows and N_{mv} columns, where N_{Ceq} is the number of stage equality constraints and N_{mv} the number of manipulated variables.
- i is the stage number from 1 (current control interval) to p+1 (end of the prediction horizon).
- x is the state vector.
- u is the input vector.

- `dmv` is the manipulated variable rate vector (change with respect to previous control interval).
- `pv` is the stage parameter vector.

If `UseMVRate` is `false`, omit the `dmv` input and the `Ceqdmv` output.

If `Stages(i).ParameterLength` is `0`, omit the `pv` input.

In summary

```
if UseMVRate is true
        if Stages(i).ParameterLength>0
            [Ceqx,Ceqmv,Ceqdmv] = myEqConJacFunction(i,x,u,dmv,pv);
        else
            [Ceqx,Ceqmv,Ceqdmv] = myEqConJacFunction(i,x,u,dmv);
        end
else
        if Stages(i).ParameterLength>0
            [Ceqx,Ceqmv] = myEqConJacFunction(i,x,u,pv);
        else
            [Ceqx,Ceqmv] = myEqConJacFunction(i,x,u);
        end
end
```

Note that you can also write separate functions for separate stages as long as their name is specified in `Stages(i).EqConJacFcn` and all functions have the required number of inputs and outputs, in the required order.

For more information, see "Specify Prediction Model for Nonlinear MPC".

`IneqConFcn` — Inequality constraint function at stage `i`
string | character vector | function handle

Inequality constraint function at stage `i` (where `i` ranges from `1` to `p+1`), specified as a string, character vector, or function handle.

You can specify your stage inequality constraint function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Stages(i).IneqConFcn = 'myIneqConFunction';`

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  `Stages(i).IneqConFcn = @myIneqConFunction;`

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

In the most general case in which UseMVRate is `true`, and both `Stages(i).ParameterLength` and `Stages(i).SlackVariableLength` are greater than `0`, the inequality constraint function must have the following inputs and outputs.

```
C = myIneqConFunction(i,x,u,dmv,e,pv);
```

Here:

- `C` is a vector expressing the inequality constraints for stage `i`. For any feasible solution of the MPC problem, `C` must be non-positive.
- `i` is the stage number from `1` (current control interval) to `p+1` (end of the prediction horizon).
- `x` is the state vector.
- `u` is the input vector.
- `dmv` is the manipulated variable rate vector (change with respect to previous control interval).
- `e` is the stage slack variable vector.
- `pv` is the stage parameter vector.

If `UseMVRate` is `false`, omit the `dmv` input.

If `Stages(i).SlackVariableLength` is `0`, omit the `e` input.

If `Stages(i).ParameterLength` is `0`, omit the `pv` input.

In summary:

```
if UseMVRate is true
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            C = myIneqConFunction(i,x,u,dmv,e,pv);
        else
            C = myIneqConFunction(i,x,u,dmv,e);
        end
    else
        if Stages(i).ParameterLength>0
            C = myIneqConFunction(i,x,u,dmv,pv);
        else
            C = myIneqConFunction(i,x,u,dmv);
        end
    end
else
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            C = myIneqConFunction(i,x,u,e,pv);
        else
            C = myIneqConFunction(i,x,u,e);
        end
    else
        if Stages(i).ParameterLength>0
            C = myIneqConFunction(i,x,u,pv);
        else
            C = myIneqConFunction(i,x,u);
        end
    end
end
```

Note that you can also write separate functions for separate stages as long as their name is specified in `Stages(i).IneqConFcn` and that all functions have the required number of inputs and outputs, in the required order.

For more information, see "Specify Prediction Model for Nonlinear MPC".

IneqConJacFcn — Jacobian of the inequality constraint function at stage i
string | character vector | function handle

Jacobian of the inequality constraint function at stage i (where i ranges from 1 to p+1), specified as a string, character vector, or function handle. It is best practice to use Jacobians whenever they are available, since they improve optimization efficiency. If you do not specify a Jacobian for a given function, the nonlinear programming solver must numerically compute the Jacobian.

You can specify your stage constraint Jacobian function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  ```
  Stages(i).IneqConJacFcn = 'myIneqConJacFunction';
  ```

- Handle to a local function, or a function defined in the current working folder or on the MATLAB path

  ```
  Stages(i).IneqConJacFcn = @myIneqConJacFunction;
  ```

  For more information on local functions, see "Local Functions".

Note Only functions defined in a separate file in the current folder or on the MATLAB path are supported for C/C++ code generation. Therefore, specifying state, cost, or constraint functions (or their Jacobians) as local functions is not recommended.

In the most general case in which UseMVRate is true, and both Stages(i).ParameterLength and Stages(i).SlackVariableLength are greater than 0, the stage cost Jacobian function must have the following inputs and outputs.

```
[Cx,Cmv,Cdmv,Ce] = myEqConJacFunction(i,x,u,dmv,e,pv);
```

Here:

- Cx is the Jacobian of the inequality constraint function for stage i, with respect to the state vector x. It must be a matrix with N_C rows and N_x columns, where N_C is the number of stage inequality constraints and N_x the number of states.
- Cmv is the Jacobian of the inequality constraint function for stage i, with respect to the manipulated variable vector mv. It must be a matrix with N_C rows and N_{mv} columns, where N_C is the number of stage inequality constraints and N_{mv} the number of manipulated variables.
- Cdmv is the Jacobian of the inequality constraint function for stage i, with respect to the manipulated variable change (rate) dmv. It must be a matrix with N_C rows and N_{mv} columns, where N_C is the number of stage inequality constraints and N_{mv} the number of manipulated variables.
- Ce is the Jacobian of the inequality constraint function for stage i, with respect to the stage slack variable vector e. It must be a matrix with N_C rows and N_e columns, where N_C is the number of stage inequality constraints and N_e the number of stage slack variables.
- i is the stage number from 1 (current control interval) to p+1 (end of the prediction horizon).
- x is the state vector.
- u is the input vector.

- dmv is the manipulated variable rate vector (change with respect to previous control interval).
- e is the stage slack variable vector.
- pv is the stage parameter vector.

If UseMVRate is false, omit the dmv input and the Cdmv output.

If Stages(i).SlackVariableLength is 0, omit the e input and the Ce output.

If Stages(i).ParameterLength is 0, omit the pv input.

In summary:

```
if UseMVRate is true
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            [Cx,Cmv,Cdmv,Ce] = myIneqConJacFunction(i,x,u,dmv,e,pv);
        else
            [Cx,Cmv,Cdmv,Ce] = myIneqConJacFunction(i,x,u,dmv,e);
        end
    else
        if Stages(i).ParameterLength>0
            [Cx,Cmv,Cdmv] = myIneqConJacFunction(i,x,u,dmv,pv);
        else
            [Cx,Cmv,Cdmv] = myIneqConJacFunction(i,x,u,dmv);
        end
    end
else
    if Stages(i).SlackVariableLength>0
        if Stages(i).ParameterLength>0
            [Cx,Cmv,Ce] = myIneqConJacFunction(i,x,u,e,pv);
        else
            [Cx,Cmv,Ce] = myIneqConJacFunction(i,x,u,e);
        end
    else
        if Stages(i).ParameterLength>0
            [Cx,Cmv] = myIneqConJacFunction(i,x,u,pv);
        else
            [Cx,Cmv] = myIneqConJacFunction(i,x,u);
        end
    end
end
```

Note that you can also write separate functions for separate stages as long as their name is specified in Stages(i).IneqConFcn and that all functions have the required number of inputs and outputs, in the required order.

For more information, see "Specify Prediction Model for Nonlinear MPC".

SlackVariableLength — Length of the stage slack variable vector
0 (default) | nonnegative integer

Length of the slack variable vector used by the cost and constraint functions at stage i, specified as a nonnegative integer. You can use slack variables to implement soft constraints for a given stage, using the corresponding IneqConFcn and CostFcn functions.

ParameterLength — Length of the parameter vector
0 (default) | nonnegative integer

Length of the parameter vector used by the cost and constraint functions at stage `i`, specified as a nonnegative integer. If any stage uses parameters, this value must be positive, and as a consequence all the stage functions must have a parameter vector as their last input argument.

`Optimization` — Custom optimization functions and solver
structure

Custom optimization functions and solver, specified as a structure with the following fields.

`CustomSolverFcn` — Custom nonlinear programming solver
`[]` (default) | string | character vector | function handle

Custom nonlinear programming solver function, specified as a string, character vector, or function handle. If you do not have Optimization Toolbox software, you must specify your own custom nonlinear programming solver. You can specify your custom solver function in one of the following ways:

- Name of a function in the current working folder or on the MATLAB path, specified as a string or character vector

  `Optimization.CustomSolverFcn = "myNLPSolver";`

- Handle to a function in the current working folder or on the MATLAB path

  `Optimization.CustomSolverFcn = @myNLPSolver;`

For more information, see "Configure Optimization Solver for Nonlinear MPC".

`SolverOptions` — Solver options
options object for `fmincon` | `[]`

Solver options, specified as an options object for `fmincon` or `[]`.

If you have Optimization Toolbox software, `SolverOptions` contains an options object for the `fmincon` solver.

If you do not have Optimization Toolbox, `SolverOptions` is `[]`.

For more information, see "Configure Optimization Solver for Nonlinear MPC".

`UseSuboptimalSolution` — Flag indicating whether a suboptimal solution is acceptable
`false` (default) | `true`

Flag indicating whether a suboptimal solution is acceptable, specified as a logical value. When the nonlinear programming solver reaches the maximum number of iterations without finding a solution (the exit flag is `0`), the controller:

- Freezes the MV values if `UseSuboptimalSolution` is `false`
- Applies the suboptimal solution found by the solver after the final iteration if `UseSuboptimalSolution` is `true`

To specify the maximum number of iterations, use `Optimization.SolverOptions.MaxIter`.

`PerturbationRatio` — Coefficient used to calculate perturbation sizes
`1e-6` (default) | positive scalar

Coefficient used to calculate the perturbation sizes applied to the decision variables when using forward finite differences to estimate derivatives. The perturbation size vector for the decision variable vector z is `PerturbationRatio*max(abs(z),1)`. The default value for this parameter is `1e−6`. If your prediction model is stiff and your cost/constraint terms are sensitive, use a smaller value such as `1e−8`.

Object Functions

generateJacobianFunction	Generate MATLAB Jacobian functions for multistage nonlinear MPC using automatic differentiation
validateFcns	Examine prediction model and custom functions of nlmpc or nlmpcMultistage objects for potential problems
nlmpcmove	Compute optimal control action for nonlinear MPC controller
getSimulationData	Create data structure to simulate multistage MPC controller with nlmpcmove

Examples

Create Multistage Nonlinear MPC object

Create a multistage nonlinear MPC object with a prediction horizon of 5 steps, 2 states, and 1 manipulated variable.

```
msobj = nlmpcMultistage(5,2,1);
```

Create Multistage Nonlinear MPC object with Measured Disturbance Inputs

Create a multistage nonlinear MPC object with a prediction horizon of 5 steps, 2 states, and 2 inputs, where the first input is a measured disturbance and the second is a manipulated variable.

```
msobj = nlmpcMultistage(5,2,'MV',2,'MD',1);
```

Create Multistage Nonlinear MPC object with Unmeasured Disturbance Inputs

Create a multistage nonlinear MPC object with a prediction horizon of 5 steps, 2 states, and 2 inputs, where the first input is a manipulated variable and the second is an unmeasured disturbance.

```
msobj = nlmpcMultistage(5,2,'MV',1,'UD',2);
```

Create Multistage Nonlinear MPC Object with Measured and Unmeasured Disturbance Inputs

Create a multistage nonlinear MPC object with a prediction horizon of 6 steps, 3 states, and 4 inputs, where the first two inputs are measured disturbances, the third is the manipulated variable, and the fourth is an unmeasured disturbance.

```
msobj = nlmpcMultistage(6, 3, 'MV',3,'MD',[1 2],'UD',4);
```

Set a sampling time of 2 seconds and display the msobj object

```
msobj.Ts = 2

msobj =

  nlmpcMultistage with properties:

                      Ts: 2
        PredictionHorizon: 6
               UseMVRate: 0
              Dimensions: [1×1 struct]
                   Model: [1×1 struct]
                  States: [1×3 struct]
     ManipulatedVariables: [1×1 struct]
     MeasuredDisturbances: [1×2 struct]
                  Stages: [1×7 struct]
            Optimization: [1×1 struct]
```

Version History
Introduced in R2021a

See Also

Functions
generateJacobianFunction | validateFcns | nlmpcmove | nlmpcmoveCodeGeneration | getSimulationData | getCodeGenerationData

Objects
nlmpc | nlmpcmoveopt

Blocks
Multistage Nonlinear MPC Controller

Topics
"Landing a Vehicle Using Multistage Nonlinear MPC"
"Truck and Trailer Automatic Parking Using Multistage Nonlinear MPC"
"Nonlinear MPC"

nlmpcmoveopt

Option set for `nlmpcmove` function

Description

To specify options for the `nlmpcmove` function, use an `nlmpcmoveopt` option set.

Using this option set, you can specify run-time values for a subset of controller properties, such as tuning weights and constraints. If you do not specify a value for one of the `nlmpcmoveopt` properties, the corresponding value defined in the `nlmpc` controller object is used instead.

Creation

Syntax

options = nlmpcmoveopt

Description

`options = nlmpcmoveopt` creates a default set of options for the `nlmpcmove` function. To modify the property values, use dot notation.

Properties

`OutputWeights` — Output variable tuning weights
[ ] (default) | row vector | matrix

Output variable tuning weights that replace the `Weights.OutputVariables` property of the controller at run time, specified as a row vector or matrix of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_y, where N_y is the number of output variables.

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, specify an array with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the output variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

`MVWeights` — Manipulated variable tuning weights
[ ] (default) | row vector | matrix

Manipulated variable tuning weights that replace the `Weights.ManipulatedVariables` property of the controller at run time, specified as a row vector or matrix of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each

row contains the manipulated variable tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

MVRateWeights — Manipulated variable rate tuning weights
[] (default) | row vector | matrix

Manipulated variable rate tuning weights that replace the `Weights.ManipulatedVariablesRate` property of the controller at run time, specified as a row vector or matrix of nonnegative values.

To use the same weights across the prediction horizon, specify a row vector of length N_{mv}, where N_{mv} is the number of manipulated variables.

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, specify an array with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the manipulated variable rate tuning weights for one prediction horizon step. If you specify fewer than p rows, the weights in the final row are used for the remaining steps of the prediction horizon.

ECR Weight — Slack variable tuning weight
[] (default) | positive scalar

Slack variable tuning weight that replaces the `Weights.ECR` property of the controller at run time, specified as a positive scalar.

OutputMin — Output variable lower bounds
[] (default) | row vector | matrix

Output variable lower bounds, specified as a row vector of length N_y or a matrix with N_y columns, where N_y is the number of output variables. `OutputMin(:,i)` replaces the `OutputVariables(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

OutputMax — Output variable upper bounds
[] (default) | row vector | matrix

Output variable upper bounds, specified as a row vector of length N_y or a matrix with N_y columns, where N_y is the number of output variables. `OutputMax(:,i)` replaces the `OutputVariables(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVMin — Manipulated variable lower bounds
[] (default) | row vector | matrix

Manipulated variable lower bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVMin(:,i)` replaces the `ManipulatedVariables(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVMax — Manipulated variable upper bounds
[ ] (default) | row vector | matrix

Manipulated variable upper bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVMax(:,i)` replaces the `ManipulatedVariables(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVRateMin — Manipulated variable rate lower bounds
[ ] (default) | row vector | matrix

Manipulated variable rate lower bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMin(:,i)` replaces the `ManipulatedVariables(i).RateMin` property of the controller at run time. `MVRateMin` bounds must be nonpositive.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVRateMax — Manipulated variable rate upper bounds
[ ] (default) | row vector | matrix

Manipulated variable rate upper bounds, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables. `MVRateMax(:,i)` replaces the `ManipulatedVariables(i).RateMax` property of the controller at run time. `MVRateMax` bounds must be nonnegative.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateMin — State lower bounds
[] (default) | row vector | matrix

State lower bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMin(:,i)` replaces the `States(i).Min` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

StateMax — State upper bounds
[] (default) | row vector | matrix

State upper bounds, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states. `StateMax(:,i)` replaces the `States(i).Max` property of the controller at run time.

To use the same bounds across the prediction horizon, specify a row vector.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the final bounds are used for the remaining steps of the prediction horizon.

MVTarget — Manipulated variable targets
[] (default) | row vector | matrix

Manipulated variable targets, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables.

To use the same manipulated variable targets across the prediction horizon, specify a row vector.

To vary the targets over the prediction horizon (previewing) from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the targets for one prediction horizon step. If you specify fewer than p rows, the final targets are used for the remaining steps of the prediction horizon.

Parameters — Parameter values
{} (default) | cell vector

Parameter values used by the prediction model, custom cost function, and custom constraints, specified as a cell vector with length equal to the `Model.NumberOfParameters` property of the controller. If the controller has no parameters, then `Parameters` must be `{}`.

The controller, `nlmpcobj`, passes these parameters to the:

- Model functions in `nlmpcobj.Model` (`StateFcn` and `OutputFcn`)
- Cost function `nlmpcobj.Optimization.CustomCostFcn`
- Constraint functions in `nlmpcobj.Optimization` (`CustomEqConFcn` and `CustomIneqConFcn`)

- Jacobian functions in `nlmpcobj.Jacobian`
- Passivity functions and their Jacobians in `nlmpcobj.Passivity`

The order of the parameters must match the order defined for these functions.

`X0` — Initial guesses for the optimal state solutions
`[ ]` (default) | vector | matrix

Initial guesses for the optimal state solutions, specified as a row vector of length N_x or a matrix with N_x columns, where N_x is the number of states.

To use the same initial guesses across the prediction horizon, specify a row vector.

To vary the initial guesses over the prediction horizon from time $k+1$ to time $k+p$, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the final guesses are used for the remaining steps of the prediction horizon.

If `X0` is `[ ]`, the default initial guesses are the current states of the prediction model (`x` input argument to `nlmpcmove`).

In general, during closed-loop simulation, you do not specify `X0` yourself. Instead, when calling `nlmpcmove`, return the `opt` output argument, which is an `nlmpcmoveopt` object. `opt.X0` contains the calculated optimal state trajectories as initial guesses. You can then pass `opt` in as the `options` input argument to `nlmpcmove` for the next control interval. These steps are a best practice, even if you do not specify any other run-time options.

`MV0` — Initial guesses for the optimal manipulated variable solutions
`[ ]` (default) | vector | matrix

Initial guesses for the optimal manipulated variable solutions, specified as a row vector of length N_{mv} or a matrix with N_{mv} columns, where N_{mv} is the number of manipulated variables.

To use the same initial guesses across the prediction horizon, specify a row vector.

To vary the initial guesses over the prediction horizon from time k to time $k+p$-1, specify a matrix with up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the final guesses are used for the remaining steps of the prediction horizon.

If `MV0` is `[ ]`, the default initial guesses are the control signals used in the plant at the previous control interval (`lastmv` input argument to `nlmpcmove`).

In general, during closed-loop simulation, you do not specify `MV0` yourself. Instead, when calling `nlmpcmove`, return the `opt` output argument, which is an `nlmpcmoveopt` object. `opt.MV0` contains the calculated optimal manipulated variable trajectories as initial guesses. You can then pass `opt` in as the `options` input argument to `nlmpcmove` for the next control interval. These steps are a best practice, even if you do not specify any other run-time options.

`Slack0` — Initial guess for the slack variable at the solution
`[ ]` (default) | nonnegative scalar

Initial guess for the slack variable at the solution, specified as a nonnegative scalar. If `Slack0` is `[ ]`, the default initial guess is `0`.

In general, during closed-loop simulation, you do not specify `Slack0` yourself. Instead, when calling `nlmpcmove`, return the `opt` output argument, which is an `nlmpcmoveopt` object. `opt.Slack` contains the calculated slack variable as an initial guess. You can then pass `opt` in as the `options` input argument to `nlmpcmove` for the next control interval. These steps are a best practice, even if you do not specify any other run-time options.

Object Functions

nlmpcmove   Compute optimal control action for nonlinear MPC controller

Examples

Specify Run-Time Parameters for Nonlinear MPC

Create a default `nlmpcmoveopt` option set.

```
options = nlmpcmoveopt;
```

Specify the run-time values for the controller prediction model parameters. For this example, assume that the controller has the following optional parameters, which are input arguments to all the prediction model functions and custom functions of the controller.

- Sample time of the model, specified as a single numeric value. Specify a value of `0.25`.
- Gain factors, specified as a two-element row vector. Specify a value of `[0.7 0.35]`.

The order in which you specify the parameters must match the order specified in the custom function argument lists. Also, the dimensions of the parameters must match the dimensions expected by the custom functions.

```
options.Parameters = {0.25,[0.7 0.35]};
```

To use these parameters when computing optional control actions for a nonlinear MPC controller, pass `options` to the `nlmpcmove` function.

Version History
Introduced in R2018b

See Also

Functions
validateFcns | nlmpcmove | nlmpcmoveCodeGeneration | getSimulationData | getCodeGenerationData

Objects
nlmpc | nlmpcMultistage

Topics
"Trajectory Optimization and Control of Flying Robot Using Nonlinear MPC"
"Plan and Execute Task- and Joint-Space Trajectories Using KINOVA Gen3 Manipulator" (Robotics System Toolbox)

"Nonlinear MPC"

Blocks

MPC Controller

Simulate model predictive controller
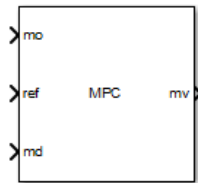
Libraries:
Model Predictive Control Toolbox

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 variable (`mv`) by solving a quadratic programming problem using either the default KWIK solver or a custom QP solver. For more information, see "QP Solvers".

To use the block in simulation and code generation, you must specify an `mpc` object, which defines a model predictive controller. This controller must have already been designed for the plant that it controls.

Because the MPC Controller block uses MATLAB Function blocks, it requires compilation each time you change the MPC object and block. Also, because MATLAB does not allow compiled code to reside in any MATLAB product folder, you must use a non-MATLAB folder to work on your Simulink model when you use MPC blocks.

Ports

Input

Required Inputs

mo — Measured outputs
vector

Measured outputs, specified as a vector signal. The block uses the measured plant outputs to improve its state estimates. If your controller uses default state estimation, you must connect the measured plant outputs to the **mo** input port. If your controller uses custom state estimation, you must connect the estimated plant states to the **x[k|k]** input port.

Dependencies

To enable this port, clear the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

x[k|k] — Custom state estimate
vector

Custom state estimate, specified as a vector signal. The block uses the connected state estimates instead of estimating the states using the built-in estimator. If your controller uses custom state

estimation, you must connect the current state estimates to the **x[k|k]** input port. If your controller uses default state estimation, you must connect the measured output to the **mo** input port.

Even though noise model states (if any) are not used in MPC optimization, the custom state vector must contain all the states defined in the `mpcstate` object of the controller, including the plant, disturbance, and noise model states.

Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

Dependencies

To enable this port, select the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

ref — Model output reference values
row vector | matrix

Plant output reference values, specified as a row vector signal or matrix signal.

To use the same reference values across the prediction horizon, connect **ref** to a row vector signal with N_Y elements, where N_y is the number of output variables. Each element specifies the reference for an output variable.

To vary the references over the prediction horizon (previewing) from time $k+1$ to time $k+p$, connect **ref** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the references for one prediction horizon step. If you specify fewer than p rows, the final references are used for the remaining steps of the prediction horizon.

Additional Inputs

md — input
row vector | matrix

If your controller prediction model has measured disturbances you must enable this port and connect to it a row vector or matrix signal.

To use the same measured disturbance values across the prediction horizon, connect **md** to a row vector signal with N_{md} elements, where N_{md} is the number of manipulated variables. Each element specifies the value for a measured disturbance.

To vary the disturbances over the prediction horizon (previewing) from time k to time $k+p$, connect **md** to a matrix signal with N_{md} columns and up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbances for one prediction horizon step. If you specify fewer than $p+1$ rows, the final disturbances are used for the remaining steps of the prediction horizon.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

ext.mv — Control signals used in plant at previous control interval
vector

Control signals used in the plant at the previous control interval, specified as a vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables. Use this input port to improve state estimation accuracy when:

- You know your controller is not always in control of the plant.
- The actual MV signals applied to the plant can potentially differ from the values generated by the controller, such as in control signal saturation.

Controller state estimation assumes that the MVs are piecewise constant. Therefore, at time t_k, the **ext.mv** value must contain the effective MVs between times t_{k-1} and t_k. For example, if the MVs are actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

Note

- Connect **ext.mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode (that is, the controller output is not driving the plant), then feeding the actual control signal to **ext.mv** can help achieve bumpless transfer when the controller is switched back online.
- When the controller is driving the plant, insert a Memory block or Unit Delay block to feed back the MV signal applied to the plant at the previous control interval. This also avoids a direct feedthrough from the **ext.mv** inport to the **mv** outport, therefore preventing algebraic loops in the Simulink model.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **External manipulated variable** parameter.

switch — Enable or disable optimization
scalar

To turn off the controller optimization calculations, connect **switch** to a nonzero signal.

Disabling optimization calculations reduces computational effort when the controller output is not needed, such as when the system is operating manually or another controller has taken over. However, the controller continues to update its internal state estimates in the usual way. Therefore, it is ready to resume optimization calculations whenever the **switch** signal returns to zero. While controller optimization is off, the block passes the current **ext.mv** signal to the controller output. If the **ext.mv** inport is not enabled, the controller output is held at the value it had when optimization was disabled.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

mv.target — Manipulated variable targets
vector

To specify manipulated variable targets, enable this input port, and connect a vector signal. To make a given manipulated variable track its specified target value, you must also specify a nonzero tuning weight for that manipulated variable.

The supplied **mv.target** values at run-time apply across the prediction horizon.

Dependencies

To enable this port, select the **Targets for manipulated variables** parameter.

Online Constraints

ymin — Minimum output variable constraints
vector | matrix

To specify run-time minimum output variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `OutputVariables.Min` property of its `mpc` controller object. If an output variable has no lower bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **ymin** to a matrix signal with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one output variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `ith` column of the **ymin** signal corresponds to the `ith` plant output, and replaces the `OutputVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `OutputVariables(i).Min` in the `mpc` object (a constant bound for the ith plant output to be applied to all prediction steps)

ymin Dimension	Replacement Behavior
Scalar **ymin** (single output, constant bound)	**ymin** replaces the constant bound defined in `OutputVariables(i).Min`
Column vector **ymin** (single output, time-varying bound)	**ymin** replaces the constant bound defined in `OutputVariables(i).Min` with a time-varying bound.
Row vector **ymin** (multiple outputs, constant bounds)	The `ith` element of **ymin** replaces the constant bound defined in `OutputVariables(i).Min`
Matrix **ymin** (multiple outputs, time-varying bounds)	The `ith` column of **ymin** replaces the constant bound defined in `OutputVariables(i).Min` with a time-varying bound.

Vector OutputVariables(i).Min in the mpc object (a time-varying bound for the ith plant output with different values at different prediction steps)

ymin Dimension	Replacement Behavior
Scalar **ymin** (single output, constant bound)	**ymin** replaces the first finite entry in OutputVariables.Min and the remaining entries in OutputVariables.Min shift up or down with the same amount of displacement to retain the profile defined by the original OutputVariables.Min vector.
Column vector **ymin** (single output, time-varying bound)	**ymin** replaces the time-varying bound defined in OutputVariables(i).Min, and the original bound profile is discarded.
Row vector **ymin** (multiple outputs, constant bounds)	The ith element of **ymin** replaces the first finite entry in OutputVariables(i).Min and the remaining entries in OutputVariables(i).Min shift up or down with the same amount of displacement to retain the profile defined by the original OutputVariables(i).Min vector.
Matrix **ymin** (multiple outputs, time-varying bounds).	The ith column of **ymin** replaces the time-varying bound defined in OutputVariables(i).Min, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Lower OV limits** parameter.

ymax — Maximum output variable constraints
vector | matrix

To specify run-time maximum output variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the OutputVariables.Max property of its mpc controller object. If an output variable has no upper bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **ymax** to a matrix signal with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one output variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The ith column of the **ymax** signal corresponds to the ith plant output, and replaces the OutputVariables(i).Max property of the mpc object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `OutputVariables(i).Max` in the `mpc` object (a constant bound for the ith plant output to be applied to all prediction steps)

ymax Dimension	Replacement Behavior
Scalar **ymax** (single output, constant bound)	**ymax** replaces the constant bound defined in `OutputVariables(i).Max`
Column vector **ymax** (single output, time-varying bound)	**ymax** replaces the constant bound defined in `OutputVariables(i).Max` with a time-varying bound.
Row vector **ymax** (multiple outputs, constant bounds)	The `ith` element of **ymax** replaces the constant bound defined in `OutputVariables(i).Max`
Matrix **ymax** (multiple outputs, time-varying bounds)	The `ith` column of **ymax** replaces the constant bound defined in `OutputVariables(i).Max` with a time-varying bound.

Vector `OutputVariables(i).Max` in the `mpc` object (a time-varying bound for the ith plant output with different values at different prediction steps)

ymax Dimension	Replacement Behavior
Scalar **ymax** (single output, constant bound)	**ymax** replaces the first finite entry in `OutputVariables.Max` and the remaining entries in `OutputVariables.Max` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables.Max` vector.
Column vector **ymax** (single output, time-varying bound)	**ymax** replaces the time-varying bound defined in `OutputVariables(i).Max`, and the original bound profile is discarded.
Row vector **ymax** (multiple outputs, constant bounds)	The `ith` element of **ymax** replaces the first finite entry in `OutputVariables(i).Max` and the remaining entries in `OutputVariables(i).Max` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables(i).Max` vector.
Matrix **ymax** (multiple outputs, time-varying bounds).	The `ith` column of **ymax** replaces the time-varying bound defined in `OutputVariables(i).Max`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Upper OV limits** parameter.

umin — Minimum manipulated variable constraints
vector | matrix

To specify run-time minimum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariables.Min` property of its `mpc` controller object. If a manipulated variable has no lower bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time k to time $k+p$-1, connect **umin** to a matrix signal with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k

is the current time, and *p* is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than *p* rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one manipulated variable, and a vector signal with no more than *p* entries is connected, then these entries are used across the prediction horizon.

The ith column of the **umin** signal corresponds to the ith manipulated variable, and replaces the `ManipulatedVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `ManipulatedVariables(i).Min` in the `mpc` object (a constant bound for the ith manipulated variable to be applied to all prediction steps)

umin Dimension	Replacement Behavior
Scalar **umin** (single output, constant bound)	**umin** replaces the constant bound defined in `ManipulatedVariables(i).Min`
Column vector **umin** (single output, time-varying bound)	**umin** replaces the constant bound defined in `ManipulatedVariables(i).Min` with a time-varying bound.
Row vector **umin** (multiple outputs, constant bounds)	The ith element of **umin** replaces the constant bound defined in `ManipulatedVariables(i).Min`
Matrix **umin** (multiple outputs, time-varying bounds)	The ith column of **umin** replaces the constant bound defined in `ManipulatedVariables(i).Min` with a time-varying bound.

Vector `ManipulatedVariables(i).Min` in the `mpc` object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

umin Dimension	Replacement Behavior
Scalar **umin** (single output, constant bound)	**umin** replaces the first finite entry in `ManipulatedVariables.Min` and the remaining entries in `ManipulatedVariables.Min` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables.Min` vector.
Column vector **umin** (single output, time-varying bound)	**umin** replaces the time-varying bound defined in `ManipulatedVariables(i).Min`, and the original bound profile is discarded.
Row vector **umin** (multiple outputs, constant bounds)	The ith component of **umin** replaces the first finite entry in `ManipulatedVariables(i).Min` and the remaining entries in `ManipulatedVariables(i).Min` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables(i).Min` vector.
Matrix **umin** (multiple outputs, time-varying bounds).	The ith column of **umin** replaces the time-varying bound defined in `ManipulatedVariables(i).Min`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Lower MV limits** parameter.

umax — Maximum manipulated variable constraints
vector | matrix

To specify run-time maximum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.Max` property of its `mpc` controller object. If a manipulated variable has no upper bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time k to time $k+p$-1, connect **umax** to a matrix signal with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one manipulated variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `ith` column of the **umax** signal corresponds to the `ith` manipulated variable, and replaces the `ManipulatedVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `ManipulatedVariables(i).Max` in the `mpc` object (a constant bound for the `ith` manipulated variable to be applied to all prediction steps)

umax Dimension	Replacement Behavior
Scalar **umax** (single output, constant bound)	**umax** replaces the constant bound defined in `ManipulatedVariables(i).Max`
Column vector **umax** (single output, time-varying bound)	**umax** replaces the constant bound defined in `ManipulatedVariables(i).Max` with a time-varying bound.
Row vector **umax** (multiple outputs, constant bounds)	The `ith` element of **umax** replaces the constant bound defined in `ManipulatedVariables(i).Max`
Matrix **umax** (multiple outputs, time-varying bounds)	The `ith` column of **umax** replaces the constant bound defined in `ManipulatedVariables(i).Max` with a time-varying bound.

Vector ManipulatedVariables(i).Max in the mpc object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

umax Dimension	Replacement Behavior
Scalar **umax** (single output, constant bound)	**umax** replaces the first finite entry in `ManipulatedVariables.Max` and the remaining entries in `ManipulatedVariables.Max` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables.Max` vector.
Column vector **umax** (single output, time-varying bound)	**umax** replaces the time-varying bound defined in `ManipulatedVariables(i).Max`, and the original bound profile is discarded.
Row vector **umax** (multiple outputs, constant bounds)	The ith element of **umax** replaces the first finite entry in `ManipulatedVariables(i).Max` and the remaining entries in `ManipulatedVariables(i).Max` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables(i).Max` vector.
Matrix **umax** (multiple outputs, time-varying bounds).	The ith column of **umax** replaces the time-varying bound defined in `ManipulatedVariables(i).Max`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Upper MV limits** parameter.

E — Manipulated variable constraint matrix
matrix

Manipulated variable constraint matrix, specified as an N_c-by-N_{mv} matrix signal, where N_c is the number of mixed input/output constraints and N_{mv} is the number of manipulated variables.

If you define E in the mpc object, you must connect a signal to the **E** input port. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **E** input port along with the **F**, **G**, and **S** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the E matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter.

F — Controlled output constraint matrix
matrix

Controlled output constraint matrix, specified as an N_c-by-N_y matrix signal, where N_c is the number of mixed input/output constraints and N_y is the number of plant outputs. If you define F in the mpc

object, you must connect a signal to the **F** input port with same number of rows. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **F** input port along with the **E**, **G**, and **S** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the F matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter.

G — Custom constraint vector
row vector

Custom constraint vector, specified as a row vector signal of length N_c, where N_c is the number of mixed input/output constraints. If you define G in the `mpc` object, you must connect a signal to the **G** input port with same number of rows. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **G** input port along with the **E**, **F**, and **S** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the G matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter.

S — Measured disturbance constraint matrix
matrix

Measured disturbance constraint matrix, specified as an N_c-by-n_N matrix signal, where N_c is the number of mixed input/output constraints, and N_v is the number of measured disturbances. If you define S in the `mpc` object, you must connect a signal to the **S** input port with same number of rows. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **S** input port along with the **E**, **F**, and **G** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the G matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter. This port is added only if the `mpc` object has measured disturbances.

Online Tuning Weights

y.wt — Output variable tuning weights
row vector | matrix

To specify run-time output variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.OutputVariables` property of its controller object. These tuning weights penalize deviations from output references.

If the MPC controller object uses constant output tuning weights over the prediction horizon, you can specify only constant output tuning weights at runtime. Similarly, if the MPC controller object uses output tuning weights that vary over the prediction horizon, you can specify only time-varying output tuning weights at runtime

To use constant tuning weights over the prediction horizon, connect **y.wt** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies a nonnegative tuning weight for an output variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, connect **y.wt** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **OV weights** parameter.

u.wt — Manipulated variable tuning weights
row vector | matrix

To specify run-time manipulated variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariables` property of its controller object. These tuning weights penalize deviations from MV targets.

If the MPC controller object uses constant manipulated variable tuning weights over the prediction horizon, you can specify only constant manipulated variable tuning weights at runtime. Similarly, if the MPC controller object uses manipulated variable tuning weights that vary over the prediction horizon, you can specify only time-varying manipulated variable tuning weights at runtime.

To use the same tuning weights over the prediction horizon, connect **u.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **u.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MV weights** parameter.

du.wt — Manipulated variable rate tuning weights
row vector | matrix

To specify run-time manipulated variable rate tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariablesRate` property of its controller object. These tuning weights penalize large changes in control moves.

If the MPC controller object uses constant manipulated variable rate tuning weights over the prediction horizon, you can specify only constant manipulated variable tuning rate weights at runtime. Similarly, if the MPC controller object uses manipulated variable rate tuning weights that vary over the prediction horizon, you can specify only time-varying manipulated variable rate tuning weights at runtime.

To use the same tuning weights over the prediction horizon, connect **du.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable rate. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **du.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MVRate weights** parameter.

ecr.wt — Slack variable tuning weight
scalar

To specify a run-time slack variable tuning weight, enable this input port and connect a scalar signal. If this port is disabled, the block uses the tuning weight specified in the `Weights.ECR` property of its controller object.

The slack variable tuning weight has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero. If there are soft constraints, increasing the **ecr.wt** value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

Dependencies

To enable this port, select the **ECR weight** parameter.

Online Horizons

p — Prediction horizon
positive integer

Prediction horizon, specified as positive integer signal. The prediction horizon signal value must be less than or equal to the **Maximum prediction horizon** parameter.

At run time, the values of **p** overrides the default prediction horizon specified in the controller object. For more information, see "Adjust Horizons at Run Time".

Dependencies

To enable this port, select the **Adjust prediction horizon and control horizon at run time** parameter.

m — Control horizon
positive integer | vector

Control horizon, specified as one of the following:

- Positive integer signal less than or equal to the prediction horizon.
- Vector signal of positive integers specifying blocking interval lengths. For more information, see "Manipulated Variable Blocking".

At run time, the values of m overrides the default control horizon specified in the controller object. For more information, see "Adjust Horizons at Run Time".

Dependencies

To enable this port, select the **Adjust prediction horizon and control horizon at run time** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, output as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the solver converges to a local optimum solution (**qp.status** is positive), then **mv** contains the optimal solution.

If the solver fails (**qp.status** is negative), then **mv** remains at its most recent successful solution; that is, the controller output freezes.

If the solver reaches the maximum number of iterations without finding an optimal solution (**qp.status** is zero) and the `Optimization.UseSuboptimalSolution` property of the controller is:

- `true`, then **mv** contains the suboptimal solution
- `false`, then **mv** then **mv** remains at its most recent successful solution

Additional Outputs

cost — Objective function cost
nonnegative scalar

Objective function cost, output as a nonnegative scalar signal. The cost quantifies the degree to which the controller has achieved its objectives. The cost value is calculated using the scaled MPC cost function in which every term is offset-free and dimensionless.

The cost value is only meaningful when the **qp.status** output is nonnegative.

Dependencies

To enable this port, select the **Optimal cost** parameter.

qp.status — Optimization status
integer

Optimization status, output as an integer signal.

If the controller solves the QP problem for 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. Therefore, a large value means a relatively slow block execution for this time interval.

The QP solver can fail to find an optimal solution for the following reasons:

- **qp.status** = 0 — The QP solver cannot find a solution within the maximum number of iterations specified in the mpc object. In this case, if the Optimizer.UseSuboptimalSolution property of the controller is false, the block holds its **mv** output at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.

- **qp.status** = -1 — The QP solver detects an infeasible QP problem. See "Monitoring Optimization Status to Detect Controller Failures" for an example where a large, sustained disturbance drives the output variable outside its specified bounds. In this case, the block holds its **mv** output at the most recent successful solution.

- **qp.status** = -2 — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem. In this case, the block holds its **mv** output at the most recent successful solution.

In a real-time application, you can use **qp.status** to set an alarm or take other special action.

Dependencies

To enable this port, select the **Optimization status** parameter.

est.state — Estimated controller states
vector

Estimated controller states at each control instant, returned as a vector signal. The estimated states include the plant, disturbance, and noise model states. If custom state estimation is used, this output signal has the same value as the **x[k|k]** input signal.

Dependencies

To enable this port, select the **Estimated controller states** parameter.

Optimal Sequences

mv.seq — Optimal manipulated variable sequence
matrix

Optimal manipulated variable sequence, returned as a matrix signal with p+1 rows and N_{mv} columns, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

The first p rows of **mv.seq** contain the calculated optimal manipulated variable values from current time k to time k+p-1. The first row of **mv.seq** contains the current manipulated variable values

(output **mv**). Since the controller does not calculate optimal control moves at time $k+p$, the final two rows of **mv.seq** are identical.

Dependencies

To enable this port, select the **Optimal control sequence** parameter.

x.seq — Optimal prediction model state sequence
matrix

Optimal prediction model state sequence, returned as a matrix signal with $p+1$ rows and N_x columns, where p is the prediction horizon and N_x is the number of states.

The first row of **x.seq** contains the current estimated state values, either from the built-in state estimator or from the custom state estimation block input **x[k|k]**. The next p rows of **x.seq** contain the calculated optimal state values from time $k+1$ to time $k+p$.

Dependencies

To enable this port, select the **Optimal state sequence** parameter.

y.seq — Optimal output variable sequence
matrix

Optimal output variable sequence, returned as a matrix signal with $p+1$ rows and N_y columns, where p is the prediction horizon and N_y is the number of output variables.

The first p rows of **y.seq** contain the calculated optimal output values from current time k to time $k+p-1$. The first row of **y.seq** is computed based on the current estimated states and the current measured disturbances (first row of input **md**). Since the controller does not calculate optimal output values at time $k+p$, the final two rows of **y.seq** are identical.

Dependencies

To enable this port, select the **Optimal output sequence** parameter.

Parameters

MPC Controller — Controller object
mpc object name

Specify an mpc object that defines an implicit MPC controller by entering the name of an mpc object from the MATLAB workspace.

Programmatic Use
Block Parameter: mpcobj
Type: string, character vector
Default: " "

Initial Controller State — Initial state
mpcstate object name

Specify the initial controller state. If you leave this parameter blank, the block uses the nominal values defined in the Model.Nominal property of the mpc object. To override the default, create an mpcstate object in your workspace, and enter its name in the field.

Use this parameter to make sure that the controller states reflect the true plant environment at the start of your simulation, to the best of your knowledge. This initial states can differ from the nominal states defined in the `mpc` object.

If custom state estimation is enabled, the block ignores the **Initial Controller State** parameter.

Programmatic Use
Block Parameter: `x0`
Type: string, character vector
Default: `""`

Design — Interactively design controller
button

To interactively modify the controller specified using the **MPC Controller** parameter, open the **MPC Designer** app by clicking **Design**. For example, you can:

- Import a new prediction model.
- Change horizons, constraints, and weights.
- Evaluate MPC performance with a linear plant.
- Export the updated controller to the MATLAB workspace.

If you have an existing `mpc` object in the MATLAB workspace, specify the name of that object using the **MPC Controller** parameter.

If you do not have an existing `mpc` object in the MATLAB workspace, leave the **MPC Controller** parameter empty. With the MPC Controller block connected to the plant, open **MPC Designer** by clicking **Design**. Using the app, linearize the Simulink model at a specified operating point, and design your controller. To use this design approach, you must have Simulink Control Design software. For more information, see "Design MPC Controller in Simulink" and "Linearize Simulink Models Using MPC Designer".

Review — Review controller for stability and robustness issues
button

Once you specify a controller using the **MPC Controller** parameter, you can review your design for run-time stability and robustness issues by clicking **Review**. For more information, see "Review Model Predictive Controller for Stability and Robustness Issues".

General Tab

Measured disturbance — Add measured disturbance input port
on (default) | off

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: `md_inport`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"on"`

External manipulated variable — Add external manipulated variable input port
off (default) | on

Select this parameter to add the **ext.mv** input port to the block.

Programmatic Use
Block Parameter: mv_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Targets for manipulated variables — Add manipulated variable target input port
off (default) | on

Select this parameter to add the **mv.target** input port to the block.

Programmatic Use
Block Parameter: uref_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal cost — Add optimal cost output port
off (default) | on

Select this parameter to add the **cost** output port to the block.

Programmatic Use
Block Parameter: return_cost
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimization status — Add optimization status output port
off (default) | on

Select this parameter to add the **qp.status** output port to the block.

Programmatic Use
Block Parameter: return_qpstatus
Type: string, character vector
Values: "off", "on"
Default: "off"

Estimated controller states — Add estimated states output port
off (default) | on

Select this parameter to add the **est.state** output port to the block.

Programmatic Use
Block Parameter: return_state
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal control sequence — Add optimal control sequence output port
off (default) | on

Select this parameter to add the **mv.seq** output port to the block.

Programmatic Use
Block Parameter: `return_mvseq`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Optimal state sequence — Add optimal state sequence output port
off (default) | on

Select this parameter to add the **x.seq** output port to the block.

Programmatic Use
Block Parameter: `return_xseq`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Optimal output sequence — Add optimal output sequence output port
off (default) | on

Select this parameter to add the **y.seq** output port to the block.

Programmatic Use
Block Parameter: `return_ovseq`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Use custom state estimation instead of using the built-in Kalman filter — Use custom state
estimate input port
off (default) | on

Select this parameter to remove the **mo** input port and add the **x[k|k]** input port.

Programmatic Use
Block Parameter: `state_inport`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Online Features Tab

Lower OV limits — Add minimum OV constraint input port
off (default) | on

Select this parameter to add the **ymin** input port to the block.

Programmatic Use
Block Parameter: `ymin_inport`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Upper OV limits — Add maximum OV constraint input port
off (default) | on

Select this parameter to add the **ymax** input port to the block.

Programmatic Use
Block Parameter: ymax_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower MV limits — Add minimum MV constraint input port
off (default) | on

Select this parameter to add the **umin** input port to the block.

Programmatic Use
Block Parameter: umin_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MV limits — Add maximum MV constraint input port
off (default) | on

Select this parameter to add the **umax** input port to the block.

Programmatic Use
Block Parameter: umax_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Custom constraints — Add custom constraints input ports
off (default) | on

Select this parameter to add the **E**, **F**, **G**, and **S** input ports to the block.

Programmatic Use
Block Parameter: cc_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

OV weights — Add OV tuning weights input port
off (default) | on

Select this parameter to add the **y.wt** input port to the block.

Programmatic Use
Block Parameter: ywt_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

MV weights — Add MV tuning weights input port
off (default) | on

Select this parameter to add the **u.wt** input port to the block.

Programmatic Use
Block Parameter: uwt_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

MVRate weights — Add MV rate tuning weights input port
off (default) | on

Select this parameter to add the **du.wt** input port to the block.

Programmatic Use
Block Parameter: duwt_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Slack variable weight — Add ECR tuning weight input port
off (default) | on

Select this parameter to add the **ecr.wt** input port to the block.

Programmatic Use
Block Parameter: rhoeps_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Adjust prediction horizon and control horizon at run time — Add horizon input ports
off (default) | on

Select this parameter to add the **p** and **m** input port to the block.

Programmatic Use
Block Parameter: pm_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Maximum prediction horizon — Add horizon input ports
10 (default) | positive integer

Select this parameter to add the **p** and **m** input port to the block.

Dependencies

To enable this parameter, select the **Adjust prediction horizon and control horizon at run time** parameter.

Programmatic Use
Block Parameter: MaximumP
Type: string, character vector
Default: "10"

Default Conditions Tab

Sample time — Default block sample time

1 (default) | positive scalar

Default block sample time for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a sample time that is compatible with your Simulink model design.

Dependencies

This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

Programmatic Use
Block Parameter: n_ts
Type: string, character vector
Default: "1"

Prediction horizon — Default prediction horizon

10 (default) | positive integer

Default prediction horizon for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a prediction horizon that is compatible with your Simulink model design.

Dependencies

This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

Programmatic Use
Block Parameter: n_p
Type: string, character vector
Default: "10"

Number of manipulated variables — Default number of manipulated variables

1 (default) | positive integer

Default number of manipulated variables for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a value that is compatible with your Simulink model design.

Dependencies

This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

Programmatic Use
Block Parameter: n_mv
Type: string, character vector
Default: "1"

Number of measured disturbances — Default number of measured disturbances

1 (default) | nonnegative integer

Default number of measured disturbances for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a value that is compatible with your Simulink model design.

Dependencies

- This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

- To use this parameter, you must select the **Measured disturbance** parameter.

Programmatic Use
Block Parameter: n_md
Type: string, character vector
Default: "1"

Number of unmeasured disturbances — Default number of unmeasured disturbances
0 (default) | nonnegative integer

Default number of unmeasured disturbances for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a value that is compatible with your Simulink model design.

Dependencies

This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

Programmatic Use
Block Parameter: n_ud
Type: string, character vector
Default: "0"

Number of measured outputs — Default number of measured outputs
1 (default) | positive integer

Default number of measured outputs for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a value that is compatible with your Simulink model design.

Dependencies

This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

Programmatic Use
Block Parameter: n_mo
Type: string, character vector
Default: "1"

Number of unmeasured outputs — Default number of unmeasured outputs
0 (default) | nonnegative integer

Default number of unmeasured outputs for performing simulation, trimming, or linearization using the **MPC Designer** app. You must specify a value that is compatible with your Simulink model design.

Dependencies

This parameter applies only when the **MPC Controller** parameter is empty and you open **MPC Designer** using the **Design** button.

Programmatic Use
Block Parameter: n_uo
Type: string, character vector
Default: "0"

Others Tab

Block data type — Specify data type of manipulated variables
double (default) | single | data type expression

Specify the block data type of the manipulated variables as one of the following:

- double — Double-precision floating point
- single — Single-precision floating point

  If you are implementing the block on a single-precision target, specify the output data type as single.
- data type expression — An expression that evaluates to either double or single. For more information, see "Control Data Types of Signals" (Simulink).

Programmatic Use
Block Parameter: BlockDataType
Type: string, character vector
Values: "double", "single", data type expression
Default: "double"

Inherit sample time — Inherit block sample time from parent subsystem
off (default) | on

Select this parameter to inherit the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see "Using MPC Controller Block Inside Function-Call and Triggered Subsystems".

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results for two reasons.

- The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last MV value occurred in the past at the sample time specified in the MPC object, and when the block is executed with a different sample rate, this assumption no longer holds.
- The built-in Kalman estimator uses the sample time specified in the MPC object to provide an estimation of the current state to the MPC optimization problem, so when the block is executed with a different sample time, the estimated state is no longer correct.

If you clear this parameter (default), the sample time of the block is inherited from the controller object.

To view the sample time of a block, in the Simulink model window, on the **Debug** tab, under **Information Overlays**, select either **colors** or **Text**. For more information, see "View Sample Time Information" (Simulink).

Programmatic Use
Block Parameter: SampleTimeInherited
Type: string, character vector
Values: "off", "on"
Default: "off"

Use external signal to enable or disable optimization — Add switch input port
off (default) | on

Select this parameter to add the **switch** input port to the block.

Programmatic Use
Block Parameter: switch_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Version History
Introduced before R2006a

R2018b: MPC Simulink block mv.seq output port signal dimensions have changed
Behavior changed in R2018b

The signal dimensions of the mv.seq output port of the MPC Controller block have changed. Previously, this signal was a p-by-N_{mv} matrix, where p is the prediction horizon and N_{mv} is the number of manipulated variables. Now, mv.seq is a $(p+1)$-by-N_{mv} matrix, where row $p+1$ duplicates row p.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Apps
MPC Designer

Functions
mpcstate

Objects
mpc

Blocks
Multiple MPC Controllers | Adaptive MPC Controller

Topics
"Design MPC Controller in Simulink"
"Simulation and Code Generation Using Simulink Coder"
"Simulation and Structured Text Generation Using Simulink PLC Coder"
"MPC Prediction Models"

Multiple MPC Controllers

Simulate switching between multiple implicit MPC controllers
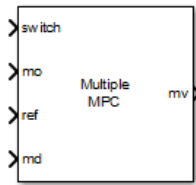
Libraries:
Model Predictive Control Toolbox

Description

At each control instant the Multiple MPC Controllers block receives the current measured plant output, reference, and measured plant disturbance (if any). In addition, it receives a switching signal that selects the *active controller* from a list of candidate MPC controllers designed at different operating points within the operating range. The active controller then solves a quadratic program to determine the optimal plant manipulated variables for the current input signals.

The Multiple MPC Controllers block enables you to achieve better control when operating conditions change. Using available measurements, you can detect the current operating region at run time and choose the appropriate active controller via the `switch` input port. Switching controllers for different operating regions is a common approach to solving nonlinear control problems using linear control techniques.

To improve efficiency, inactive controllers do not compute optimal control moves. However, to provide bumpless transfer between controllers, the inactive controllers continue to perform state estimation.

The Multiple MPC Controllers block lacks several optional features found in the MPC Controller block, as follows:

- You cannot disable optimization. One controller must always be active.

- You cannot initiate a controller design from within the block dialog box; that is, there is no **Design** button. Design all candidate controllers before configuring the Multiple MPC Controllers block.

- Similarly, there is no **Review** button. Instead, use the `review` command or the **MPC Designer** app.

- You cannot update custom constraints on linear combinations of inputs and outputs at run time.

Both the Multiple MPC Controllers block and the Adaptive MPC Controller block enable your control system to adapt to changing operating conditions at run time. The following table lists the advantages of using each block.

Block	Adaptive MPC Controller	Multiple MPC Controllers
Adaptation approach	Update prediction model for a single controller as operating conditions change	Switch between multiple controllers designed for different operating regions

Block	Adaptive MPC Controller	Multiple MPC Controllers
Advantages	• Only need to design a single controller offline • Less run-time computational effort and smaller memory footprint • More robust to real-life changes in plant conditions	• No need for online estimation of plant model • Controllers can have different sample time, horizons, and weights • Prediction models can have different orders or time domains • Finite set of candidate controllers can be tested thoroughly

Ports

Input

Required Inputs

ref — Model output reference values
row vector | matrix

Plant output reference values, specified as a row vector signal or matrix signal.

To use the same reference values across the prediction horizon, connect **ref** to a row vector signal with N_Y elements, where N_y is the number of output variables. Each element specifies the reference for an output variable.

To vary the references over the prediction horizon (previewing) from time $k+1$ to time $k+p$, connect **ref** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the references for one prediction horizon step. If you specify fewer than p rows, the final references are used for the remaining steps of the prediction horizon.

switch — Controller selection
integer

Use the **switch** input port to select the active controller. The **switch** input signal must be a scalar integer from 1 to N_c, where N_c is the number of specified candidate controllers. At each control instant, this signal designates the active controller. A switch value of 1 corresponds to the first entry in the cell array of candidate controllers, a value of 2 corresponds to the second controller, and so on.

If the switch signal is outside of the range 1 to N_c, the block retains the previous controller output.

mo — Measured output
vector

Measured output signals, specified as a vector signal. The candidate controllers use the measured plant outputs to improve their state estimates.

All candidate controllers must use the same state estimation option, either default or custom. If your candidate controllers use default state estimation, you must connect the measured plant outputs to the **mo** input port. If your candidate controllers use custom state estimation, you must connect the estimated plant state signal to the **x[k|k]** input port.

Dependencies

To enable this port, clear the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

x[k|k] — Custom state estimate
vector

Custom state estimate, specified as a vector signal. The candidate controllers use the connected state estimates instead of estimating the states using the built-in estimator. Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

All candidate controllers must use the same state estimation option, either default or custom. If your candidate controllers use custom state estimation, you must connect current state estimates to the **x[k|k]** input port. If your candidate controllers use default state estimation, you must connect the measured outputs to the **mo** input port.

When you use custom state estimation, all candidate controllers must have the same dimensions. All candidate controllers must use the same state definitions (number and order of states) for their respective plant, disturbance, and measurement noise models.

Dependencies

To enable this port, select the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

Additional Inputs

md — input
row vector | matrix

If your controller prediction model has measured disturbances you must enable this port and connect to it a row vector or matrix signal.

To use the same measured disturbance values across the prediction horizon, connect **md** to a row vector signal with N_{md} elements, where N_{md} is the number of manipulated variables. Each element specifies the value for a measured disturbance.

To vary the disturbances over the prediction horizon (previewing) from time k to time $k+p$, connect **md** to a matrix signal with N_{md} columns and up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbances for one prediction horizon step. If you specify fewer than $p+1$ rows, the final disturbances are used for the remaining steps of the prediction horizon.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

ext.mv — Control signals used in plant at previous control interval
vector

Control signals used in the plant at the previous control interval, specified as a vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables. All candidate controllers use this signal to update their controller state estimates at each control interval. This helps minimize bumpless transfer when the driving controller is switched. Using this input also improves state estimation

accuracy when the manipulated variables (MV) vector used in the plant differs from the MV vector calculated by the block, for example, due to signal saturation or an override condition.

Controller state estimation assumes that the MV vector is piecewise constant. Therefore, at time t_k, the **ext.mv** value must be the effective MV vector between times t_{k-1} and t_k. For example, if the MVs are actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

Note

- Connect **ext.mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the driving controller block, though this is not always the case. If the controller block is not driving the plant, then feeding the actual control signal to **ext.mv** can also help achieve bumpless transfer when the controller is switched back online.

- Using this option when the controller is driving the plant can cause an algebraic loop in the Simulink model, since there is direct feedthrough from the **ext.mv** input to the **mv** outport. To prevent such algebraic loops, insert a Memory block or Unit Delay block.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **External manipulated variable** parameter.

Online Constraints

ymin — Minimum output variable constraints
vector | matrix

To specify run-time minimum output variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `OutputVariables.Min` property of its `mpc` controller object. If an output variable has no lower bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **ymin** to a matrix signal with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one output variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `ith` column of the **ymin** signal corresponds to the `ith` plant output, and replaces the `OutputVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `OutputVariables(i).Min` in the `mpc` object (a constant bound for the ith plant output to be applied to all prediction steps)

ymin Dimension	Replacement Behavior
Scalar **ymin** (single output, constant bound)	**ymin** replaces the constant bound defined in `OutputVariables(i).Min`
Column vector **ymin** (single output, time-varying bound)	**ymin** replaces the constant bound defined in `OutputVariables(i).Min` with a time-varying bound.
Row vector **ymin** (multiple outputs, constant bounds)	The ith element of **ymin** replaces the constant bound defined in `OutputVariables(i).Min`
Matrix **ymin** (multiple outputs, time-varying bounds)	The ith column of **ymin** replaces the constant bound defined in `OutputVariables(i).Min` with a time-varying bound.

Vector `OutputVariables(i).Min` in the `mpc` object (a time-varying bound for the ith plant output with different values at different prediction steps)

ymin Dimension	Replacement Behavior
Scalar **ymin** (single output, constant bound)	**ymin** replaces the first finite entry in `OutputVariables.Min` and the remaining entries in `OutputVariables.Min` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables.Min` vector.
Column vector **ymin** (single output, time-varying bound)	**ymin** replaces the time-varying bound defined in `OutputVariables(i).Min`, and the original bound profile is discarded.
Row vector **ymin** (multiple outputs, constant bounds)	The ith element of **ymin** replaces the first finite entry in `OutputVariables(i).Min` and the remaining entries in `OutputVariables(i).Min` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables(i).Min` vector.
Matrix **ymin** (multiple outputs, time-varying bounds).	The ith column of **ymin** replaces the time-varying bound defined in `OutputVariables(i).Min`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Lower OV limits** parameter.

ymax — Maximum output variable constraints
vector | matrix

To specify run-time maximum output variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `OutputVariables.Max` property of its `mpc` controller object. If an output variable has no upper bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **ymax** to a matrix signal with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the

current time, and *p* is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than *p* rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one output variable, and a vector signal with no more than *p* entries is connected, then these entries are used across the prediction horizon.

The ith column of the **ymax** signal corresponds to the ith plant output, and replaces the OutputVariables(i).Max property of the mpc object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar OutputVariables(i).Max in the mpc object (a constant bound for the ith plant output to be applied to all prediction steps)

ymax Dimension	Replacement Behavior
Scalar **ymax** (single output, constant bound)	**ymax** replaces the constant bound defined in OutputVariables(i).Max
Column vector **ymax** (single output, time-varying bound)	**ymax** replaces the constant bound defined in OutputVariables(i).Max with a time-varying bound.
Row vector **ymax** (multiple outputs, constant bounds)	The ith element of **ymax** replaces the constant bound defined in OutputVariables(i).Max
Matrix **ymax** (multiple outputs, time-varying bounds)	The ith column of **ymax** replaces the constant bound defined in OutputVariables(i).Max with a time-varying bound.

Vector OutputVariables(i).Max in the mpc object (a time-varying bound for the ith plant output with different values at different prediction steps)

ymax Dimension	Replacement Behavior
Scalar **ymax** (single output, constant bound)	**ymax** replaces the first finite entry in OutputVariables.Max and the remaining entries in OutputVariables.Max shift up or down with the same amount of displacement to retain the profile defined by the original OutputVariables.Max vector.
Column vector **ymax** (single output, time-varying bound)	**ymax** replaces the time-varying bound defined in OutputVariables(i).Max, and the original bound profile is discarded.
Row vector **ymax** (multiple outputs, constant bounds)	The ith element of **ymax** replaces the first finite entry in OutputVariables(i).Max and the remaining entries in OutputVariables(i).Max shift up or down with the same amount of displacement to retain the profile defined by the original OutputVariables(i).Max vector.
Matrix **ymax** (multiple outputs, time-varying bounds).	The ith column of **ymax** replaces the time-varying bound defined in OutputVariables(i).Max, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Upper OV limits** parameter.

umin — Minimum manipulated variable constraints
vector | matrix

To specify run-time minimum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariables.Min` property of its `mpc` controller object. If a manipulated variable has no lower bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time k to time $k+p$-1, connect **umin** to a matrix signal with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one manipulated variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `i`th column of the **umin** signal corresponds to the `i`th manipulated variable, and replaces the `ManipulatedVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `ManipulatedVariables(i).Min` in the `mpc` object (a constant bound for the `i`th manipulated variable to be applied to all prediction steps)

umin Dimension	Replacement Behavior
Scalar **umin** (single output, constant bound)	**umin** replaces the constant bound defined in `ManipulatedVariables(i).Min`
Column vector **umin** (single output, time-varying bound)	**umin** replaces the constant bound defined in `ManipulatedVariables(i).Min` with a time-varying bound.
Row vector **umin** (multiple outputs, constant bounds)	The `i`th element of **umin** replaces the constant bound defined in `ManipulatedVariables(i).Min`
Matrix **umin** (multiple outputs, time-varying bounds)	The `i`th column of **umin** replaces the constant bound defined in `ManipulatedVariables(i).Min` with a time-varying bound.

Vector ManipulatedVariables(i).Min in the `mpc` object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

umin Dimension	Replacement Behavior
Scalar **umin** (single output, constant bound)	**umin** replaces the first finite entry in `ManipulatedVariables.Min` and the remaining entries in `ManipulatedVariables.Min` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables.Min` vector.
Column vector **umin** (single output, time-varying bound)	**umin** replaces the time-varying bound defined in `ManipulatedVariables(i).Min`, and the original bound profile is discarded.
Row vector **umin** (multiple outputs, constant bounds)	The ith component of **umin** replaces the first finite entry in `ManipulatedVariables(i).Min` and the remaining entries in `ManipulatedVariables(i).Min` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables(i).Min` vector.
Matrix **umin** (multiple outputs, time-varying bounds).	The ith column of **umin** replaces the time-varying bound defined in `ManipulatedVariables(i).Min`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Lower MV limits** parameter.

umax — Maximum manipulated variable constraints
vector | matrix

To specify run-time maximum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.Max` property of its `mpc` controller object. If a manipulated variable has no upper bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time k to time $k+p$-1, connect **umax** to a matrix signal with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one manipulated variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The ith column of the **umax** signal corresponds to the ith manipulated variable, and replaces the `ManipulatedVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `ManipulatedVariables(i).Max` in the `mpc` object (a constant bound for the ith manipulated variable to be applied to all prediction steps)

umax Dimension	Replacement Behavior
Scalar **umax** (single output, constant bound)	**umax** replaces the constant bound defined in `ManipulatedVariables(i).Max`
Column vector **umax** (single output, time-varying bound)	**umax** replaces the constant bound defined in `ManipulatedVariables(i).Max` with a time-varying bound.
Row vector **umax** (multiple outputs, constant bounds)	The `ith` element of **umax** replaces the constant bound defined in `ManipulatedVariables(i).Max`
Matrix **umax** (multiple outputs, time-varying bounds)	The `ith` column of **umax** replaces the constant bound defined in `ManipulatedVariables(i).Max` with a time-varying bound.

Vector `ManipulatedVariables(i).Max` in the `mpc` object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

umax Dimension	Replacement Behavior
Scalar **umax** (single output, constant bound)	**umax** replaces the first finite entry in `ManipulatedVariables.Max` and the remaining entries in `ManipulatedVariables.Max` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables.Max` vector.
Column vector **umax** (single output, time-varying bound)	**umax** replaces the time-varying bound defined in `ManipulatedVariables(i).Max`, and the original bound profile is discarded.
Row vector **umax** (multiple outputs, constant bounds)	The `ith` element of **umax** replaces the first finite entry in `ManipulatedVariables(i).Max` and the remaining entries in `ManipulatedVariables(i).Max` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables(i).Max` vector.
Matrix **umax** (multiple outputs, time-varying bounds).	The `ith` column of **umax** replaces the time-varying bound defined in `ManipulatedVariables(i).Max`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Upper MV limits** parameter.

Online Tuning Weights

y.wt — Output variable tuning weights
row vector | matrix

To specify run-time output variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.OutputVariables` property of its controller object. These tuning weights penalize deviations from output references.

If the MPC controller object uses constant output tuning weights over the prediction horizon, you can specify only constant output tuning weights at runtime. Similarly, if the MPC controller object uses output tuning weights that vary over the prediction horizon, you can specify only time-varying output tuning weights at runtime

To use constant tuning weights over the prediction horizon, connect **y.wt** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies a nonnegative tuning weight for an output variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, connect **y.wt** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **OV weights** parameter.

u.wt — Manipulated variable tuning weights
row vector | matrix

To specify run-time manipulated variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariables` property of its controller object. These tuning weights penalize deviations from MV targets.

If the MPC controller object uses constant manipulated variable tuning weights over the prediction horizon, you can specify only constant manipulated variable tuning weights at runtime. Similarly, if the MPC controller object uses manipulated variable tuning weights that vary over the prediction horizon, you can specify only time-varying manipulated variable tuning weights at runtime.

To use the same tuning weights over the prediction horizon, connect **u.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p-1$, connect **u.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MV weights** parameter.

du.wt — Manipulated variable rate tuning weights
row vector | matrix

To specify run-time manipulated variable rate tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariablesRate` property of its controller object. These tuning weights penalize large changes in control moves.

If the MPC controller object uses constant manipulated variable rate tuning weights over the prediction horizon, you can specify only constant manipulated variable tuning rate weights at runtime. Similarly, if the MPC controller object uses manipulated variable rate tuning weights that vary over the prediction horizon, you can specify only time-varying manipulated variable rate tuning weights at runtime.

To use the same tuning weights over the prediction horizon, connect **du.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable rate. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **du.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MVRate weights** parameter.

ecr.wt — Slack variable tuning weight
scalar

To specify a run-time slack variable tuning weight, enable this input port and connect a scalar signal. If this port is disabled, the block uses the tuning weight specified in the `Weights.ECR` property of its controller object.

The slack variable tuning weight has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero. If there are soft constraints, increasing the **ecr.wt** value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

Dependencies

To enable this port, select the **ECR weight** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, output as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables. The Multiple MPC Controllers block passes the output of the active controller to the **mv** output port.

If the solver of the active controller converges to a local optimum solution (**qp.status** is positive), then **mv** contains the optimal solution.

If the solver fails (**qp.status** is negative), then **mv** remains at its most recent successful solution; that is, the controller output freezes.

If the solver reaches the maximum number of iterations without finding an optimal solution (**qp.status** is zero) and the `Optimization.UseSuboptimalSolution` property of the active controller is:

- `true`, then **mv** contains the suboptimal solution
- `false`, then **mv** then **mv** remains at its most recent successful solution

Additional Outputs

cost — Objective function cost
nonnegative scalar

Objective function cost, output as a nonnegative scalar signal. The cost quantifies the degree to which the controller has achieved its objectives. The cost value is calculated using the scaled MPC cost function in which every term is offset-free and dimensionless.

The cost value is only meaningful when the **qp.status** output is nonnegative.

Dependencies

To enable this port, select the **Optimal cost** parameter.

qp.status — Optimization status
integer

Optimization status of the active controller, output as an integer signal.

If the active controller solves the QP problem for 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. Therefore, a large value means a relatively slow block execution for this time interval.

The QP solver can fail to find an optimal solution for the following reasons:

- **qp.status** = 0 — The QP solver cannot find a solution within the maximum number of iterations specified in the `mpc` object. In this case, if the `Optimizer.UseSuboptimalSolution` property of the active controller is `false`, the block holds its **mv** output at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.
- **qp.status** = -1 — The QP solver detects an infeasible QP problem. See "Monitoring Optimization Status to Detect Controller Failures" for an example where a large, sustained disturbance drives the output variable outside its specified bounds. In this case, the block holds its **mv** output at the most recent successful solution.
- **qp.status** = -2 — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem. In this case, the block holds its **mv** output at the most recent successful solution.

In a real-time application, you can use **qp.status** to set an alarm or take other special action.

Dependencies

To enable this port, select the **Optimization status** parameter.

est.state — Estimated controller states
vector

Estimated controller states of the active controller, output as a vector signal. The estimated states include the plant, disturbance, and noise model states.

Dependencies

To enable this port, select the **Estimated controller states** parameter.

Optimal Sequences

mv.seq — Optimal manipulated variable sequence
matrix

Optimal manipulated variable sequence, returned as a matrix signal with $p+1$ rows and N_{mv} columns, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

The first p rows of **mv.seq** contain the calculated optimal manipulated variable values from current time k to time $k+p$-1. The first row of **mv.seq** contains the current manipulated variable values (output **mv**). Since the controller does not calculate optimal control moves at time $k+p$, the final two rows of **mv.seq** are identical.

Dependencies

To enable this port, select the **Optimal control sequence** parameter.

x.seq — Optimal prediction model state sequence
matrix

Optimal prediction model state sequence, returned as a matrix signal with $p+1$ rows and N_x columns, where p is the prediction horizon and N_x is the number of states.

The first row of **x.seq** contains the current estimated state values, either from the built-in state estimator or from the custom state estimation block input **x[k|k]**. The next p rows of **x.seq** contain the calculated optimal state values from time $k+1$ to time $k+p$.

Dependencies

To enable this port, select the **Optimal state sequence** parameter.

y.seq — Optimal output variable sequence
matrix

Optimal output variable sequence, returned as a matrix signal with $p+1$ rows and N_y columns, where p is the prediction horizon and N_y is the number of output variables.

The first p rows of **y.seq** contain the calculated optimal output values from current time k to time $k+p$-1. The first row of **y.seq** is computed based on the current estimated states and the current measured disturbances (first row of input **md**). Since the controller does not calculate optimal output values at time $k+p$, the final two rows of **y.seq** are identical.

Dependencies

To enable this port, select the **Optimal output sequence** parameter.

Parameters

Cell Array of MPC Controllers — Candidate controllers
cell array of `mpc` objects | cell array of strings | cell array of character vectors

Candidate controllers, specified as one of the following:

- Cell array of `mpc` objects.
- Cell array of strings or a cell array of character vectors, where each element is the name of an `mpc` object in the MATLAB workspace.

The specified array must contain at least two candidate controllers. The first entry in the cell array is the controller that corresponds to a switch input value of 1, the second corresponds to a switch input value of 2, and so on.

Programmatic Use
Block Parameter: `mpcobjs`
Type: string, character vector, cell array of strings, cell array of character vectors
Default: `""`

Cell Array of Initial Controller States — Initial state
cell array of `mpcstate` objects | cell array of strings | cell array of character vectors

Initial states for the candidate controllers, specified as one of the following:

- Cell array of `mpcstate` objects.
- Cell array of strings or a cell array of character vectors, where each element is the name of an `mpcstate` object in the MATLAB workspace.
- `{[],[],...}`, `{'[]','[]',...}`, or `{"[]","[]",...}` — Use the nominal condition defined in `Model.Nominal` property of each candidate controller as its initial state.

Use this parameter make the controller states reflect the true plant environment at the start of your simulation to the best of your knowledge. This initial states can differ from the nominal states defined in the `mpc` objects.

If custom state estimation is enabled, the block ignores **Cell Array of Initial Controller States** parameter.

Programmatic Use
Block Parameter: `x0s`
Type: string, character vector, cell array of strings, cell array of character vectors
Default: `""`

General Tab

Measured disturbances — Add measured disturbance input port
on (default) | off

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: `md_inport_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"on"`

External manipulated variable — Add external manipulated variable input port
off (default) | on

Select this parameter to add the **ext.mv** input port to the block.

Programmatic Use
Block Parameter: mv_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Targets for manipulated variables — Add manipulated variable target input port
off (default) | on

Select this parameter to add the **mv.target** input port to the block.

Programmatic Use
Block Parameter: uref_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal cost — Add optimal cost output port
off (default) | on

Select this parameter to add the **cost** output port to the block.

Programmatic Use
Block Parameter: return_cost_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimization status — Add optimization status output port
off (default) | on

Select this parameter to add the **qp.status** output port to the block.

Programmatic Use
Block Parameter: return_qpstatus_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Estimated controller states — Add estimated states output port
off (default) | on

Select this parameter to add the **est.state** output port to the block.

Programmatic Use
Block Parameter: return_state_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal control sequence — Add optimal control sequence output port
off (default) | on

Select this parameter to add the **mv.seq** output port to the block.

Programmatic Use
Block Parameter: `return_mvseq_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Optimal state sequence — Add optimal state sequence output port
off (default) | on

Select this parameter to add the **x.seq** output port to the block.

Programmatic Use
Block Parameter: `return_xseq_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Optimal output sequence — Add optimal output sequence output port
off (default) | on

Select this parameter to add the **y.seq** output port to the block.

Programmatic Use
Block Parameter: `return_ovseq_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Use custom state estimation instead of using the built-in Kalman filter — Use custom state estimate input port
off (default) | on

Select this parameter to remove the **mo** input port and add the **x[k|k]** input port.

Programmatic Use
Block Parameter: `state_inport_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Online Features Tab

Lower OV limits — Add minimum OV constraint input port
off (default) | on

Select this parameter to add the **ymin** input port to the block.

Programmatic Use
Block Parameter: `ymin_inport_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Upper OV limits — Add maximum OV constraint input port
off (default) | on

Select this parameter to add the **ymax** input port to the block.

Programmatic Use
Block Parameter: ymax_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower MV limits — Add minimum MV constraint input port
off (default) | on

Select this parameter to add the **umin** input port to the block.

Programmatic Use
Block Parameter: umin_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MV limits — Add maximum MV constraint input port
off (default) | on

Select this parameter to add the **umax** input port to the block.

Programmatic Use
Block Parameter: umax_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Custom constraints — Add custom constraints input ports
off (default) | on

Select this parameter to add the **E**, **F**, **G**, and **S** input ports to the block.

Programmatic Use
Block Parameter: cc_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

OV weights — Add OV tuning weights input port
off (default) | on

Select this parameter to add the **y.wt** input port to the block.

Programmatic Use
Block Parameter: ywt_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

MV weights — Add MV tuning weights input port
off (default) | on

Select this parameter to add the **u.wt** input port to the block.

Programmatic Use
Block Parameter: uwt_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

MVRate weights — Add MV rate tuning weights input port
off (default) | on

Select this parameter to add the **du.wt** input port to the block.

Programmatic Use
Block Parameter: duwt_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Slack variable weight — Add ECR tuning weight input port
off (default) | on

Select this parameter to add the **ecr.wt** input port to the block.

Programmatic Use
Block Parameter: rhoeps_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Others Tab

Block data type — Specify data type of manipulated variables
double (default) | single | data type expression

Specify the block data type of the manipulated variables as one of the following:

- double — Double-precision floating point
- single — Single-precision floating point

   If you are implementing the block on a single-precision target, specify the output data type as single.

- data type expression — An expression that evaluates to either double or single. For more information see "Control Data Types of Signals" (Simulink).

Programmatic Use
Block Parameter: BlockDataType_multiple
Type: string, character vector
Values: "double", "single", data type expression
Default: "double"

Inherit sample time — Inherit block sample time from parent subsystem
off (default) | on

Select this parameter to inherit the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see "Using MPC Controller Block Inside Function-Call and Triggered Subsystems".

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results for two reasons.

- The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last MV value occurred in the past at the sample time specified in the MPC object, and when the block is executed with a different sample rate, this assumption no longer holds.

- The built-in Kalman estimator uses the sample time specified in the MPC object to provide an estimation of the current state to the MPC optimization problem, so when the block is executed with a different sample time, the estimated state is no longer correct.

If you clear this parameter (default), the sample time of the block is inherited from the controller object.

To view the sample time of a block, in the Simulink model window, on the **Debug** tab, under **Information Overlays**, select either **colors** or **Text**. For more information, see "View Sample Time Information" (Simulink).

Programmatic Use
Block Parameter: `SampleTimeInherited_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Version History

Introduced in R2008b

R2018b: MPC Simulink block `mv.seq` output port signal dimensions have changed
Behavior changed in R2018b

The signal dimensions of the `mv.seq` output port of the Multiple MPC Controllers block have changed. Previously, this signal was a p-by-N_{mv} matrix, where p is the prediction horizon and N_{mv} is the number of manipulated variables. Now, `mv.seq` is a $(p+1)$-by-N_{mv} matrix, where row $p+1$ duplicates row p.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Functions
`mpcmoveMultiple`

Objects
`mpc` | `mpcstate`

Blocks
MPC Controller | Multiple Explicit MPC Controllers

Topics
"Simulation and Code Generation Using Simulink Coder"
"Simulation and Structured Text Generation Using Simulink PLC Coder"
"Gain-Scheduled MPC"
"Design Workflow"

Explicit MPC Controller

Explicit model predictive controller
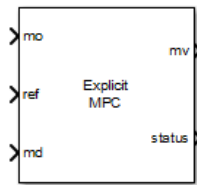
Libraries:
Model Predictive Control Toolbox

Description

The Explicit MPC Controller block uses the following input signals:

- Either measured plant outputs (`mo`) or custom state estimate (`x[k|k]`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any

The Explicit MPC Controller block uses a lookup table to store the precalculated piecewise-affine optimal control law instead of solving a quadratic programming optimization problem at runtime at each control interval like the MPC Controller block. Given the same MPC problem, the two blocks return the same solution. The Explicit MPC Controller block requires less online computational effort, which is useful for applications that need a short control interval. It has, however, a heavier offline computational effort and a larger memory footprint. Indeed, the combinatorial nature of explicit MPC restricts its usage to applications with relatively few inputs, outputs, and state variables, a short prediction horizon, and few output constraints.

The Explicit MPC Controller supports only a subset of optional MPC features, as outlined in the following table.

Supported Features	Unsupported Features
• Built-in (Kalman filter) and custom state estimation • Outport for state estimation results • External manipulated variable feedback signal inport • Single-precision block data (default is double precision) • Inherited sample time	• Online tuning (penalty weight adjustments) • Online constraint adjustments • Online manipulated variable target adjustments • Reference and/or measured disturbance previewing

Ports

Input

Required Inputs

mo — Measured outputs
vector

Measured outputs, specified as a vector signal. The block uses the measured plant outputs to improve its state estimates. If your controller uses default state estimation, you must connect the measured plant outputs to the **mo** input port. If your controller uses custom state estimation, you must connect the estimated plant states to the **x[k|k]** input port.

Dependencies

To enable this port, clear the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

x[k|k] — Custom state estimate
vector

Custom state estimate, specified as a vector signal. The block uses the connected state estimates instead of estimating the states using the built-in estimator. If your controller uses custom state estimation, you must connect the current state estimates to the **x[k|k]** input port. If your controller uses default state estimation, you must connect the measured output to the **mo** input port.

Even though noise model states (if any) are not used in MPC optimization, the custom state vector must contain all the states defined in the mpcstate object of the controller, including the plant, disturbance, and noise model states.

Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

Dependencies

To enable this port, select the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

ref — Model reference output
vector

At each control instant, the ref signal must contain the current reference values (targets or setpoints) for the n_y output variables, where n_y is the total number of outputs, including measured and unmeasured outputs. Since this block does not support reference previewing, ref must be a vector signal.

Additional Inputs

md — Measured disturbances
vector

If your controller prediction model has measured disturbances, you must enable this port and connect to it a row vector signal containing N_{md} elements, where N_{md} is the number of measured disturbances.

Since this block does not support measured disturbance previewing, md must be a vector signal.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

ext.mv — Control signals used in plant at previous control interval
vector

Control signals used in the plant at the previous control interval, specified as a vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables. Use this input port to improve state estimation accuracy when:

- You know your controller is not always in control of the plant.
- The actual MV signals applied to the plant can potentially differ from the values generated by the controller, such as in control signal saturation.

Controller state estimation assumes that the MVs are piecewise constant. Therefore, at time t_k, the **ext.mv** value must contain the effective MVs between times t_{k-1} and t_k. For example, if the MVs are actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

Note

- Connect **ext.mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode (that is, the controller output is not driving the plant), then feeding the actual control signal to **ext.mv** can help achieve bumpless transfer when the controller is switched back online.
- When the controller is driving the plant, insert a Memory block or Unit Delay block to feed back the MV signal applied to the plant at the previous control interval. This also avoids a direct feedthrough from the **ext.mv** inport to the **mv** outport, therefore preventing algebraic loops in the Simulink model.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **External manipulated variable** parameter.

switch — Disable evaluation
scalar

To turn off the controller evaluation, connect **switch** to a nonzero signal.

Disabling controller evaluation reduces computational effort when the controller output is not needed, such as when the system is operating manually or another controller has taken over. However, the controller continues to update its internal state estimates in the usual way. Therefore, it is ready to resume evaluations whenever the **switch** signal returns to zero. While controller evaluation is off, the block passes the current **ext.mv** signal to the controller output. If the **ext.mv** inport is not enabled, the controller output is held at the value it had when evaluation was disabled.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **Use external signal to enable controller evaluation** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, returned as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

The controller updates its **mv** output at each control instant using the control law contained in the explicit MPC controller object. If the control law evaluation is not successful, mv is unchanged (that is, it is held at the previous successful result) and the `status` output, if present, becomes either 0 or –1.

Additional Outputs

status — Status of piecewise affine function evaluation
1 | 0 | -1

This output indicates whether the latest explicit MPC control-law evaluation succeeded. The outport provides a scalar signal that has one of the following values:

- 1 — Successful explicit control law evaluation
- 0 — Failure due to one or more control law parameters out of range
- –1 — Control law parameters were within the valid range but an extrapolation was necessary

If **status** is either 0 or –1, the **mv** outport remains at the last known good value.

Dependencies

To enable this port, select the **Status of piecewise affine function evaluation** parameter.

region — Region number of evaluated piecewise affine function
nonnegative integer

This output provides the index of the polyhedral region used in the latest explicit control law evaluation. If the control law evaluation fails, the signal at this outport is zero.

Dependencies

To enable this port, select the **Region number of evaluated piecewise affine function** parameter.

est.state — Estimated controller states
vector

Estimated controller states at each control instant, returned as a vector signal. The estimated states include the plant, disturbance, and noise model states. If custom state estimation is used, this output signal has the same value as the **x[k|k]** input signal.

Dependencies

To enable this port, select the **Estimated controller states** parameter.

Parameters

Explicit MPC Controller — Explicit controller object
explicitMPC object name

An explicitMPC object containing the control law to be used. It must exist in the MATLAB workspace. Use the generateExplicitMPC command to create this object.

Programmatic Use
Block Parameter: empcobj
Type: string, character vector
Default: ""

Initial Controller State — Initial state
mpcstate object name

An optional mpcstate object specifying the initial controller state. If you leave this parameter blank, the block uses the nominal values defined in the Model.Nominal property of the explicitMPC object. To override the default values, create an mpcstate object in your workspace, and enter its name in the field. You can use this parameter to make the controller states reflect the true plant environment at the start of your simulation to the best of your knowledge.

If custom state estimation is enabled, the block ignores the **Initial Controller State** parameter.

Programmatic Use
Block Parameter: x0
Type: string, character vector
Default: ""

General Tab

Measured disturbance — Add measured disturbance input port
on (default) | off

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: md_inport
Type: string, character vector
Values: "off", "on"
Default: "on"

External manipulated variable — Add external manipulated variable input port
off (default) | on

Select this parameter to add the **ext.mv** input port to the block.

Programmatic Use
Block Parameter: mv_inport
Type: string, character vector

Values: "off", "on"
Default: "off"

Status of piecewise affine function evaluation — Add evaluation status output port
off (default) | on

Select this parameter to add the **status** output port to the block.

Programmatic Use
Block Parameter: return_status
Type: string, character vector
Values: "off", "on"
Default: "on"

Region number of evaluated piecewise affine function — Add region number output port
off (default) | on

Select this parameter to add the **region** output port to the block.

Programmatic Use
Block Parameter: return_region
Type: string, character vector
Values: "off", "on"
Default: "off"

Estimated controller states — Add estimated states output port
off (default) | on

Select this parameter to add the **est.state** output port to the block.

Programmatic Use
Block Parameter: return_state
Type: string, character vector
Values: "off", "on"
Default: "off"

Use custom state estimation instead of using the built-in Kalman filter — Use custom state estimate input port
off (default) | on

Select this parameter to remove the **mo** input port and add the **x[k|k]** input port.

Programmatic Use
Block Parameter: state_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Others Tab

Block data type — Specify data type of manipulated variables
double (default) | single | data type expression

Specify the block data type of the manipulated variables as one of the following:

- double — Double-precision floating point

- `single` — Single-precision floating point

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

- `data type expression` — An expression that evaluates to either `double` or `single`. For more information, see "Control Data Types of Signals" (Simulink).

Programmatic Use
Block Parameter: `BlockDataType`
Type: string, character vector
Values: `"double"`, `"single"`, `data type expression`
Default: `"double"`

Inherit sample time — Inherit block sample time from parent subsystem
`off` (default) | `on`

Select this parameter to inherit the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see "Using MPC Controller Block Inside Function-Call and Triggered Subsystems".

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results for two reasons.

- The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last MV value occurred in the past at the sample time specified in the MPC object, and when the block is executed with a different sample rate, this assumption no longer holds.

- The built-in Kalman estimator uses the sample time specified in the MPC object to provide an estimation of the current state to the MPC optimization problem, so when the block is executed with a different sample time, the estimated state is no longer correct.

If you clear this parameter (default), the sample time of the block is inherited from the controller object.

To view the sample time of a block, in the Simulink model window, on the **Debug** tab, under **Information Overlays**, select either **colors** or **Text**. For more information, see "View Sample Time Information" (Simulink).

Programmatic Use
Block Parameter: `SampleTimeInherited`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Use external signal to enable controller evaluation — Add a switch-off input port
`off` (default) | `on`

Select this parameter to add the **switch** input port to the block. Whenever a nonzero signal is fed to the switch input port, the controller evaluation is turned off. See the **switch** input port for more details.

Programmatic Use
Block Parameter: switch_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Version History
Introduced in R2014b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Functions
generateExplicitMPC | mpcmoveExplicit

Objects
mpc | mpcstate

Blocks
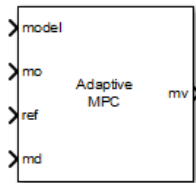MPC Controller | Multiple Explicit MPC Controllers

Topics
"Explicit MPC Control of a Single-Input-Single-Output Plant"
"Explicit MPC"
"Design Workflow for Explicit MPC"

Adaptive MPC Controller

Simulate adaptive and time-varying model predictive controllers

Libraries:
Model Predictive Control Toolbox

Description

The Adaptive MPC Controller block uses the following input signals:

- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any

In addition, the required `model` input signal specifies the prediction model to use when computing the optimal plant manipulated variables `mv`. The linear prediction model can change at each control interval in response to changes in the real plant at run time. The prediction model can represent a single LTI plant used for all prediction steps (adaptive MPC mode) or an array of LTI plants for different prediction steps (time-varying MPC mode). Two common ways to modify this model are as follows:

- Given a nonlinear plant model, linearize it at the current operating point.
- Use plant data to estimate parameters in an empirical linear-time-varying (LTV) model.

By default, the block estimates its prediction model states. Since the prediction model parameters change at run time, the static Kalman filter used in the MPC Controller block is inappropriate. Instead, the Adaptive MPC Controller block uses a linear-time-varying Kalman filter (LTVKF). For more information, see "Adaptive MPC".

In all other ways, the Adaptive MPC Controller block mimics the MPC Controller block. Since the adaptive version involves additional overhead, use the MPC Controller block unless you need to control a nonlinear plant across a wide range of operating conditions where plant dynamics vary significantly.

Both the Adaptive MPC Controller block and the Multiple MPC Controllers block enable your control system to adapt to changing operating conditions at run time. The following table lists the advantages of using each block.

Block	Adaptive MPC Controller	Multiple MPC Controllers
Adaptation approach	Update prediction model for a single controller as operating conditions change	Switch between multiple controllers designed for different operating regions

Block	Adaptive MPC Controller	Multiple MPC Controllers
Advantages	• Only need to design a single controller offline • Less run-time computational effort and smaller memory footprint • More robust to real-life changes in plant conditions	• No need for online estimation of plant model • Controllers can have different sample time, horizons, and weights • Prediction models can have different orders or time domains • Finite set of candidate controllers can be tested thoroughly

Ports

Input

Required Inputs

model — Updated plant model and nominal operating point
bus signal

Updated plant model and nominal operating point, specified as a bus signal. a bus signal to the `model` inport. At the beginning of each control interval, this signal modifies the controller object `Model.Plant` and `Model.Nominal` properties.

The Adaptive MPC Controller requires the plant model to be an LTI discrete-time state-space object with no delays. The following command extracts the state-space matrices comprising such a model.

`[A,B,C,D] = ssdata(mpcobj.Model.Plant)`

The purpose of the `model` input is to replace these matrices with new ones having the same dimensions and representing the same control interval. You must also retain the sequence in which the input, output, and state variables appear in the `Model.Plant` property of the controller.

When operating in:

- Adaptive MPC mode, the bus you connect to the `model` inport must contain the following signals, each identified by the specified name:

  - A — n_x-by-n_x matrix signal, where n_x is the number of plant model states.
  - B — n_x-by-n_u matrix signal, where n_u is the total number of plant model inputs (i.e., manipulated variables, measured disturbances, and unmeasured disturbances).
  - C — n_y-by-n_x matrix signal, where n_y is the number of plant model outputs.
  - D — n_y-by-n_u matrix signal.
  - X — Vector signal of length n_x, replacing the controller `Model.Nominal.X` property.
  - Y — Vector signal of length n_y, replacing the controller `Model.Nominal.Y` property.
  - U — Vector signal of length n_u, replacing the controller `Model.Nominal.U` property.
  - DX — Vector signal of length n_x, replacing the controller `Model.Nominal.DX` property. It must be appropriate for use with a discrete-time model of the assumed control interval. For more information, see "Adaptive MPC".

To compute DX values, use the discrete-time state update function (*f*) for your model. Here, u_k and x_k are the respective input and state values for the current time step.

$$DX = f(u_k, x_k) - x_k$$

- Time-varying MPC mode, the bus you connect to the `model` inport must contain the following 3–dimensional bus signals:

  - A — n_x-by-n_x-by-$(p+1)$ matrix signal
  - B — n_x-by-n_u-by-$(p+1)$ matrix signal
  - C — n_y-by-n_x-by-$(p+1)$
  - D — n_y-by-n_u-by-$(p+1)$ matrix signal
  - X — n_x-by-$(p+1)$ matrix signal
  - Y — n_y-by-$(p+1)$ matrix signal
  - U — n_u-by-$(p+1)$ matrix signal
  - DX — n_x-by-$(p+1)$ matrix signal

  Here, *p* is the controller prediction horizon. For each signal, specify *p*+1 values representing the model and nominal conditions at each step of the prediction horizon. For more information, see "Time-Varying MPC".

One way to form the bus is to use a Bus Creator block.

Dependencies

The dimensions of the bus elements in **model** depend on the operating mode of the controller. To place the controller in:

- Adaptive MPC mode, clear the **Linear Time-Varying (LTV) plants** parameter
- Time-varying MPC mode, select the **Linear Time-Varying (LTV) plants** parameter

ref — Model output reference values
row vector | matrix

Plant output reference values, specified as a row vector signal or matrix signal.

To use the same reference values across the prediction horizon, connect **ref** to a row vector signal with N_Y elements, where N_y is the number of output variables. Each element specifies the reference for an output variable.

To vary the references over the prediction horizon (previewing) from time *k*+1 to time *k*+*p*, connect **ref** to a matrix signal with N_y columns and up to *p* rows. Here, *k* is the current time and *p* is the prediction horizon. Each row contains the references for one prediction horizon step. If you specify fewer than *p* rows, the final references are used for the remaining steps of the prediction horizon.

mo — Measured outputs
vector

Measured outputs, specified as a vector signal. The block uses the measured plant outputs to improve its state estimates. If your controller uses default state estimation, you must connect the measured plant outputs to the **mo** input port. If your controller uses custom state estimation, you must connect the estimated plant states to the **x[k|k]** input port.

Dependencies

To enable this port, clear the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

x[k|k] — Custom state estimate
vector

Custom state estimate, specified as a vector signal. The block uses the connected state estimates instead of estimating the states using the built-in estimator. If your controller uses custom state estimation, you must connect the current state estimates to the **x[k|k]** input port. If your controller uses default state estimation, you must connect the measured output to the **mo** input port.

Even though noise model states (if any) are not used in MPC optimization, the custom state vector must contain all the states defined in the mpcstate object of the controller, including the plant, disturbance, and noise model states.

Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

Dependencies

To enable this port, select the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

Additional Inputs

md — input
row vector | matrix

If your controller prediction model has measured disturbances you must enable this port and connect to it a row vector or matrix signal.

To use the same measured disturbance values across the prediction horizon, connect **md** to a row vector signal with N_{md} elements, where N_{md} is the number of manipulated variables. Each element specifies the value for a measured disturbance.

To vary the disturbances over the prediction horizon (previewing) from time k to time $k+p$, connect **md** to a matrix signal with N_{md} columns and up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbances for one prediction horizon step. If you specify fewer than $p+1$ rows, the final disturbances are used for the remaining steps of the prediction horizon.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

ext.mv — Control signals used in plant at previous control interval
vector

Control signals used in the plant at the previous control interval, specified as a vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables. Use this input port to improve state estimation accuracy when:

• You know your controller is not always in control of the plant.

- The actual MV signals applied to the plant can potentially differ from the values generated by the controller, such as in control signal saturation.

Controller state estimation assumes that the MVs are piecewise constant. Therefore, at time t_k, the **ext.mv** value must contain the effective MVs between times t_{k-1} and t_k. For example, if the MVs are actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

Note

- Connect **ext.mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode (that is, the controller output is not driving the plant), then feeding the actual control signal to **ext.mv** can help achieve bumpless transfer when the controller is switched back online.
- When the controller is driving the plant, insert a Memory block or Unit Delay block to feed back the MV signal applied to the plant at the previous control interval. This also avoids a direct feedthrough from the **ext.mv** inport to the **mv** outport, therefore preventing algebraic loops in the Simulink model.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **External manipulated variable** parameter.

switch — Enable or disable optimization
scalar

To turn off the controller optimization calculations, connect **switch** to a nonzero signal.

Disabling optimization calculations reduces computational effort when the controller output is not needed, such as when the system is operating manually or another controller has taken over. However, the controller continues to update its internal state estimates in the usual way. Therefore, it is ready to resume optimization calculations whenever the **switch** signal returns to zero. While controller optimization is off, the block passes the current **ext.mv** signal to the controller output. If the **ext.mv** inport is not enabled, the controller output is held at the value it had when optimization was disabled.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

mv.target — Manipulated variable targets
vector

To specify manipulated variable targets, enable this input port, and connect a vector signal. To make a given manipulated variable track its specified target value, you must also specify a nonzero tuning weight for that manipulated variable.

The supplied **mv.target** values at run-time apply across the prediction horizon.

Dependencies

To enable this port, select the **Targets for manipulated variables** parameter.

Online Constraints

ymin — Minimum output variable constraints
vector | matrix

To specify run-time minimum output variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `OutputVariables.Min` property of its `mpc` controller object. If an output variable has no lower bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **ymin** to a matrix signal with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one output variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `ith` column of the **ymin** signal corresponds to the `ith` plant output, and replaces the `OutputVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `OutputVariables(i).Min` in the `mpc` object (a constant bound for the `ith` plant output to be applied to all prediction steps)

ymin Dimension	Replacement Behavior
Scalar **ymin** (single output, constant bound)	**ymin** replaces the constant bound defined in `OutputVariables(i).Min`
Column vector **ymin** (single output, time-varying bound)	**ymin** replaces the constant bound defined in `OutputVariables(i).Min` with a time-varying bound.
Row vector **ymin** (multiple outputs, constant bounds)	The `ith` element of **ymin** replaces the constant bound defined in `OutputVariables(i).Min`
Matrix **ymin** (multiple outputs, time-varying bounds)	The `ith` column of **ymin** replaces the constant bound defined in `OutputVariables(i).Min` with a time-varying bound.

Vector `OutputVariables(i).Min` in the `mpc` object (a time-varying bound for the `ith` plant output with different values at different prediction steps)

ymin Dimension	Replacement Behavior
Scalar **ymin** (single output, constant bound)	**ymin** replaces the first finite entry in `OutputVariables.Min` and the remaining entries in `OutputVariables.Min` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables.Min` vector.
Column vector **ymin** (single output, time-varying bound)	**ymin** replaces the time-varying bound defined in `OutputVariables(i).Min`, and the original bound profile is discarded.
Row vector **ymin** (multiple outputs, constant bounds)	The `ith` element of **ymin** replaces the first finite entry in `OutputVariables(i).Min` and the remaining entries in `OutputVariables(i).Min` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables(i).Min` vector.
Matrix **ymin** (multiple outputs, time-varying bounds).	The `ith` column of **ymin** replaces the time-varying bound defined in `OutputVariables(i).Min`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Lower OV limits** parameter.

ymax — Maximum output variable constraints
vector | matrix

To specify run-time maximum output variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `OutputVariables.Max` property of its `mpc` controller object. If an output variable has no upper bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **ymax** to a matrix signal with N_y columns and up to p rows. Here, N_y is the number of plant outputs, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one output variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `ith` column of the **ymax** signal corresponds to the `ith` plant output, and replaces the `OutputVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `OutputVariables(i).Max` in the `mpc` object (a constant bound for the ith plant output to be applied to all prediction steps)

ymax Dimension	Replacement Behavior
Scalar **ymax** (single output, constant bound)	**ymax** replaces the constant bound defined in `OutputVariables(i).Max`
Column vector **ymax** (single output, time-varying bound)	**ymax** replaces the constant bound defined in `OutputVariables(i).Max` with a time-varying bound.
Row vector **ymax** (multiple outputs, constant bounds)	The `i`th element of **ymax** replaces the constant bound defined in `OutputVariables(i).Max`
Matrix **ymax** (multiple outputs, time-varying bounds)	The `i`th column of **ymax** replaces the constant bound defined in `OutputVariables(i).Max` with a time-varying bound.

Vector `OutputVariables(i).Max` in the `mpc` object (a time-varying bound for the ith plant output with different values at different prediction steps)

ymax Dimension	Replacement Behavior
Scalar **ymax** (single output, constant bound)	**ymax** replaces the first finite entry in `OutputVariables.Max` and the remaining entries in `OutputVariables.Max` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables.Max` vector.
Column vector **ymax** (single output, time-varying bound)	**ymax** replaces the time-varying bound defined in `OutputVariables(i).Max`, and the original bound profile is discarded.
Row vector **ymax** (multiple outputs, constant bounds)	The `i`th element of **ymax** replaces the first finite entry in `OutputVariables(i).Max` and the remaining entries in `OutputVariables(i).Max` shift up or down with the same amount of displacement to retain the profile defined by the original `OutputVariables(i).Max` vector.
Matrix **ymax** (multiple outputs, time-varying bounds).	The `i`th column of **ymax** replaces the time-varying bound defined in `OutputVariables(i).Max`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Upper OV limits** parameter.

umin — Minimum manipulated variable constraints
vector | matrix

To specify run-time minimum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariables.Min` property of its `mpc` controller object. If a manipulated variable has no lower bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time k to time $k+p$-1, connect **umin** to a matrix signal with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k

is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one manipulated variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The ith column of the **umin** signal corresponds to the ith manipulated variable, and replaces the `ManipulatedVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `ManipulatedVariables(i).Min` in the `mpc` object (a constant bound for the ith manipulated variable to be applied to all prediction steps)

umin Dimension	Replacement Behavior
Scalar **umin** (single output, constant bound)	**umin** replaces the constant bound defined in `ManipulatedVariables(i).Min`
Column vector **umin** (single output, time-varying bound)	**umin** replaces the constant bound defined in `ManipulatedVariables(i).Min` with a time-varying bound.
Row vector **umin** (multiple outputs, constant bounds)	The ith element of **umin** replaces the constant bound defined in `ManipulatedVariables(i).Min`
Matrix **umin** (multiple outputs, time-varying bounds)	The ith column of **umin** replaces the constant bound defined in `ManipulatedVariables(i).Min` with a time-varying bound.

Vector `ManipulatedVariables(i).Min` in the `mpc` object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

umin Dimension	Replacement Behavior
Scalar **umin** (single output, constant bound)	**umin** replaces the first finite entry in `ManipulatedVariables.Min` and the remaining entries in `ManipulatedVariables.Min` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables.Min` vector.
Column vector **umin** (single output, time-varying bound)	**umin** replaces the time-varying bound defined in `ManipulatedVariables(i).Min`, and the original bound profile is discarded.
Row vector **umin** (multiple outputs, constant bounds)	The ith component of **umin** replaces the first finite entry in `ManipulatedVariables(i).Min` and the remaining entries in `ManipulatedVariables(i).Min` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables(i).Min` vector.
Matrix **umin** (multiple outputs, time-varying bounds).	The ith column of **umin** replaces the time-varying bound defined in `ManipulatedVariables(i).Min`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Lower MV limits** parameter.

umax — Maximum manipulated variable constraints
vector | matrix

To specify run-time maximum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.Max` property of its `mpc` controller object. If a manipulated variable has no upper bound specified in the controller object, then at run time the block ignores the corresponding connected signal.

To change the bounds over the prediction horizon from time k to time $k+p$-1, connect **umax** to a matrix signal with N_{mv} columns and up to p rows. Here, N_{mv} is the number of manipulated variables, k is the current time, and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon. If there is only one manipulated variable, and a vector signal with no more than p entries is connected, then these entries are used across the prediction horizon.

The `ith` column of the **umax** signal corresponds to the `ith` manipulated variable, and replaces the `ManipulatedVariables(i).Max` property of the `mpc` object at run time. The replacement behavior depends on the dimensions of both variables.

Scalar `ManipulatedVariables(i).Max` in the `mpc` object (a constant bound for the `ith` manipulated variable to be applied to all prediction steps)

umax Dimension	Replacement Behavior
Scalar **umax** (single output, constant bound)	**umax** replaces the constant bound defined in `ManipulatedVariables(i).Max`
Column vector **umax** (single output, time-varying bound)	**umax** replaces the constant bound defined in `ManipulatedVariables(i).Max` with a time-varying bound.
Row vector **umax** (multiple outputs, constant bounds)	The `ith` element of **umax** replaces the constant bound defined in `ManipulatedVariables(i).Max`
Matrix **umax** (multiple outputs, time-varying bounds)	The `ith` column of **umax** replaces the constant bound defined in `ManipulatedVariables(i).Max` with a time-varying bound.

Vector ManipulatedVariables(i).Max in the mpc object (a time-varying bound for the ith manipulated variable with different values at different prediction steps)

umax Dimension	Replacement Behavior
Scalar **umax** (single output, constant bound)	**umax** replaces the first finite entry in `ManipulatedVariables.Max` and the remaining entries in `ManipulatedVariables.Max` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables.Max` vector.
Column vector **umax** (single output, time-varying bound)	**umax** replaces the time-varying bound defined in `ManipulatedVariables(i).Max`, and the original bound profile is discarded.
Row vector **umax** (multiple outputs, constant bounds)	The ith element of **umax** replaces the first finite entry in `ManipulatedVariables(i).Max` and the remaining entries in `ManipulatedVariables(i).Max` shift up or down with the same amount of displacement to retain the profile defined by the original `ManipulatedVariables(i).Max` vector.
Matrix **umax** (multiple outputs, time-varying bounds).	The ith column of **umax** replaces the time-varying bound defined in `ManipulatedVariables(i).Max`, and the original bound profile is discarded.

Dependencies

To enable this port, select the **Upper MV limits** parameter.

E — Manipulated variable constraint matrix
matrix

Manipulated variable constraint matrix, specified as an N_c-by-N_{mv} matrix signal, where N_c is the number of mixed input/output constraints and N_{mv} is the number of manipulated variables.

If you define E in the mpc object, you must connect a signal to the **E** input port. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **E** input port along with the **F**, **G**, and **S** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the E matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter.

F — Controlled output constraint matrix
matrix

Controlled output constraint matrix, specified as an N_c-by-N_y matrix signal, where N_c is the number of mixed input/output constraints and N_y is the number of plant outputs. If you define F in the mpc

object, you must connect a signal to the **F** input port with same number of rows. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **F** input port along with the **E**, **G**, and **S** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the F matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter.

G — Custom constraint vector
row vector

Custom constraint vector, specified as a row vector signal of length N_c, where N_c is the number of mixed input/output constraints. If you define G in the `mpc` object, you must connect a signal to the **G** input port with same number of rows. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **G** input port along with the **E**, **F**, and **S** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the G matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter.

S — Measured disturbance constraint matrix
matrix

Measured disturbance constraint matrix, specified as an N_c-by-n_N matrix signal, where N_c is the number of mixed input/output constraints, and N_v is the number of measured disturbances. If you define S in the `mpc` object, you must connect a signal to the **S** input port with same number of rows. Otherwise, connect a zero matrix with the correct size.

To specify run-time mixed input/output constraints, use the **S** input port along with the **E**, **F**, and **G** ports. These constraints replace the mixed input/output constraints previously set using `setconstraint`. For more information on mixed input/output constraints, see "Constraints on Linear Combinations of Inputs and Outputs".

The number of mixed input/output constraints cannot change at run time. Therefore, N_c must match the number of rows in the G matrix you specified using `setconstraint`.

Dependencies

To enable this port, select the **Custom constraints** parameter. This port is added only if the `mpc` object has measured disturbances.

Online Tuning Weights

y.wt — Output variable tuning weights
row vector | matrix

To specify run-time output variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.OutputVariables` property of its controller object. These tuning weights penalize deviations from output references.

If the MPC controller object uses constant output tuning weights over the prediction horizon, you can specify only constant output tuning weights at runtime. Similarly, if the MPC controller object uses output tuning weights that vary over the prediction horizon, you can specify only time-varying output tuning weights at runtime

To use constant tuning weights over the prediction horizon, connect **y.wt** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies a nonnegative tuning weight for an output variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, connect **y.wt** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **OV weights** parameter.

u.wt — Manipulated variable tuning weights
row vector | matrix

To specify run-time manipulated variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariables` property of its controller object. These tuning weights penalize deviations from MV targets.

If the MPC controller object uses constant manipulated variable tuning weights over the prediction horizon, you can specify only constant manipulated variable tuning weights at runtime. Similarly, if the MPC controller object uses manipulated variable tuning weights that vary over the prediction horizon, you can specify only time-varying manipulated variable tuning weights at runtime.

To use the same tuning weights over the prediction horizon, connect **u.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **u.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MV weights** parameter.

du.wt — Manipulated variable rate tuning weights
row vector | matrix

To specify run-time manipulated variable rate tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariablesRate` property of its controller object. These tuning weights penalize large changes in control moves.

If the MPC controller object uses constant manipulated variable rate tuning weights over the prediction horizon, you can specify only constant manipulated variable tuning rate weights at runtime. Similarly, if the MPC controller object uses manipulated variable rate tuning weights that vary over the prediction horizon, you can specify only time-varying manipulated variable rate tuning weights at runtime.

To use the same tuning weights over the prediction horizon, connect **du.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable rate. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **du.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MVRate weights** parameter.

ecr.wt — Slack variable tuning weight
scalar

To specify a run-time slack variable tuning weight, enable this input port and connect a scalar signal. If this port is disabled, the block uses the tuning weight specified in the `Weights.ECR` property of its controller object.

The slack variable tuning weight has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero. If there are soft constraints, increasing the **ecr.wt** value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

Dependencies

To enable this port, select the **ECR weight** parameter.

Online Horizons

p — Prediction horizon
positive integer

Prediction horizon, specified as positive integer signal. The prediction horizon signal value must be less than or equal to the **Maximum prediction horizon** parameter.

At run time, the values of p overrides the default prediction horizon specified in the controller object. For more information, see "Adjust Horizons at Run Time".

Dependencies

To enable this port, select the **Adjust prediction horizon and control horizon at run time** parameter.

m — Control horizon
positive integer | vector

Control horizon, specified as one of the following:

- Positive integer signal less than or equal to the prediction horizon.
- Vector signal of positive integers specifying blocking interval lengths. For more information, see "Manipulated Variable Blocking".

At run time, the values of m overrides the default control horizon specified in the controller object. For more information, see "Adjust Horizons at Run Time".

Dependencies

To enable this port, select the **Adjust prediction horizon and control horizon at run time** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, output as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the solver converges to a local optimum solution (**qp.status** is positive), then **mv** contains the optimal solution.

If the solver fails (**qp.status** is negative), then **mv** remains at its most recent successful solution; that is, the controller output freezes.

If the solver reaches the maximum number of iterations without finding an optimal solution (**qp.status** is zero) and the `Optimization.UseSuboptimalSolution` property of the controller is:

- `true`, then **mv** contains the suboptimal solution
- `false`, then **mv** then **mv** remains at its most recent successful solution

Additional Outputs

cost — Objective function cost
nonnegative scalar

Objective function cost, output as a nonnegative scalar signal. The cost quantifies the degree to which the controller has achieved its objectives. The cost value is calculated using the scaled MPC cost function in which every term is offset-free and dimensionless.

The cost value is only meaningful when the **qp.status** output is nonnegative.

Dependencies

To enable this port, select the **Optimal cost** parameter.

qp.status — Optimization status
integer

Optimization status, output as an integer signal.

If the controller solves the QP problem for 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. Therefore, a large value means a relatively slow block execution for this time interval.

The QP solver can fail to find an optimal solution for the following reasons:

- **qp.status** = 0 — The QP solver cannot find a solution within the maximum number of iterations specified in the mpc object. In this case, if the Optimizer.UseSuboptimalSolution property of the controller is false, the block holds its **mv** output at the most recent successful solution. Otherwise, it uses the suboptimal solution found during the last solver iteration.

- **qp.status** = -1 — The QP solver detects an infeasible QP problem. See "Monitoring Optimization Status to Detect Controller Failures" for an example where a large, sustained disturbance drives the output variable outside its specified bounds. In this case, the block holds its **mv** output at the most recent successful solution.

- **qp.status** = -2 — The QP solver has encountered numerical difficulties in solving a severely ill-conditioned QP problem. In this case, the block holds its **mv** output at the most recent successful solution.

In a real-time application, you can use **qp.status** to set an alarm or take other special action.

Dependencies

To enable this port, select the **Optimization status** parameter.

est.state — Estimated controller states
vector

Estimated controller states at each control instant, returned as a vector signal. The estimated states include the plant, disturbance, and noise model states. If custom state estimation is used, this output signal has the same value as the **x[k|k]** input signal.

Dependencies

To enable this port, select the **Estimated controller states** parameter.

Optimal Sequences

mv.seq — Optimal manipulated variable sequence
matrix

Optimal manipulated variable sequence, returned as a matrix signal with $p+1$ rows and N_{mv} columns, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

The first p rows of **mv.seq** contain the calculated optimal manipulated variable values from current time k to time $k+p-1$. The first row of **mv.seq** contains the current manipulated variable values

(output **mv**). Since the controller does not calculate optimal control moves at time $k+p$, the final two rows of **mv.seq** are identical.

Dependencies

To enable this port, select the **Optimal control sequence** parameter.

x.seq — Optimal prediction model state sequence
matrix

Optimal prediction model state sequence, returned as a matrix signal with $p+1$ rows and N_x columns, where p is the prediction horizon and N_x is the number of states.

The first row of **x.seq** contains the current estimated state values, either from the built-in state estimator or from the custom state estimation block input **x[k|k]**. The next p rows of **x.seq** contain the calculated optimal state values from time $k+1$ to time $k+p$.

Dependencies

To enable this port, select the **Optimal state sequence** parameter.

y.seq — Optimal output variable sequence
matrix

Optimal output variable sequence, returned as a matrix signal with $p+1$ rows and N_y columns, where p is the prediction horizon and N_y is the number of output variables.

The first p rows of **y.seq** contain the calculated optimal output values from current time k to time $k+p$-1. The first row of **y.seq** is computed based on the current estimated states and the current measured disturbances (first row of input **md**). Since the controller does not calculate optimal output values at time $k+p$, the final two rows of **y.seq** are identical.

Dependencies

To enable this port, select the **Optimal output sequence** parameter.

Parameters

Adaptive MPC Controller — Controller object
mpc object name

Specify an mpc object that defines an MPC controller by entering the name of an mpc object designed at the nominal operating point of the block. At run time, the controller replaces the original prediction model (A, B, C, and D) and nominal values (U, Y, X, and DX) with the data specified in the **model** input port at each control instant.

By default, the block assumes all other controller object properties (for example tuning weights, constraints) are constant. You can override this assumption using the options in the **Online Features** section.

The following restrictions apply to the mpc controller object:

- It must exist in the MATLAB workspace.
- Its prediction model must be an LTI discrete-time, state-space object with no delays. Use the absorbDelay command to convert delays to discrete states. The dimensions of the A, B, C, and D matrices in the prediction determine the dimensions required by the model inport signal.

Programmatic Use
Block Parameter: mpcobj
Type: string, character vector
Default: ""

Initial Controller State — Initial state
mpcstate object name

Specify the initial controller state. If you leave this parameter blank, the block uses the nominal values defined in the Model.Nominal property of the mpc object. To override the default, create an mpcstate object in your workspace, and enter its name in the field.

Use this parameter to make sure that the controller states reflect the true plant environment at the start of your simulation, to the best of your knowledge. This initial states can differ from the nominal states defined in the mpc object.

If custom state estimation is enabled, the block ignores the **Initial Controller State** parameter.

Programmatic Use
Block Parameter: x0
Type: string, character vector
Default: ""

General Tab

Measured disturbance — Add measured disturbance input port
on (default) | off

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: md_inport
Type: string, character vector
Values: "off", "on"
Default: "on"

External manipulated variable — Add external manipulated variable input port
off (default) | on

Select this parameter to add the **ext.mv** input port to the block.

Programmatic Use
Block Parameter: mv_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Targets for manipulated variables — Add manipulated variable target input port
off (default) | on

Select this parameter to add the **mv.target** input port to the block.

Programmatic Use
Block Parameter: uref_inport
Type: string, character vector

Values: "off", "on"
Default: "off"

Optimal cost — Add optimal cost output port
off (default) | on

Select this parameter to add the **cost** output port to the block.

Programmatic Use
Block Parameter: return_cost
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimization status — Add optimization status output port
off (default) | on

Select this parameter to add the **qp.status** output port to the block.

Programmatic Use
Block Parameter: return_qpstatus
Type: string, character vector
Values: "off", "on"
Default: "off"

Estimated controller states — Add estimated states output port
off (default) | on

Select this parameter to add the **est.state** output port to the block.

Programmatic Use
Block Parameter: return_state
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal control sequence — Add optimal control sequence output port
off (default) | on

Select this parameter to add the **mv.seq** output port to the block.

Programmatic Use
Block Parameter: return_mvseq
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal state sequence — Add optimal state sequence output port
off (default) | on

Select this parameter to add the **x.seq** output port to the block.

Programmatic Use
Block Parameter: return_xseq
Type: string, character vector
Values: "off", "on"

Default: `"off"`

Optimal output sequence — Add optimal output sequence output port
off (default) | on

Select this parameter to add the **y.seq** output port to the block.

Programmatic Use
Block Parameter: `return_ovseq`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Use custom state estimation instead of using the built-in Kalman filter — Use custom state estimate input port
off (default) | on

Select this parameter to remove the **mo** input port and add the **x[k|k]** input port.

Programmatic Use
Block Parameter: `state_inport`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Online Features Tab

Linear Time-Varying (LTV) plants — Use custom state estimate input port
off (default) | on

To operate your controller in time-varying MPC mode, select this option. When operating in this mode, connect a 3–dimensional bus signal to the **model** input port

For an example, see "Time-Varying MPC Control of a Time-Varying Plant".

Programmatic Use
Block Parameter: `isltv_plant`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Lower OV limits — Add minimum OV constraint input port
off (default) | on

Select this parameter to add the **ymin** input port to the block.

Programmatic Use
Block Parameter: `ymin_inport`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Upper OV limits — Add maximum OV constraint input port
off (default) | on

Select this parameter to add the **ymax** input port to the block.

Programmatic Use
Block Parameter: ymax_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower MV limits — Add minimum MV constraint input port
off (default) | on

Select this parameter to add the **umin** input port to the block.

Programmatic Use
Block Parameter: umin_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MV limits — Add maximum MV constraint input port
off (default) | on

Select this parameter to add the **umax** input port to the block.

Programmatic Use
Block Parameter: umax_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Custom constraints — Add custom constraints input ports
off (default) | on

Select this parameter to add the **E**, **F**, **G**, and **S** input ports to the block.

Programmatic Use
Block Parameter: cc_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

OV weights — Add OV tuning weights input port
off (default) | on

Select this parameter to add the **y.wt** input port to the block.

Programmatic Use
Block Parameter: ywt_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

MV weights — Add MV tuning weights input port
off (default) | on

Select this parameter to add the **u.wt** input port to the block.

4-75

Programmatic Use
Block Parameter: uwt_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

MVRate weights — Add MV rate tuning weights input port
off (default) | on

Select this parameter to add the **du.wt** input port to the block.

Programmatic Use
Block Parameter: duwt_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Slack variable weight — Add ECR tuning weight input port
off (default) | on

Select this parameter to add the **ecr.wt** input port to the block.

Programmatic Use
Block Parameter: rhoeps_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Adjust prediction horizon and control horizon at run time — Add horizon input ports
off (default) | on

Select this parameter to add the **p** and **m** input port to the block.

Programmatic Use
Block Parameter: pm_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Maximum prediction horizon — Add horizon input ports
10 (default) | positive integer

Select this parameter to add the **p** and **m** input port to the block.

Dependencies

To enable this parameter, select the **Adjust prediction horizon and control horizon at run time** parameter.

Programmatic Use
Block Parameter: MaximumP
Type: string, character vector
Default: "10"

Others Tab

Inherit sample time — Inherit block sample time from parent subsystem
off (default) | on

Select this parameter to inherit the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see "Using MPC Controller Block Inside Function-Call and Triggered Subsystems".

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results for two reasons.

- The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last MV value occurred in the past at the sample time specified in the MPC object, and when the block is executed with a different sample rate, this assumption no longer holds.

- The built-in Kalman estimator uses the sample time specified in the MPC object to provide an estimation of the current state to the MPC optimization problem, so when the block is executed with a different sample time, the estimated state is no longer correct.

If you clear this parameter (default), the sample time of the block is inherited from the controller object.

To view the sample time of a block, in the Simulink model window, on the **Debug** tab, under **Information Overlays**, select either **colors** or **Text**. For more information, see "View Sample Time Information" (Simulink).

Programmatic Use
Block Parameter: SampleTimeInherited
Type: string, character vector
Values: "off", "on"
Default: "off"

Use external signal to enable or disable optimization — Add switch input port
off (default) | on

Select this parameter to add the **switch** input port to the block.

Programmatic Use
Block Parameter: switch_inport
Type: string, character vector
Values: "off", "on"
Default: "off"

Version History
Introduced in R2014b

R2018b: MPC Simulink block mv.seq output port signal dimensions have changed
Behavior changed in R2018b

The signal dimensions of the `mv.seq` output port of the Adaptive MPC Controller block have changed. Previously, this signal was a p-by-N_{mv} matrix, where p is the prediction horizon and N_{mv} is the number of manipulated variables. Now, `mv.seq` is a $(p+1)$-by-N_{mv} matrix, where row $p+1$ duplicates row p.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Functions
`mpcmoveAdaptive`

Objects
`mpc` | `mpcstate`

Blocks
MPC Controller | Multiple MPC Controllers

Topics
"Simulation and Code Generation Using Simulink Coder"
"Simulation and Structured Text Generation Using Simulink PLC Coder"
"Adaptive MPC"
"Time-Varying MPC"

Multiple Explicit MPC Controllers

Multiple explicit MPC controllers
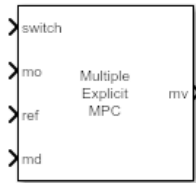
Libraries:
Model Predictive Control Toolbox

Description

The Multiple Explicit MPC Controllers block uses the following input signals:

- Measured plant outputs (`mo`)
- Reference or setpoint (`ref`)
- Measured plant disturbance (`md`), if any
- Switching signal (`switch`)

The Multiple Explicit MPC Controllers block enables you to transition between multiple explicit MPC controllers in real time based on the current operating conditions. Typically, you design each controller for a particular region of the operating space. Using available measurements, you detect the current operating region and select the appropriate active controller using the `switch` inport.

The switching signal selects the *active controller* among a list of two or more candidate explicit MPC controllers. These controllers reduce online computational effort by using a table-lookup control law during each control interval instead of solving a quadratic programming problem. For more information, see Explicit MPC Controller.

To improve efficiency, inactive controllers do not evaluate their control law. However, to provide bumpless transfer between controllers, the inactive controllers continue to perform state estimation.

Like for the Multiple MPC Controllers block, you cannot disable evaluation for the Multiple Explicit MPC Controllers block. One controller must always be active.

Like the Explicit MPC Controller block, the Multiple Explicit MPC Controllers block supports only a subset of optional MPC features, as outlined in the following table.

Supported Features	Unsupported Features
• Built-in (Kalman filter) and custom state estimation • Outport for state estimation results • External manipulated variable feedback signal inport • Single-precision block data (default is double precision) • Inherited sample time	• Online tuning (penalty weight adjustments) • Online constraint adjustments • Online manipulated variable target adjustments • Reference and/or measured disturbance previewing

Ports

Input

Required Inputs

switch — Controller selection
integer

Use the **switch** input port to select the active controller. The **switch** input signal must be a scalar integer from 1 to N_c, where N_c is the number of specified candidate controllers. At each control instant, this signal designates the active controller. A switch value of 1 corresponds to the first entry in the cell array of candidate controllers, a value of 2 corresponds to the second controller, and so on.

If the switch signal is outside of the range 1 to N_c, the block retains the previous controller output.

mo — Measured output
vector

Measured output signals, specified as a vector signal. The candidate controllers use the measured plant outputs to improve their state estimates.

All candidate controllers must use the same state estimation option, either default or custom. If your candidate controllers use default state estimation, you must connect the measured plant outputs to the **mo** input port. If your candidate controllers use custom state estimation, you must connect the estimated plant state signal to the **x[k|k]** input port.

Dependencies

To enable this port, clear the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

x[k|k] — Custom state estimate
vector

Custom state estimate, specified as a vector signal. The candidate controllers use the connected state estimates instead of estimating the states using the built-in estimator. Use custom state estimates when an alternative estimation technique is considered superior to the built-in estimator or when the states are fully measurable.

All candidate controllers must use the same state estimation option, either default or custom. If your candidate controllers use custom state estimation, you must connect current state estimates to the

x[k|k] input port. If your candidate controllers use default state estimation, you must connect the measured outputs to the **mo** input port.

When you use custom state estimation, all candidate controllers must have the same dimensions. All candidate controllers must use the same state definitions (number and order of states) for their respective plant, disturbance, and measurement noise models.

Dependencies

To enable this port, select the **Use custom state estimation instead of using the built-in Kalman filter** parameter.

ref — Model reference output
vector

At each control instant, the `ref` signal must contain the current reference values (targets or setpoints) for the n_y output variables, where n_y is the total number of outputs, including measured and unmeasured outputs. Since this block does not support reference previewing, `ref` must be a vector signal.

Additional Inputs

md — Measured disturbances
vector

If your controller prediction model has measured disturbances, you must enable this port and connect to it a row vector signal containing N_{md} elements, where N_{md} is the number of measured disturbances.

Since this block does not support measured disturbance previewing, `md` must be a vector signal.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

ext.mv — Control signals used in plant at previous control interval
vector

Control signals used in the plant at the previous control interval, specified as a vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables. All candidate controllers use this signal to update their controller state estimates at each control interval. This helps minimize bumpless transfer when the driving controller is switched. Using this input also improves state estimation accuracy when the manipulated variables (MV) vector used in the plant differs from the MV vector calculated by the block, for example, due to signal saturation or an override condition.

Controller state estimation assumes that the MV vector is piecewise constant. Therefore, at time t_k, the **ext.mv** value must be the effective MV vector between times t_{k-1} and t_k. For example, if the MVs are actually varying over this interval, you might supply the time-averaged value evaluated at time t_k.

Note

- Connect **ext.mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the driving controller block, though this is not always the case. If the controller block is not driving the plant, then feeding the actual control

signal to **ext.mv** can also help achieve bumpless transfer when the controller is switched back online.

- Using this option when the controller is driving the plant can cause an algebraic loop in the Simulink model, since there is direct feedthrough from the **ext.mv** input to the **mv** outport. To prevent such algebraic loops, insert a Memory block or Unit Delay block.

For an example that uses the external manipulated variable input port for bumpless transfer, see "Switch Controller Online and Offline with Bumpless Transfer".

Dependencies

To enable this port, select the **External manipulated variable** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, returned as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

The Multiple Explicit MPC Controllers block passes the output of the active controller to the **mv** output. Therefore, the active controller updates the **mv** output at each control instant using the control law contained in its explicit MPC controller object. If the control law evaluation fails, this signal is unchanged; that is, it is held at the previous successful result.

Additional Outputs

status — Status of piecewise affine function evaluation
1 | 0 | -1

This output indicates whether the latest explicit MPC control-law evaluation succeeded. The outport provides a scalar signal that has one of the following values:

- 1 — Successful explicit control law evaluation
- 0 — Failure due to one or more control law parameters out of range
- –1 — Control law parameters were within the valid range but an extrapolation was necessary

If **status** is either 0 or –1, the **mv** outport remains at the last known good value.

Dependencies

To enable this port, select the **Status of piecewise affine function evaluation** parameter.

region — Region number of evaluated piecewise affine function
nonnegative integer

This output provides the index of the polyhedral region used in the latest explicit control law evaluation. If the control law evaluation fails, the signal at this outport is zero.

Dependencies

To enable this port, select the **Region number of evaluated piecewise affine function** parameter.

est.state — Estimated controller states
vector

Estimated controller states at each control instant, returned as a vector signal. The estimated states include the plant, disturbance, and noise model states. If custom state estimation is used, this output signal has the same value as the **x[k|k]** input signal.

Dependencies

To enable this port, select the **Estimated controller states** parameter.

Parameters

Cell Array of Explicit MPC Controllers — Candidate controllers
cell array of `explicitMPC` objects | cell array of strings | cell array of character vectors

Candidate controllers, specified as one of the following:

- Cell array of `explicitMPC` objects

- Cell array of strings or a cell array of character vectors, where each element is the name of an `explicitMPC` object in the MATLAB workspace

The specified array must contain at least two candidate controllers. The first entry in the cell array is the controller that corresponds to a switch input value of 1, the second corresponds to a switch input value of 2, and so on.

Programmatic Use
Block Parameter: `empcobjs`
Type: string, character vector, cell array of strings, cell array of character vectors
Default: `""`

Cell Array of Initial Controller States — Initial state
cell array of `mpcstate` objects | cell array of strings | cell array of character vectors

Initial states for the candidate controllers, specified as one of the following:

- Cell array of `mpcstate` objects.

- Cell array of strings or a cell array of character vectors, where each element is the name of an `mpcstate` object in the MATLAB workspace.

- `{[],[],...}`, `{'[]','[]',...}`, or `{"[]","[]",...}` — Use the nominal condition defined in `Model.Nominal` property of each candidate controller as its initial state.

If you leave this parameter blank, the block uses the nominal values defined in the `Model.Nominal` property of the `explicitMPC` objects. You can use this parameter to make the controller states reflect the true plant environment at the start of your simulation to the best of your knowledge.

If custom state estimation is enabled, the block ignores **Cell Array of Initial Controller States** parameter.

Programmatic Use
Block Parameter: `x0s`
Type: string, character vector, cell array of strings, cell array of character vectors
Default: `""`

General Tab

Measured disturbances — Add measured disturbance input port
on (default) | off

If your controller has measured disturbances, you must select this parameter to add the **md** output
port to the block.

Programmatic Use
Block Parameter: md_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "on"

External manipulated variable — Add external manipulated variable input port
off (default) | on

Select this parameter to add the **ext.mv** input port to the block.

Programmatic Use
Block Parameter: mv_inport_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Status of piecewise affine function evaluation — Add evaluation status output port
off (default) | on

Select this parameter to add the **status** output port to the block.

Programmatic Use
Block Parameter: return_status_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Region number of evaluated piecewise affine function — Add region number output port
off (default) | on

Select this parameter to add the **region** output port to the block.

Programmatic Use
Block Parameter: return_region_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Estimated controller states — Add estimated states output port
off (default) | on

Select this parameter to add the **est.state** output port to the block.

Programmatic Use
Block Parameter: return_state_multiple
Type: string, character vector
Values: "off", "on"

Default: `"off"`

Use custom state estimation instead of using the built-in Kalman filter — Use custom state estimate input port
`off` (default) | `on`

Select this parameter to remove the **mo** input port and add the **x[k|k]** input port.

Programmatic Use
Block Parameter: `state_inport_multiple`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Others Tab

Block data type — Specify data type of manipulated variables
`double` (default) | `single` | `data type expression`

Specify the block data type of the manipulated variables as one of the following:

- `double` — Double-precision floating point
- `single` — Single-precision floating point

  If you are implementing the block on a single-precision target, specify the output data type as `single`.

- `data type expression` — An expression that evaluates to either `double` or `single`. For more information see "Control Data Types of Signals" (Simulink).

Programmatic Use
Block Parameter: `BlockDataType_multiple`
Type: string, character vector
Values: `"double"`, `"single"`, `data type expression`
Default: `"double"`

Inherit sample time — Inherit block sample time from parent subsystem
`off` (default) | `on`

Select this parameter to inherit the sample time of the parent subsystem as the block sample time. Doing so allows you to conditionally execute this block inside Function-Call Subsystem or Triggered Subsystem blocks. For an example, see "Using MPC Controller Block Inside Function-Call and Triggered Subsystems".

Note You must execute Function-Call Subsystem or Triggered Subsystem blocks at the sample rate of the controller. Otherwise, you can see unexpected results for two reasons.

- The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last MV value occurred in the past at the sample time specified in the MPC object, and when the block is executed with a different sample rate, this assumption no longer holds.
- The built-in Kalman estimator uses the sample time specified in the MPC object to provide an estimation of the current state to the MPC optimization problem, so when the block is executed with a different sample time, the estimated state is no longer correct.

If you clear this parameter (default), the sample time of the block is inherited from the controller object.

To view the sample time of a block, in the Simulink model window, on the **Debug** tab, under **Information Overlays**, select either **colors** or **Text**. For more information, see "View Sample Time Information" (Simulink).

Programmatic Use
Block Parameter: SampleTimeInherited_multiple
Type: string, character vector
Values: "off", "on"
Default: "off"

Version History
Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Functions
mpcmove

Objects
mpc | mpcstate

Blocks
Explicit MPC Controller | Multiple MPC Controllers

Topics
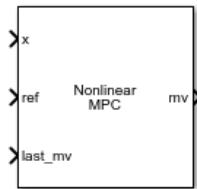"Simulation and Code Generation Using Simulink Coder"
"Simulation and Structured Text Generation Using Simulink PLC Coder"
"Gain-Scheduled MPC"
"Design Workflow for Explicit MPC"

Nonlinear MPC Controller

Simulate nonlinear model predictive controllers

Libraries:
Model Predictive Control Toolbox

Description

The Nonlinear MPC Controller block simulates a nonlinear model predictive controller. At each control interval, the block computes optimal control moves by solving a nonlinear programming problem. For more information on nonlinear MPC, see "Nonlinear MPC".

To use this block, you must first create an `nlmpc` object in the MATLAB workspace.

Limitations

• None of the Nonlinear MPC Controller block parameters are tunable.

Ports

Input

Required Inputs

x — input
vector

Current prediction model states, specified as a vector signal of length N_x, where N_x is the number of prediction model states. Since the nonlinear MPC controller does not perform state estimation, you must either measure or estimate the current prediction model states at each control interval.

ref — Model output reference values
row vector | matrix

Plant output reference values, specified as a row vector signal or matrix signal.

To use the same reference values across the prediction horizon, connect **ref** to a row vector signal with N_Y elements, where N_y is the number of output variables. Each element specifies the reference for an output variable.

To vary the references over the prediction horizon (previewing) from time $k+1$ to time $k+p$, connect **ref** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the references for one prediction horizon step. If you specify fewer than p rows, the final references are used for the remaining steps of the prediction horizon.

last_mv — Control signals used in plant at previous control interval
vector

Control signals used in plant at previous control interval, specified as a vector signal of lengthN_{mv}, where N_{mv} is the number of manipulated variables.

Note Connect **last_mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though this is not always the case. For example, if your controller is offline and running in tracking mode; that is, the controller output is not driving the plant, then feeding the actual control signal to **last_mv** can help achieve bumpless transfer when the controller is switched back online.

Additional Inputs

md — input
row vector | matrix

If your controller prediction model has measured disturbances you must enable this port and connect to it a row vector or matrix signal.

To use the same measured disturbance values across the prediction horizon, connect **md** to a row vector signal with N_{md} elements, where N_{md} is the number of manipulated variables. Each element specifies the value for a measured disturbance.

To vary the disturbances over the prediction horizon (previewing) from time k to time $k+p$, connect **md** to a matrix signal with N_{md} columns and up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbances for one prediction horizon step. If you specify fewer than $p+1$ rows, the final disturbances are used for the remaining steps of the prediction horizon.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

params — Optional parameters
bus

If your controller uses optional parameters in its prediction model, custom cost function, or custom constraint functions, enable this input port, and connect a parameter bus signal with N_p elements, where N_p is the number of parameters. For more information on creating a parameter bus signal, see `createParameterBus`. The controller passes these parameters to its model functions, cost function, constraint functions, passivity functions and Jacobian functions.

If your controller does not use optional parameters, you must disable **params**.

Dependencies

To enable this port, select the **Model parameters** parameter.

mv.target — Manipulated variable targets
vector

To specify manipulated variable targets, enable this input port, and connect a vector signal. To make a given manipulated variable track its specified target value, you must also specify a nonzero tuning weight for that manipulated variable.

The supplied **mv.target** values at run-time apply across the prediction horizon.

Dependencies

To enable this port, select the **Targets for manipulated variables** parameter.

Online Constraints

y.min — Minimum output variable constraints
vector | matrix

To specify run-time minimum output variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `OutputVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **y.min** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies the lower bound for an output variable.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **y.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower OV limits** parameter.

y.max — Maximum output variable constraints
vector | matrix

To specify run-time maximum output variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `OutputVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **y.max** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies the upper bound for an output variable.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **y.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper OV limits** parameter.

mv.min — Minimum manipulated variable constraints
vector | matrix

To specify run-time minimum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **mv.min** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the lower bound for a manipulated variable.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, connect **mv.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower MV limits** parameter.

mv.max — Maximum manipulated variable constraints
vector | matrix

To specify run-time maximum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.Max` property of its controller object.

To use the same bounds over the prediction horizon, connect **mv.max** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the upper bound for a manipulated variable.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, connect **mv.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper MV limits** parameter.

dmv.min — Minimum manipulated variable rate constraints
vector | matrix

To specify run-time minimum manipulated variable rate constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariable.RateMin` property of its controller object. **dmv.min** bounds must be nonpositive.

To use the same bounds over the prediction horizon, connect **dmv.min** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the lower bound for a manipulated variable rate of change.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, connect **dmv.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower MVRate limits** parameter.

dmv.max — Maximum manipulated variable rate constraints
vector | matrix

To specify run-time maximum manipulated variable rate constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.RateMax` property of its controller object. **dmv.max** bounds must be nonnegative.

To use the same bounds over the prediction horizon, connect **dmv.max** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the upper bound for a manipulated variable rate of change.

To vary the bounds over the prediction horizon from time k to time $k+p$-1, connect **dmv.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper MVRate limits** parameter.

x.min — Minimum state constraints
vector | matrix

To specify run-time minimum state constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `States.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **x.min** to a row vector signal with N_x elements, where N_x is the number of outputs. Each element specifies the lower bound for a state.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **x.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower state limits** parameter.

x.max — Maximum state constraints
vector | matrix

To specify run-time maximum state constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `States.Max` property of its controller object.

To use the same bounds over the prediction horizon, connect **x.max** to a row vector signal with N_x elements, where N_x is the number of outputs. Each element specifies the upper bound for a state.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **x.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper state limits** parameter.

Online Tuning Weights

y.wt — Output variable tuning weights
row vector | matrix

To specify run-time output variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.OutputVariables` property of its controller object. These tuning weights penalize deviations from output references.

If the MPC controller object uses constant output tuning weights over the prediction horizon, you can specify only constant output tuning weights at runtime. Similarly, if the MPC controller object uses output tuning weights that vary over the prediction horizon, you can specify only time-varying output tuning weights at runtime

To use constant tuning weights over the prediction horizon, connect **y.wt** to a row vector signal with N_y elements, where N_y is the number of outputs. Each element specifies a nonnegative tuning weight for an output variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time $k+1$ to time $k+p$, connect **y.wt** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **OV weights** parameter.

mv.wt — Manipulated variable tuning weights
row vector | matrix

To specify run-time manipulated variable tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariables` property of its controller object. These tuning weights penalize deviations from MV targets.

To use the same tuning weights over the prediction horizon, connect **mv.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **mv.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MV weights** parameter.

dmv.wt — Manipulated variable rate tuning weights
row vector | matrix

To specify run-time manipulated variable rate tuning weights, enable this input port. If this port is disabled, the block uses the tuning weights specified in the `Weights.ManipulatedVariablesRate` property of its controller object. These tuning weights penalize large changes in control moves.

To use the same tuning weights over the prediction horizon, connect **dmv.wt** to a row vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies a nonnegative tuning weight for a manipulated variable rate. For more information on specifying tuning weights, see "Tune Weights".

To vary the tuning weights over the prediction horizon from time k to time $k+p$-1, connect **dmv.wt** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the tuning weights for one prediction horizon step. If you specify fewer than p rows, the tuning weights in the final row apply for the remainder of the prediction horizon. For more information on varying weights over the prediction horizon, see "Setting Time-Varying Weights and Constraints with MPC Designer".

Dependencies

To enable this port, select the **MVRate weights** parameter.

ecr.wt — Slack variable tuning weight
scalar

To specify a run-time slack variable tuning weight, enable this input port and connect a scalar signal. If this port is disabled, the block uses the tuning weight specified in the `Weights.ECR` property of its controller object.

The slack variable tuning weight has no effect unless your controller object defines soft constraints whose associated ECR values are nonzero. If there are soft constraints, increasing the **ecr.wt** value makes these constraints relatively harder. The controller then places a higher priority on minimizing the magnitude of the predicted worst-case constraint violation.

Dependencies

To enable this port, select the **ECR weight** parameter.

Initial Guesses

mv.init — Initial guesses for the optimal manipulated variable solutions
vector | matrix

To specify initial guesses for the optimal manipulated variable solutions, enable this input port. If this port is disabled, the block uses the optimal control sequences calculated in the previous control interval as initial guesses.

To use the same initial guesses over the prediction horizon, connect **mv.init** to a vector signal with N_{mv} elements, where N_{mv} is the number of manipulated variables. Each element specifies the initial guess for a manipulated variable.

To vary the initial guesses over the prediction horizon from time k to time $k+p$-1, connect **mv.init** to a matrix signal with N_{mv} columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the guesses in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Initial guess** parameter.

x.init — Initial guesses for the optimal state variable solutions
vector | matrix

To specify initial guesses for the optimal state solutions, enable this input port. If this port is disabled, the block uses the optimal state sequences calculated in the previous control interval as initial guesses.

To use the same initial guesses over the prediction horizon, connect **x.init** to a vector signal with N_x elements, where N_x is the number of states. Each element specifies the initial guess for a state.

To vary the initial guesses over the prediction horizon from time k to time $k+p$-1, connect **x.init** to a matrix signal with N_x columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the initial guesses for one prediction horizon step. If you specify fewer than p rows, the guesses in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Initial guess** parameter.

e.init — Initial guess for the slack variable at the solution
nonnegative scalar

To specify an initial guess for the slack variable at the solution, enable this input port and connect a nonnegative scalar signal. If this port is disabled, the block uses an initial guess of 0.

Dependencies

To enable this port, select the **Initial guess** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, output as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the solver converges to a local optimum solution (**nlp.status** is positive), then **mv** contains the optimal solution.

If the solver reaches the maximum number of iterations without finding an optimal solution (**nlp.status** is zero) and the `Optimization.UseSuboptimalSolution` property of the controller is:

* `true`, then **mv** contains the suboptimal solution
* `false`, then **mv** is the same as **last_mv**

If the solver fails (**nlp.status** is negative), then **mv** is the same as **last_mv**.

Additional Outputs

cost — Objective function cost
nonnegative scalar

Objective function cost, output as a nonnegative scalar signal. The cost quantifies the degree to which the controller has achieved its objectives.

The cost value is only meaningful when the **nlp.status** output is nonnegative.

Dependencies

To enable this port, select the **Optimal cost** parameter.

slack — Slack variable
0 | nonnegative scalar

Slack variable, ε, used in constraint softening, output as 0 or a positive scalar value.

- ε = 0 — All soft constraints are satisfied over the entire prediction horizon.
- ε > 0 — At least one soft constraint is violated. When more than one constraint is violated, ε represents the worst-case soft constraint violation (scaled by the ECR values for each constraint).

Dependencies

To enable this port, select the **Slack variable** parameter.

nlp.status — Optimization status
scalar

Optimization status, output as one of the following:

- Positive Integer — Solver converged to an optimal solution
- 0 — Maximum number of iterations reached without converging to an optimal solution
- Negative integer — Solver failed

Dependencies

To enable this port, select the **Optimization status** parameter.

Optimal Sequences

mv.seq — Optimal manipulated variable sequence
matrix

Optimal manipulated variable sequence, returned as a matrix signal with $p+1$ rows and N_{mv} columns, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

The first p rows of **mv.seq** contain the calculated optimal manipulated variable values from current time k to time $k+p$-1. The first row of **mv.seq** contains the current manipulated variable values (output **mv**). Since the controller does not calculate optimal control moves at time $k+p$, the final two rows of **mv.seq** are identical.

Dependencies

To enable this port, select the **Optimal control sequence** parameter.

x.seq — Optimal prediction model state sequence
matrix

Optimal prediction model state sequence, returned as a matrix signal with $p+1$ rows and N_x columns, where p is the prediction horizon and N_x is the number of states.

The first row of **x.seq** contains the current estimated state values, either from the built-in state estimator or from the custom state estimation block input **x[k|k]**. The next p rows of **x.seq** contain the calculated optimal state values from time $k+1$ to time $k+p$.

Dependencies

To enable this port, select the **Optimal state sequence** parameter.

y.seq — Optimal output variable sequence
matrix

Optimal output variable sequence, returned as a matrix signal with $p+1$ rows and N_y columns, where p is the prediction horizon and N_y is the number of output variables.

The first p rows of **y.seq** contain the calculated optimal output values from current time k to time $k+p$-1. The first row of **y.seq** is computed based on the current estimated states and the current measured disturbances (first row of input **md**). Since the controller does not calculate optimal output values at time $k+p$, the final two rows of **y.seq** are identical.

Dependencies

To enable this port, select the **Optimal output sequence** parameter.

Parameters

Nonlinear MPC Controller — Controller object
nlmpc object name

You must provide an nlmpc object that defines a nonlinear MPC controller. To do so, enter the name of an nlmpc object in the MATLAB workspace.

Programmatic Use
Block Parameter: nlmpcobj
Type: string, character vector
Default: ""

Use prediction model sample time — Flag for using the prediction model sample time
on (default) | off

Select this parameter to run the controller using the same sample time as its prediction model. To use a different controller sample time, clear this parameter, and specify the sample time using the **Make block run at a different sample time** parameter.

To limit the number of decision variables and improve computational efficiency, you can run the controller with a sample time that is different from the prediction horizon. For example, consider the case of a nonlinear MPC controller running at 10 Hz. If the plant and controller sample times match, predicting plant behavior for ten seconds requires a prediction horizon of length 100, which produces a large number of decision variables. To reduce the number of decision variables, you can use a plant sample time of 1 second and a prediction horizon of length 10.

Programmatic Use
Block Parameter: UseObjectTs
Type: string, character vector
Values: "off", "on"
Default: "on"

Make block run at a different sample time — Controller sample time
positive finite scalar | -1

Specify this parameter to run the controller using a different sample time from its prediction model. Setting this parameter to -1 allows the block to inherit the sample time from its parent subsystem.

Note The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last value of the MV occurred in the past at the sample time specified in the MPC object. When the block is executed with a different sample rate, this assumption no longer holds, therefore, in this case, you must make sure that the weights and constraints defined in the controller handle the first element of the MV rate vector correctly.

Dependencies

To enable this parameter, clear the **Use prediction model sample time** parameter.

Programmatic Use
Block Parameter: TsControl
Type: string, character vector
Default: ""

Use MEX to speed up simulation — Flag for simulating controller use MEX function
off (default) | on

Select this parameter to simulate the controller using a MEX function generated using buildMEX. Doing so reduces the simulation time of the controller. To specify the name of the MEX function, use the **Specify MEX function name** parameter.

Programmatic Use
Block Parameter: UseMEX
Type: string, character vector
Values: "off", "on"
Default: "off"

Specify MEX function name — Controller MEX function name
string

Use this parameter to specify the name of the MEX function to use during simulation. To create the MEX function, use the buildMEX function.

Dependencies

To enable this parameter, select the **Use MEX to speed up simulation** parameter.

Programmatic Use
Block Parameter: mexname
Type: string, character vector
Default: ""

General Tab

Measured disturbances — Add measured disturbance input port
off (default) | on

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: md_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Targets for manipulated variables — Add manipulated variable target input port
off (default) | on

Select this parameter to add the **mv.target** input port to the block.

Programmatic Use
Block Parameter: mvtarget_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Model parameters — Add model parameters input port
off (default) | on

If your controller uses optional parameters, you must select this parameter to add the **params** output port to the block.

For more information on creating a parameter bus signal, see createParameterBus.

Programmatic Use
Block Parameter: param_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal cost — Add optimal cost output port
off (default) | on

Select this parameter to add the **cost** output port to the block.

Programmatic Use
Block Parameter: cost_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal control sequence — Add optimal control sequence output port
off (default) | on

Select this parameter to add the **mv.seq** output port to the block.

Programmatic Use
Block Parameter: mvseq_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal state sequence — Add optimal state sequence output port
off (default) | on

Select this parameter to add the **x.seq** output port to the block.

Programmatic Use
Block Parameter: stateseq_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal output sequence — Add optimal output sequence output port
off (default) | on

Select this parameter to add the **y.seq** output port to the block.

Programmatic Use
Block Parameter: ovseq_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Slack variable — Add slack variable output port
off (default) | on

Select this parameter to add the **slack** output port to the block.

Programmatic Use
Block Parameter: slack_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimization status — Add optimization status output port
off (default) | on

Select this parameter to add the **nlp.status** output port to the block.

Programmatic Use
Block Parameter: status_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Online Features Tab

Lower OV limits — Add minimum OV constraint input port
off (default) | on

Select this parameter to add the **ov.min** input port to the block.

Programmatic Use
Block Parameter: ov_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper OV limits — Add maximum OV constraint input port
off (default) | on

Select this parameter to add the **ov.max** input port to the block.

Programmatic Use
Block Parameter: ov_max
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower MV limits — Add minimum MV constraint input port
off (default) | on

Select this parameter to add the **mv.min** input port to the block.

Programmatic Use
Block Parameter: mv_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MV limits — Add maximum MV constraint input port
off (default) | on

Select this parameter to add the **mv.max** input port to the block.

Programmatic Use
Block Parameter: mv_max
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower MVRate limits — Add minimum MV rate constraint input port
off (default) | on

Select this parameter to add the **dmv.min** input port to the block.

Programmatic Use
Block Parameter: mvrate_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MVRate limits — Add maximum MV rate constraint input port
off (default) | on

Select this parameter to add the **dmv.max** input port to the block.

Programmatic Use
Block Parameter: mvrate_max
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower state limits — Add minimum state constraint input port
off (default) | on

Select this parameter to add the **x.min** input port to the block.

Programmatic Use
Block Parameter: state_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper state limits — Add maximum state constraint input port
off (default) | on

Select this parameter to add the **x.max** input port to the block.

Programmatic Use
Block Parameter: state_max
Type: string, character vector
Values: "off", "on"
Default: "off"

OV weights — Add OV tuning weights input port
off (default) | on

Select this parameter to add the **y.wt** input port to the block.

Programmatic Use
Block Parameter: ov_weight
Type: string, character vector
Values: "off", "on"
Default: "off"

MV weights — Add MV tuning weights input port
off (default) | on

Select this parameter to add the **mv.wt** input port to the block.

Programmatic Use
Block Parameter: mv_weight
Type: string, character vector
Values: "off", "on"
Default: "off"

MVRate weights — Add MV rate tuning weights input port
off (default) | on

Select this parameter to add the **dmv.wt** input port to the block.

Programmatic Use
Block Parameter: mvrate_weight
Type: string, character vector
Values: "off", "on"
Default: "off"

ECR weight — Add ECR tuning weight input port
off (default) | on

Select this parameter to add the **ecr.wt** input port to the block.

Programmatic Use
Block Parameter: `ecr_weight`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Initial guess — Add initial guess input ports
off (default) | on

Select this parameter to add the **mv.init**, **x.init**, and **e.init** input ports to the block.

Note By default, the Nonlinar MPC Controller block uses the calculated optimal manipulated variable and state trajectories from one control interval as the initial guesses for the next control interval.

Enable the initial guess ports only if it is necessary for your application.

Programmatic Use
Block Parameter: `nlp_initialize`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Version History
Introduced in R2018b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The Nonlinear MPC Controller block supports generating code only for nonlinear MPC controllers that use the default `fmincon` solver with the SQP algorithm.
- Code generation for single-precision or fixed-point computations is not supported.
- When used for code generation, nonlinear MPC controllers do not support expressing prediction model functions, stage cost functions or constraint functions as anonymous functions.
- If your controller uses optional parameters, you must also generate code for the Bus Creator block connected to the **params** input port. To do so, place the Nonlinear MPC Controller and Bus Creator blocks within a subsystem, and generate code for that subsystem.
- The **Support non-finite numbers** check box in the **Interface** section of the **Code Generation** options, under the model **Configuration Parameters** dialog box, must be checked (default option).
- When generating code using Embedded Coder®, the **Support variable-size signals** in the **Interface** section of the **Code Generation** options, under the model **Configuration Parameters** dialog box, must be checked. By default this check box is unchecked and you must check it before generating code.

See Also

Blocks
Multistage Nonlinear MPC Controller | Multiple MPC Controllers | Adaptive MPC Controller
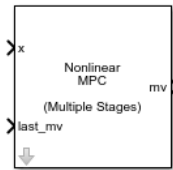
Functions
nlmpcmove | createParameterBus

Objects
nlmpc

Topics
"Nonlinear MPC"

Multistage Nonlinear MPC Controller

Simulate multistage nonlinear model predictive controllers

Libraries:
Model Predictive Control Toolbox

Description

The Multistage Nonlinear MPC Controller block simulates a multistage nonlinear model predictive controller. At each control interval, the block computes optimal control moves by solving a nonlinear programming problem in which different cost functions and constraints are defined for different prediction steps (stages). For more information on nonlinear MPC, see "Nonlinear MPC".

To use this block, you must first create an `nlmpcMultistage` object in the MATLAB workspace.

Limitations

• None of the Multistage Nonlinear MPC Controller block parameters are tunable.

Ports

Input

Required Inputs

x — input
vector

Current prediction model states, specified as a vector signal of length N_x, where N_x is the number of prediction model states. Since the nonlinear MPC controller does not perform state estimation, you must either measure or estimate the current prediction model states at each control interval.

last_mv — Control signals used in the plant at the previous control interval
vector

Control signals used in plant at previous control interval, specified as a vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

Note Connect **last_mv** to the MV signals actually applied to the plant in the previous control interval. Typically, these MV signals are the values generated by the controller, though sometimes they can come from a different source. For example, if your controller is offline and running in tracking mode, (that is, the controller output is not driving the plant), then feeding the actual plant input to **last_mv** can help achieve bumpless transfer when the controller is switched back online.

Additional Inputs

md — input
row vector | matrix

If your controller prediction model has measured disturbances you must enable this port and connect to it a row vector or matrix signal.

To use the same measured disturbance values across the prediction horizon, connect **md** to a row vector signal with N_{md} elements, where N_{md} is the number of manipulated variables. Each element specifies the value for a measured disturbance.

To vary the disturbances over the prediction horizon (previewing) from time k to time $k+p$, connect **md** to a matrix signal with N_{md} columns and up to $p+1$ rows. Here, k is the current time and p is the prediction horizon. Each row contains the disturbances for one prediction horizon step. If you specify fewer than $p+1$ rows, the final disturbances are used for the remaining steps of the prediction horizon.

Dependencies

To enable this port, select the **Measured disturbances** parameter.

state.param — Optional parameters
vector

If your controller uses optional parameters in its prediction model, enable this input port, and connect a vector signal with N_{pm} elements, where N_{pm} is the number of state parameters (equal to the `Model.ParameterLength` property of the `nlmpcMultistage` controller object). The controller passes these parameters to its model state transition and state Jacobian functions.

If your controller does not use optional parameters, you must disable the **state.param** port.

Dependencies

To enable this port, select the **StateFcn parameters** parameter.

stage.param — Optional parameters
vector

If your controller uses optional parameters in any stage cost or constraint function, enable this input port, and connect a vector signal with N_{pv} elements, where N_{pv} is the total number of parameters for all stage functions, and is equal to `sum(Stages.ParameterLength)`. The parameters for all stages are stacked in the parameter vector as follows.

```
[parameter vector for stage 1;
 parameter vector for stage 2;
 ...
 parameter vector for stage p+1;
]
```

At each stage, the controller passes the relevant parameter vector to the stage cost and constraint functions active at that stage.

If your controller does not use optional parameters, you must disable the **stage.param** port. For more information, see `nlmpcMultistage` and `nlmpcmove`.

Dependencies

To enable this port, select the **Stacked stage parameters** parameter.

Online Constraints

mv.min — Minimum manipulated variable constraints
vector | matrix

To specify run-time minimum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariables.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **mv.min** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the lower bound for a manipulated variable.

To vary the bounds over the prediction horizon from time k to time $k+p$–1, connect **mv.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower MV limits** parameter.

mv.max — Maximum manipulated variable constraints
vector | matrix

To specify run-time maximum manipulated variable constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.Max` property of its controller object.

To use the same bounds over the prediction horizon, connect **mv.max** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the upper bound for a manipulated variable.

To vary the bounds over the prediction horizon from time k to time $k+p$–1, connect **mv.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper MV limits** parameter.

dmv.min — Minimum manipulated variable rate constraints
vector | matrix

To specify run-time minimum manipulated variable rate constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `ManipulatedVariable.RateMin` property of its controller object. **dmv.min** bounds must be nonpositive.

To use the same bounds over the prediction horizon, connect **dmv.min** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the lower bound for a manipulated variable rate of change.

To vary the bounds over the prediction horizon from time k to time $k+p$–1, connect **dmv.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower MVRate limits** parameter.

dmv.max — Maximum manipulated variable rate constraints
vector | matrix

To specify run-time maximum manipulated variable rate constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `ManipulatedVariables.RateMax` property of its controller object. **dmv.max** bounds must be nonnegative.

To use the same bounds over the prediction horizon, connect **dmv.max** to a row vector signal with N_{mv} elements, where N_{mv} is the number of outputs. Each element specifies the upper bound for a manipulated variable rate of change.

To vary the bounds over the prediction horizon from time k to time $k+p$–1, connect **dmv.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper MVRate limits** parameter.

x.min — Minimum state constraints
vector | matrix

To specify run-time minimum state constraints, enable this input port. If this port is disabled, the block uses the lower bounds specified in the `States.Min` property of its controller object.

To use the same bounds over the prediction horizon, connect **x.min** to a row vector signal with N_x elements, where N_x is the number of outputs. Each element specifies the lower bound for a state.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **x.min** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Lower state limits** parameter.

x.max — Maximum state constraints
vector | matrix

To specify run-time maximum state constraints, enable this input port. If this port is disabled, the block uses the upper bounds specified in the `States.Max` property of its controller object.

To use the same bounds over the prediction horizon, connect **x.max** to a row vector signal with N_x elements, where N_x is the number of outputs. Each element specifies the upper bound for a state.

To vary the bounds over the prediction horizon from time $k+1$ to time $k+p$, connect **x.max** to a matrix signal with N_y columns and up to p rows. Here, k is the current time and p is the prediction horizon. Each row contains the bounds for one prediction horizon step. If you specify fewer than p rows, the bounds in the final row apply for the remainder of the prediction horizon.

Dependencies

To enable this port, select the **Upper state limits** parameter.

Others

x.terminal — Terminal state
vector

Terminal state, specified as a vector signal of length N_x. To specify desired terminal state constraints, enable this input port. To specify desired terminal states at run-time via this input port, you must specify finite values in the `TerminalState` field of the `Model` property of the `nlmpcMultistage` object that is passed as a parameter to the block. Specify `inf` for the states that you do not need to constrain to a terminal value. At run time, the block ignores any values in the input signal that correspond to `inf` values in the object. If you do not specify any terminal value condition in the `nlmpcMultistage` object, the signal at this input port is ignored at runtime.

If this port is not enabled the terminal state constraint (if present) does not change at run time.

Dependencies

To enable this port, select the **Terminal state** parameter.

z0 — Initial guesses for the decision variables vector
vector

To specify initial guesses for the decision variable vector, enable this input port. If this port is disabled, the block uses the decision variable sequences calculated in the previous control interval as initial guesses. Good initial guesses are important since they help the solver to converge to a solution faster.

z0 is a column vector of length equal to the sum of the lengths of all the decision variable vectors for each stage. The initial guesses must be stacked as follows.

```
[state vector guess for stage 1;
 manipulated variable vector guess for stage 1;
 manipulated variable vector rate guess for stage 1; % if used
 slack variable vector guess for stage 1; % if used
 state vector guess for stage 2;
 manipulated variable vector guess for stage 2;
 manipulated variable vector rate guess for stage 2; % if used
 slack variable vector guess for stage 2; % if used
 ...
 state vector guess for stage p;
 manipulated variable vector guess for stage p;
 manipulated variable vector rate guess for stage p; % if used
 slack variable vector guess for stage p; % if used
 state vector guess for stage p+1;
 slack variable vector guess for stage p+1; % if used
]
```

For more information, see `nlmpcMultistage` and `nlmpcmove`.

Dependencies

To enable this port, select the **Initial guess** parameter.

Output

Required Output

mv — Optimal manipulated variable control action
column vector

Optimal manipulated variable control action, output as a column vector signal of length N_{mv}, where N_{mv} is the number of manipulated variables.

If the solver converges to a local optimum solution (**nlp.status** is positive), then **mv** contains the optimal solution.

If the solver reaches the maximum number of iterations without finding an optimal solution (**nlp.status** is zero) and the `Optimization.UseSuboptimalSolution` property of the controller is `true`, then **mv** contains the suboptimal solution, otherwise, **mv** is the same as **last_mv**.

If the solver fails (**nlp.status** is negative), then **mv** is the same as **last_mv**.

Additional Outputs

cost — Objective function cost
nonnegative scalar

Objective function cost, output as a nonnegative scalar signal. The cost quantifies the degree to which the controller has achieved its objectives.

The cost value is meaningful only when the **nlp.status** output is nonnegative.

Dependencies

To enable this port, select the **Optimal cost** parameter.

slack — Stacked slack variables vector
nonnegative vector

Stacked slack variables vector, used in constraint softening. If all elements are zero, then all soft constraints are satisfied over the entire prediction horizon. If any element is greater than zero, then at least one soft constraint is violated.

The slack variable vector for all stages are stacked as follows.

```
[slack variable vector for stage 1; % if used
 slack variable vector for stage 2; % if used
 ...
 slack variable vector for stage p+1; % if used
]
```

nlp.status — Optimization status
scalar

Optimization status, output as one of the following:

- Positive Integer — Solver converged to an optimal solution
- `0` — Maximum number of iterations reached without converging to an optimal solution
- Negative integer — Solver failed

Dependencies

To enable this port, select the **Optimization status** parameter.

Optimal Sequences

mv.seq — Optimal manipulated variable sequence
matrix

Optimal manipulated variable sequence, returned as a matrix signal with $p+1$ rows and N_{mv} columns, where p is the prediction horizon and N_{mv} is the number of manipulated variables.

The first p rows of **mv.seq** contain the calculated optimal manipulated variable values from current time k to time $k+p$-1. The first row of **mv.seq** contains the current manipulated variable values (output **mv**). Since the controller does not calculate optimal control moves at time $k+p$, the final two rows of **mv.seq** are identical.

Dependencies

To enable this port, select the **Optimal control sequence** parameter.

x.seq — Optimal prediction model state sequence
matrix

Optimal prediction model state sequence, returned as a matrix signal with $p+1$ rows and N_x columns, where p is the prediction horizon and N_x is the number of states.

The first row of **x.seq** contains the current estimated state values, either from the built-in state estimator or from the custom state estimation block input **x[k|k]**. The next p rows of **x.seq** contain the calculated optimal state values from time $k+1$ to time $k+p$.

Dependencies

To enable this port, select the **Optimal state sequence** parameter.

Parameters

Multistage Nonlinear MPC Controller — Controller object
nlmpcMultistage object name

You must provide an nlmpcMultistage object that defines a nonlinear MPC controller. To do so, enter the name of an nlmpc object in the MATLAB workspace.

Programmatic Use
Block Parameter: nlmpcobj
Type: string, character vector
Default: " "

Use prediction model sample time — Flag for using the prediction model sample time
on (default) | off

Select this parameter to run the controller using the same sample time as its prediction model. To use a different controller sample time, clear this parameter, and specify the sample time using the **Make block run at a different sample time** parameter.

To limit the number of decision variables and improve computational efficiency, you can run the controller with a sample time that is different from the prediction horizon. For example, consider the case of a nonlinear MPC controller running at 10 Hz. If the plant and controller sample times match, predicting plant behavior for ten seconds requires a prediction horizon of length 100, which produces a large number of decision variables. To reduce the number of decision variables, you can use a plant sample time of 1 second and a prediction horizon of length 10.

Programmatic Use
Block Parameter: `UseObjectTs`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"on"`

Make block run at a different sample time — Controller sample time
positive finite scalar | `-1`

Specify this parameter to run the controller using a different sample time from its prediction model. Setting this parameter to `-1` allows the block to inherit the sample time from its parent subsystem.

Note The first element of the MV rate vector (which is the difference between the current and the last value of the manipulated variable) is normally weighted and constrained assuming that the last value of the MV occurred in the past at the sample time specified in the MPC object. When the block is executed with a different sample rate, this assumption no longer holds, therefore, in this case, you must make sure that the weights and constraints defined in the controller handle the first element of the MV rate vector correctly.

Dependencies

To enable this parameter, clear the **Use prediction model sample time** parameter.

Programmatic Use
Block Parameter: `TsControl`
Type: string, character vector
Default: `""`

Use MEX to speed up simulation — Flag for simulating controller use MEX function
off (default) | on

Select this parameter to simulate the controller using a MEX function generated using `buildMEX`. Doing so reduces the simulation time of the controller. To specify the name of the MEX function, use the **Specify MEX function name** parameter.

Programmatic Use
Block Parameter: `UseMEX`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Specify MEX function name — Controller MEX function name
string

Use this parameter to specify the name of the MEX function to use during simulation. To create the MEX function, use the `buildMEX` function.

Dependencies

To enable this parameter, select the **Use MEX to speed up simulation** parameter.

Programmatic Use
Block Parameter: `mexname`
Type: string, character vector
Default: `""`

General Tab

Measured disturbances — Add measured disturbance input port
off (default) | on

If your controller has measured disturbances, you must select this parameter to add the **md** output port to the block.

Programmatic Use
Block Parameter: `md_enabled`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

StateFcn parameter — Add state function parameters input port
off (default) | on

If your prediction model uses optional parameters, you must select this parameter to add the **state.param** input port to the block.

Programmatic Use
Block Parameter: `stateparam_enabled`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Stacked stage parameters — Add stage functions parameter input port
off (default) | on

If your cost or constraint functions use parameters at any stage, you must select this parameter to add the **stage.param** input port to the block.

Programmatic Use
Block Parameter: `stageparam_enabled`
Type: string, character vector
Values: `"off"`, `"on"`
Default: `"off"`

Optimal cost — Add optimal cost output port
off (default) | on

Select this parameter to add the **cost** output port to the block.

Programmatic Use
Block Parameter: cost_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal control sequence — Add optimal control sequence output port
off (default) | on

Select this parameter to add the **mv.seq** output port to the block.

Programmatic Use
Block Parameter: mvseq_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimal state sequence — Add optimal state sequence output port
off (default) | on

Select this parameter to add the **x.seq** output port to the block.

Programmatic Use
Block Parameter: stateseq_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Slack variable — Add slack variable output port
off (default) | on

Select this parameter to add the **slack** output port to the block.

Programmatic Use
Block Parameter: slack_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Optimization status — Add optimization status output port
off (default) | on

Select this parameter to add the **nlp.status** output port to the block.

Programmatic Use
Block Parameter: status_enabled
Type: string, character vector
Values: "off", "on"
Default: "off"

Online Features Tab

Lower MV limits — Add minimum MV constraint input port
off (default) | on

Select this parameter to add the **mv.min** input port to the block.

Programmatic Use
Block Parameter: mv_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MV limits — Add maximum MV constraint input port
off (default) | on

Select this parameter to add the **mv.max** input port to the block.

Programmatic Use
Block Parameter: mv_max
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower MVRate limits — Add minimum MV rate constraint input port
off (default) | on

Select this parameter to add the **dmv.min** input port to the block.

Programmatic Use
Block Parameter: mvrate_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper MVRate limits — Add maximum MV rate constraint input port
off (default) | on

Select this parameter to add the **dmv.max** input port to the block.

Programmatic Use
Block Parameter: mvrate_max
Type: string, character vector
Values: "off", "on"
Default: "off"

Lower state limits — Add minimum state constraint input port
off (default) | on

Select this parameter to add the **x.min** input port to the block.

Programmatic Use
Block Parameter: state_min
Type: string, character vector
Values: "off", "on"
Default: "off"

Upper state limits — Add maximum state constraint input port
off (default) | on

Select this parameter to add the **x.max** input port to the block.

Programmatic Use
Block Parameter: state_max
Type: string, character vector
Values: "off", "on"
Default: "off"

Terminal state — Terminal State
off (default) | on

Select this parameter to add the **x.terminal** input port to the block.

Programmatic Use
Block Parameter: terminal_state
Type: string, character vector
Values: "off", "on"
Default: "off"

Initial guess — Add initial guess input port
off (default) | on

Select this parameter to add the **z0** input port to the block.

Note By default, the Nonlinar MPC Controller block uses the calculated optimal states, manipulated variables, and slack variables from one control interval as initial guesses for the next control interval.

Enable the initial guess port only if you need it for your application.

Programmatic Use
Block Parameter: nlp_initialize
Type: string, character vector
Values: "off", "on"
Default: "off"

Version History
Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The Multistage Nonlinear MPC Controller block supports generating code only for multistage nonlinear MPC controllers that use the default fmincon solver with the SQP algorithm.
- Code generation for single-precision or fixed-point computations is not supported.
- When used for code generation, nonlinear MPC controllers do not support expressing prediction model functions, stage cost functions, or constraint functions as anonymous functions.
- In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, you must select the following parameters.

- **Support non-finite numbers** — This parameter is selected by default.
- **Support variable-size signals** when using Embedded Coder — This parameter is not selected by default.

See Also

Blocks
Nonlinear MPC Controller | Multiple MPC Controllers | Adaptive MPC Controller

Functions
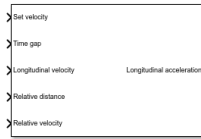nlmpcmove

Objects
nlmpcMultistage

Topics
"Nonlinear MPC"

Adaptive Cruise Control System

Simulate adaptive cruise control using model predictive controller

Libraries:
Model Predictive Control Toolbox / Automated Driving

Description

The Adaptive Cruise Control System block simulates an adaptive cruise control (ACC) system that tracks a set velocity and maintains a safe distance from a lead vehicle by adjusting the longitudinal acceleration of an ego vehicle. The block computes optimal control actions while satisfying safe distance, velocity, and acceleration constraints using model predictive control (MPC).
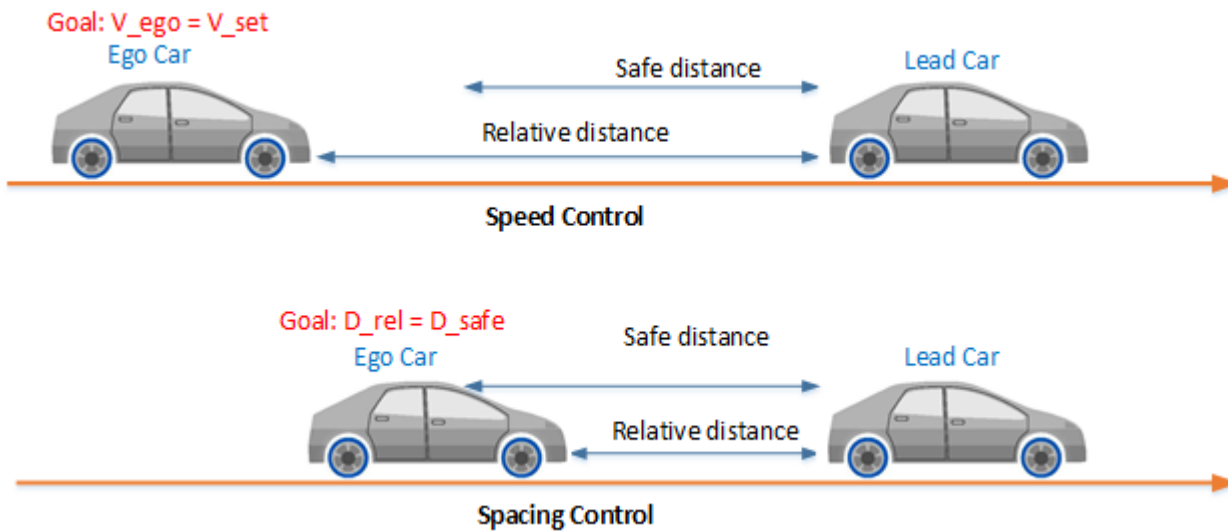
To customize your controller, for example to use advanced MPC features or modify controller initial conditions, click **Create ACC subsystem**.

Ports

Input

Set velocity — Ego vehicle velocity setpoint
nonnegative scalar

Ego vehicle velocity setpoint in m/s. When there is no lead vehicle, the controller tracks this velocity.

Time gap — Safe time gap
nonnegative scalar

Safe time gap in seconds between the lead vehicle and the ego vehicle. This time gap is used to calculate the minimum safe following distance constraint. For more information, see "Safe Following Distance" on page 4-123.

Longitudinal velocity — Ego vehicle velocity
nonnegative scalar

Ego vehicle velocity in m/s.

Relative distance — Distance between lead vehicle and ego vehicle
positive scalar

Distance in meters between lead vehicle and ego vehicle. To calculate this signal, subtract the ego vehicle position from the lead vehicle position.

Relative velocity — Velocity difference between lead vehicle and ego vehicle
scalar

Velocity difference in meters per second between lead vehicle and ego vehicle. To calculate this signal, subtract the ego vehicle velocity from the lead vehicle velocity.

Minimum longitudinal acceleration — Minimum ego vehicle acceleration
negative scalar

Minimum ego vehicle longitudinal acceleration constraint in m/s^2. Use this input port when the minimum acceleration varies at run time.

Dependencies

To enable this port, select **Use external source** for the **Minimum longitudinal acceleration** parameter.

Maximum longitudinal acceleration — Maximum ego vehicle acceleration
positive scalar

Maximum ego vehicle longitudinal acceleration constraint in m/s^2. Use this input port when the maximum acceleration varies at run time.

Dependencies

To enable this port, select **Use external source** for the **Maximum longitudinal acceleration** parameter.

Enable optimization — Controller optimization enable signal
scalar

Controller optimization enable signal. When this signal is:

• Nonzero, the controller performs optimization calculations and generates a **Longitudinal acceleration** control signal.

• Zero, the controller does not perform optimization calculations. In this case, the **Longitudinal acceleration** output signal remains at the value it had when the optimization was disabled. The controller continues to update its internal state estimates.

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

External control signal — Longitudinal acceleration applied to ego vehicle
scalar

Actual longitudinal acceleration in m/s^2 applied to the ego vehicle. The controller uses this signal to estimate the ego vehicle model states. Use this input port when the control signal applied to the ego vehicle does not match the optimal control signal computed by the model predictive controller. This mismatch can occur when, for example:

- The Adaptive Cruise Control System is not the active controller. Maintaining an accurate state estimate when the controller is not active prevents bumps in the control signal when the controller becomes active.
- The acceleration actuator fails and does not provide the correct control signal to the ego vehicle.

Dependencies

To enable this port, select the **Use external control signal for bumpless transfer between ACC and other controllers** parameter.

Output

Longitudinal acceleration — Acceleration control signal
scalar

Acceleration control signal in m/s^2 generated by the controller.

Parameters

Parameters Tab

Ego Vehicle Model

Linear model from longitudinal acceleration to longitudinal velocity — Ego vehicle model
`tf(1,[0.5,1,0])` (default) | LTI model | linear System Identification Toolbox model

The linear model from the ego vehicle longitudinal acceleration to its longitudinal velocity, specified as an LTI model or a linear System Identification Toolbox model. The controller creates its internal predictive model by augmenting the ego vehicle dynamic model.

Programmatic Use
Block Parameter: `EgoModel`
Type: string, character vector
Default: `"tf(1,[0.5,1,0])"`

Initial condition for longitudinal velocity — Initial velocity of the ego vehicle model
20 (default) | nonnegative scalar

Initial velocity in m/s of the ego vehicle model, which can differ from the actual ego vehicle initial velocity.

This value is used to configure the initial conditions of the model predictive controller. For more information, see "Initial Conditions" on page 4-123.

Programmatic Use
Block Parameter: `InitialEgoVelocity`
Type: string, character vector
Default: `"20"`

Default spacing — Minimum spacing to lead vehicle
10 (default) | nonnegative scalar

Minimum spacing in meters between the lead vehicle and the ego vehicle. This value corresponds to the target relative distance between the ego and lead vehicles when the ego vehicle velocity is zero.

This value is used to calculate the:

• Minimum safe following distance. For more information, see "Safe Following Distance" on page 4-123.
• Controller initial conditions. For more information, see "Initial Conditions" on page 4-123.

Programmatic Use
Block Parameter: `DefaultSpacing`
Type: string, character vector
Default: `"10"`

Maximum velocity — Maximum longitudinal velocity
50 (default) | positive scalar

Maximum ego vehicle longitudinal velocity in m/s.

Programmatic Use
Block Parameter: `MaxVelocity`
Type: string, character vector
Default: `"50"`

Adaptive Cruise Controller Constraints

Minimum longitudinal acceleration — Minimum ego vehicle acceleration
-3 (default) | negative scalar

Minimum ego vehicle longitudinal acceleration constraint in m/s^2.

If the minimum acceleration varies over time, add the **Minimum longitudinal acceleration** input port to the block by selecting **Use external source**.

Programmatic Use
Block Parameter: `MinAcceleration`
Type: string, character vector
Default: `"-3"`

Maximum longitudinal acceleration — Maximum ego vehicle acceleration
2 (default) | nonnegative scalar

Maximum ego vehicle longitudinal acceleration constraint in m/s^2.

If the maximum acceleration varies over time, add the **Maximum longitudinal acceleration** input port to the block by selecting **Use external source**.

Programmatic Use
Block Parameter: MaxAcceleration
Type: string, character vector
Default: "2"

Model Predictive Controller Settings

Sample time — Controller sample time
0.1 (default) | positive scalar

Controller sample time in seconds.

Programmatic Use
Block Parameter: Ts
Type: string, character vector
Default: "0.1"

Prediction horizon — Controller prediction horizon
10 (default) | positive integer

Controller prediction horizon steps. The controller prediction time is the product of the sample time and the prediction horizon.

Programmatic Use
Block Parameter: PredictionHorizon
Type: string, character vector
Default: "30"

Controller behavior — Closed-loop controller performance
0.5 (default) | scalar between 0 and 1

Closed-loop controller performance. The default parameter value provides a balanced controller design. Specifying a:

- Smaller value produces a more robust controller with smoother control actions.
- Larger value produces a more aggressive controller with a faster response time.

When you modify this parameter, the change is applied to the controller immediately.

Programmatic Use
Block Parameter: ControllerBehavior
Type: string, character vector
Default: "0.5"

Block Tab

Use suboptimal solution — Apply suboptimal solution after specified number of iterations
off (default) | on

Configure the controller to apply a suboptimal solution after a specified maximum number of iterations, which guarantees the worst-case execution time for your controller.

For more information, see "Suboptimal QP Solution".

Dependencies

After selecting this parameter, specify the **Maximum iteration number** parameter.

Programmatic Use
Block Parameter: `suboptimal`
Type: string, character vector
Default: `"off"`

Maximum iteration number — Maximum optimization iterations
10 (default) | positive integer

Maximum number of controller optimization iterations.

Dependencies

To enable this parameter, select the **Use suboptimal solution** parameter.

Programmatic Use
Block Parameter: `maxiter`
Type: string, character vector
Default: `"10"`

Use external signal to enable or disable optimization — Add port for enabling optimization
off (default) | on

To add the **Enable optimization** input port to the block, select this parameter.

Programmatic Use
Block Parameter: `optmode`
Type: string, character vector
Default: `"off"`

Use external signal for bumpless transfer between ACC and other controllers — Add external
control signal input port
off (default) | on

Select this parameter to add the **External control signal** input port to the block.

Programmatic Use
Block Parameter: `trackmode`
Type: string, character vector
Default: `"off"`

Create ACC subsystem — Create custom controller
button

Generate a custom ACC subsystem, which you can modify for your application. The configuration data
for the custom controller is exported to the MATLAB workspace as a structure.

You can modify the custom controller subsystem to:

- Modify default MPC settings or use advanced MPC features.
- Modify the default controller initial conditions.
- Use different application settings, such as a custom safe following distance definition.

Algorithms

Safe Following Distance

By default, the model predictive controller computes the safe following distance constraint; that is, the minimum relative distance between the lead and ego vehicle, as:

$$D_R = D_S + G_T * V_E$$

Here:

- D_S is the **Default spacing** parameter.
- G_T is the **Time gap** input signal.
- V_E is the **Longitudinal velocity** input signal.

To define a different safe following distance constraint, create a custom cruise control system by, on the **Block** tab, clicking **Create ACC subsystem**.

Initial Conditions

By default, the model predictive controller assumes the following initial conditions:

- Longitudinal velocity of both the ego vehicle and the lead vehicle equal the **Initial condition for longitudinal velocity** parameter value.
- Ego vehicle longitudinal acceleration is zero.
- Relative distance between the lead vehicle and ego vehicle is:

$$D_R = D_S + G_T * V_E$$

Here:

- D_S is the **Default spacing** parameter.
- G_T is the time gap and is assumed to be `1.4`.
- V_E is the **Initial longitudinal velocity** parameter.

If the initial conditions in your model do not match these conditions, the **Longitudinal acceleration** output can exhibit an initial bump at the start of the simulation.

To modify the controller initial conditions to match your simulation, create a custom cruise control system by, on the **Block** tab, clicking **Create ACC subsystem**.

Version History
Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
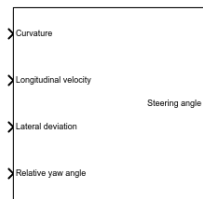
See Also

Blocks
MPC Controller | Path Following Control System | Lane Keeping Assist System

Topics
"Automated Driving Using Model Predictive Control"

Lane Keeping Assist System

Simulate lane-keeping assistance using adaptive model predictive controller

Libraries:
Model Predictive Control Toolbox / Automated Driving

Description

The Lane Keeping Assist System block simulates a lane keeping assist (LKA) system that keeps an ego vehicle traveling along the center of a straight or curved road by adjusting the front steering angle. The controller reduces the lateral deviation and relative yaw angle of the ego vehicle with respect to the lane centerline. The block computes optimal control actions while satisfying steering angle constraints using adaptive model predictive control (MPC).
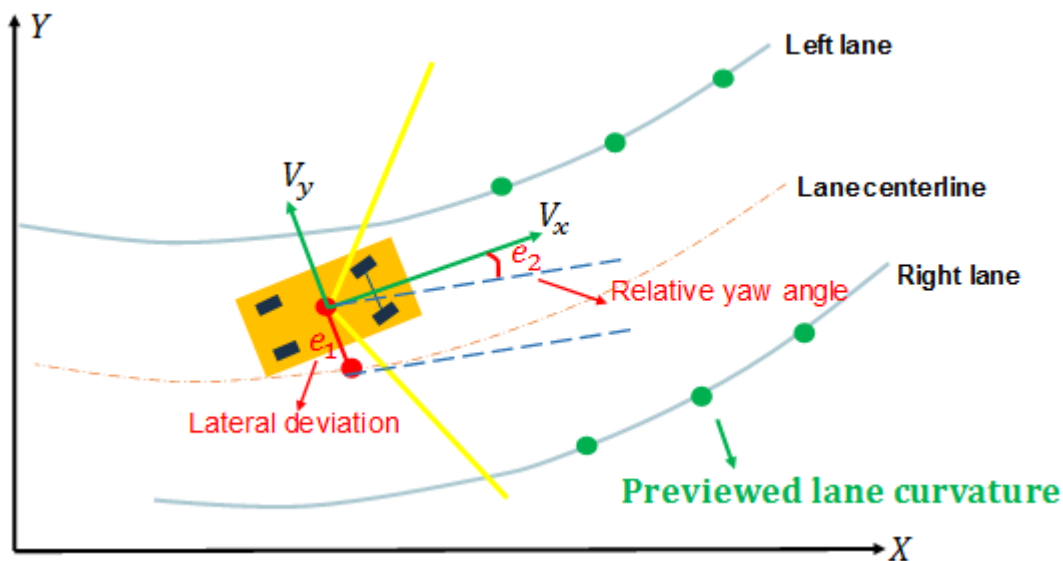
To customize your controller, for example to use advanced MPC features or modify controller initial conditions, click **Create LKA subsystem**.

Ports

Input

Curvature — Road curvature
scalar | vector

Road curvature, specified as $1/R$, where R is the radius of the curve in meters.

The road curvature is:

* Positive when the road curves toward the positive Y axis of the global coordinate system.
* Negative when the road curves toward the negative Y axis of the global coordinate system.
* Zero for a straight road.

The controller models the road curvature as a measured disturbance with previewing. You can specify the curvature as a:

* Scalar signal — Specify the curvature for the current control interval. The controller uses this curvature value across the prediction horizon.
* Vector signal with length less than or equal to the **Prediction Horizon** parameter — Specify the current and predicted curvature values across the prediction horizon. If the length of the vector is less than the prediction horizon, then the controller uses the final curvature value in the vector for the remainder of the prediction horizon.

Longitudinal velocity — Ego vehicle velocity
nonnegative scalar

Ego vehicle velocity in m/s.

Lateral deviation — Ego vehicle lateral deviation
scalar

Ego vehicle lateral deviation in meters from the centerline of the lane. The lateral deviation e_1 is positive when the ego vehicle is to the right of the centerline and negative when the ego vehicle is to the left of the centerline.

Relative yaw angle — Angle from lane centerline
scalar

Ego vehicle longitudinal axis angle in radians from the centerline of the lane, defined as:

$$e_2 = \theta_e - \theta_c$$

Here, θ_e is the ego vehicle angle and θ_c is the centerline angle, with both angles defined in the global coordinate frame.

Minimum steering angle — Minimum front steering angle
scalar

Minimum front steering angle constraint in radians. Use this input port when the minimum steering angle varies at run time.

Dependencies

To enable this port, select corresponding **Use external source** parameter.

Maximum steering angle — Maximum front steering angle
scalar

Maximum front steering angle constraint in radians. Use this input port when the maximum steering angle varies at run time.

Dependencies

To enable this port, select corresponding **Use external source** parameter.

Enable optimization — Controller optimization enable signal
scalar

Controller optimization enable signal. When this signal is:

- Nonzero, the controller performs optimization calculations and generates a **Steering angle** control signal.
- Zero, the controller does not perform optimization calculations. In this case, the **Steering angle** output signal remains at the value it had when the optimization was disabled. The controller continues to update its internal state estimates.

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

External control signal — Steering angle applied to ego vehicle
scalar

Actual steering angle in radians applied to the ego vehicle. The controller uses this signal to estimate the ego vehicle model states. Use this input port when the control signal applied to the ego vehicle does not match the optimal control signal computed by the model predictive controller. This mismatch can occur when, for example:

- The Lane Keeping Assist System is not the active controller. Maintaining an accurate state estimate when the controller is not active prevents bumps in the control signal when the controller becomes active.
- The steering actuator fails and does not provide the correct control signal to the ego vehicle.

Dependencies

To enable this port, select the **Use external control signal for bumpless transfer between PFC and other controllers** parameter.

Vehicle dynamics matrix A — State matrix of ego vehicle predictive model
square matrix

State matrix of ego vehicle predictive model. The number of rows in the state matrix corresponds to the number of states in the predictive model. This matrix must be square.

The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Dependencies

To enable this port, select the **Use vehicle model** parameter.

Vehicle dynamics matrix B — Input-to-state matrix of ego vehicle predictive model
column vector

Input-to-state matrix of ego vehicle predictive model. The number of rows in this signal must match the number of rows in **Vehicle dynamics matrix A**.

The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Dependencies

To enable this port, select the **Use vehicle model** parameter.

Vehicle dynamics matrix C — State-to-output matrix of ego vehicle predictive model
matrix with two rows

State-to-output matrix of ego vehicle predictive model. The number of columns in this signal must match the number of rows in **Vehicle dynamics matrix A**.

The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Dependencies

To enable this port, select the **Use vehicle model** parameter.

Output

Steering angle — Front steering angle control signal
scalar

Front steering angle control signal in radians generated by the controller. The front steering angle is the angle of the front tires from the longitudinal axis of the vehicle. The steering angle is positive towards the positive lateral axis of the ego vehicle.

Parameters

Parameters Tab

Ego Vehicle

Use vehicle parameters — Define ego vehicle model using vehicle properties
on (default) | off

Select this parameter to define the ego vehicle model used by the MPC controller by specifying properties of the ego vehicle. The ego vehicle model is the linear model from the front steering angle to the lateral velocity and yaw angle rate. For more information, see "Ego Vehicle Predictive Model" on page 4-135.

To define the vehicle model, specify the following block parameters:

- **Total mass**
- **Yaw moment of inertia**
- **Longitudinal distance from center of gravity to front tires**
- **Longitudinal distance from center of gravity to rear tires**
- **Cornering stiffness of front tires**
- **Cornering stiffness of rear tires**

For more information on the ego vehicle model, see "Ego Vehicle Predictive Model" on page 4-135.

Selecting this parameter clears the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: ModelType
Type: string, character vector
Default: "Use vehicle parameters"

Use vehicle model — Define ego vehicle model using state-space matrices
off (default) | on

Select this parameter to define the state-space matrices of the ego vehicle model used by the MPC controller. This model is the linear model from the front steering angle in radians to the lateral velocity in meters per second and yaw angle rate in radians per second. For more information on the ego vehicle model, see "Ego Vehicle Predictive Model" on page 4-135.

To define the initial internal model, specify the **A**, **B**, and **C** state-space matrices. The internal model must be a minimal realization with no direct feedthrough, and the dimensions of **A**, **B**, and **C** must be consistent.

Typically, the ego vehicle steering model is velocity-dependent, and therefore, it varies over time. To update the internal model at run time, use the **Vehicle dynamics A**, **Vehicle dynamics B**, and **Vehicle dynamics C** input ports.

Selecting this parameter clears the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: ModelType
Type: string, character vector
Default: "Use vehicle parameters"

Total mass — Ego vehicle mass
1575 (default) | positive scalar

Ego vehicle mass in kg.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: VehicleMass
Type: string, character vector
Default: "1575"

Yaw moment of inertia — Moment of inertia about the ego vehicle vertical axis
2875 (default) | positive scalar

Moment of inertia about the ego vehicle vertical axis in $Kg \cdot m^2$.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: VehicleYawInertia
Type: string, character vector
Default: "2875"

Longitudinal distance from center of gravity to front tires — Distance from the ego vehicle center of mass to its front tires
1.2 (default) | positive scalar

Distance from the ego vehicle center of mass to its front tires in meters, measured along the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: LengthToFront
Type: string, character vector
Default: "1.2"

Longitudinal distance from center of gravity to rear tires — Distance from the ego vehicle center of mass to its rear tires
1.6 (default) | positive scalar

Distance from the ego vehicle center of mass to its rear tires in meters, measured along the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: LengthToRear
Type: string, character vector
Default: "1.6"

Cornering stiffness of front tires — Front tire stiffness
19000 (default) | positive scalar

Front tire stiffness in N/rad, defined as the relationship between the side force on the front tires and the angle of the tires to the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: FrontTireStiffness
Type: string, character vector
Default: "19000"

Cornering stiffness of rear tires — Rear tire stiffness
33000 (default) | positive scalar

Rear tire stiffness in N/rad, defined as the relationship between the side force on the rear tires and the angle of the tires to the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `RearTireStiffness`
Type: string, character vector
Default: `"33000"`

A — Initial state matrix of ego vehicle predictive model
square matrix

Initial state matrix of ego vehicle predictive model. The number of rows in the state matrix corresponds to the number of states in the predictive model. This matrix must be square.

The initial ego vehicle predictive model defined by **A**, **B**, and **C** must be minimal.

Typically, the ego vehicle model varies over time. To update the state matrix at run time, use the **Vehicle dynamics A** input port.

Dependencies

To enable this parameter, select the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: `EgoModelMatrixA`
Type: string, character vector
Default: `"[-4.4021 ,-12.4603;1.3913,-5.1868]"`

B — Initial input-to-state matrix of ego vehicle predictive model
column vector

Initial input-to-state matrix of ego vehicle predictive model. The number of rows in this parameter must match the number of rows in **A**.

The initial ego vehicle predictive model defined by **A**, **B**, and **C** must be minimal.

Typically, the ego vehicle model varies over time. To update the input-to-state matrix at run time, use the **Vehicle dynamics B** input port.

Dependencies

To enable this parameter, select the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: `EgoModelMatrixB`
Type: string, character vector
Default: `"[24.1270;15.8609]"`

C — Initial state-to-output matrix of ego vehicle predictive model
matrix with two rows

Initial state-to-output matrix of ego vehicle predictive model. The number of columns in this parameter must match the number of rows in **A**.

The initial ego vehicle predictive model defined by **A**, **B**, and **C** must be minimal.

Typically, the ego vehicle model varies over time. To update the state-to-output matrix at run time, use the **Vehicle dynamics C** input port.

4-131

Dependencies

To enable this parameter, select the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: EgoModelMatrixC
Type: string, character vector
Default: "[1,0;0,1]"

Initial longitudinal velocity — Initial velocity of the ego vehicle
15 (default) | positive scalar

Initial velocity of the ego vehicle model when the lane-keeping assist is enabled in m/s. This velocity can differ from the actual ego vehicle initial velocity.

Note A very small initial velocity, for example eps, can produce a nonminimal realization for the controller plant model, causing an error. To prevent this error, set the initial velocity to a larger value, for example 1e-3.

Programmatic Use
Block Parameter: InitialLongVel
Type: string, character vector
Default: "15"

Transport lag between model inputs and outputs — Total transport lag in ego vehicle model
0 (default) | nonnegative scalar

Total transport lag, τ, in the ego vehicle model in seconds. This lag includes actuator, sensor, and communication lags. For each input-output channel, the transport lag is approximated by:

$$\frac{1}{\tau s + 1}$$

Programmatic Use
Block Parameter: TransportLag
Type: string, character vector
Default: "0"

Lane Keeping Controller Constraints

Minimum steering angle — Minimum front steering angle
-0.26 (default) | scalar between -pi/2 and pi/2

Minimum front steering angle constraint in radians.

If the minimum steering angle varies over time, add the **Minimum steering angle** input port to the block by selecting **Use external source**.

Dependencies

This parameter must be less than the **Maximum steering angle** parameter.

Programmatic Use
Block Parameter: MinSteering

Type: string, character vector
Default: "-0.26"

Maximum steering angle — Maximum front steering angle
0.26 (default) | scalar between -pi/2 and pi/2

Maximum front steering angle constraint in radians.

If the maximum steering angle varies over time, add the **Maximum steering angle** input port to the block by selecting **Use external source**.

Dependencies

This parameter must be greater than the **Minimum steering angle** parameter.

Programmatic Use
Block Parameter: MaxSteering
Type: string, character vector
Default: "0.26"

Model Predictive Controller Settings

Sample time — Controller sample time
0.1 (default) | positive scalar

Controller sample time in seconds.

Programmatic Use
Block Parameter: Ts
Type: string, character vector
Default: "0.1"

Prediction horizon — Controller prediction horizon
10 (default) | positive integer

Controller prediction horizon steps. The controller prediction time is the product of the sample time and the prediction horizon.

Programmatic Use
Block Parameter: PredictionHorizon
Type: string, character vector
Default: "30"

Controller behavior — Closed-loop controller performance
0.5 (default) | scalar between 0 and 1

Closed-loop controller performance. The default parameter value provides a balanced controller design. Specifying a:

- Smaller value produces a more robust controller with smoother control actions.

- Larger value produces a more aggressive controller with a faster response time.

When you modify this parameter, the change is applied to the controller immediately.

Programmatic Use
Block Parameter: ControllerBehavior

Type: string, character vector
Default: "0.5"

Block Tab

Use suboptimal solution — Apply suboptimal solution after specified number of iterations
off (default) | on

Configure the controller to apply a suboptimal solution after a specified maximum number of iterations, which guarantees the worst-case execution time for your controller.

For more information, see "Suboptimal QP Solution".

Dependencies

After selecting this parameter, specify the **Maximum iteration number** parameter.

Programmatic Use
Block Parameter: suboptimal
Type: string, character vector
Default: "off"

Maximum iteration number — Maximum optimization iterations
10 (default) | positive integer

Maximum number of controller optimization iterations.

Dependencies

To enable this parameter, select the **Use suboptimal solution** parameter.

Programmatic Use
Block Parameter: maxiter
Type: string, character vector
Default: "10"

Use external signal to enable or disable optimization — Add port for enabling optimization
off (default) | on

To add the **Enable optimization** input port to the block, select this parameter.

Programmatic Use
Block Parameter: optmode
Type: string, character vector
Default: "off"

Use external signal for bumpless transfer between LKA and other controllers — Add external control signal input port
off (default) | on

To add the **External control signal** input port to the block, select this parameter.

Programmatic Use
Block Parameter: trackmode
Type: string, character vector
Default: "off"

Create LKA subsystem — Create custom controller
button

Generate a custom LKA subsystem, which you can modify for your application. The controller configuration data for the custom controller is exported to the MATLAB workspace as a structure.

You can modify the custom controller subsystem to:

- Modify default MPC settings or use advanced MPC features.
- Modify the default controller initial conditions.

Algorithms

Ego Vehicle Predictive Model

The default ego vehicle predictive model is the following state-space model:

$$A = \begin{bmatrix} -2(C_F + C_R)/m/V_X & -V_X - 2(C_F L_F - C_R L_R)/m/V_X \\ -2(C_F L_F - C_R L_R)/I_Z/V_X & -2\left(C_F L_F^2 + C_R L_R^2\right)/I_Z/V_X \end{bmatrix}$$

$$B = 2C_F\begin{bmatrix} 1/m \\ L_F/I_Z \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Here:

- V_X is the longitudinal velocity of the car. At the start of the simulation, this velocity is equal to the **Initial condition for longitudinal velocity** parameter. At run time, this velocity is equal to the **Longitudinal velocity** input signal.
- m is the **Total mass** parameter.
- I_Z is the **Yaw moment of inertia** parameter.
- L_F is the **Longitudinal distance from center of gravity to front tires** parameter.
- L_R is the **Longitudinal distance from center of gravity to rear tires** parameter.
- C_F is the **Cornering stiffness of front tires** parameter.
- C_R is the **Cornering stiffness of rear tires** parameter.

The input to this model is the steering angle in radians, and the outputs are the lateral velocity in meters per second and yaw angle rate in radians per second.

To define a different ego vehicle predictive model, select the **Use vehicle model** parameter, and specify the initial state-space model. Then, specify the run-time values of the state-space matrices using the **Vehicle dynamics A**, **Vehicle dynamics B**, and **Vehicle dynamics C** input signals.

The controller creates its internal predictive model by augmenting the ego vehicle dynamic model. The augmented model includes the road curvature as a measured disturbance input signal.

Initial Conditions

By default, the model predictive controller assumes the following initial conditions for the ego vehicle:

- Longitudinal velocity is equal to the **Initial longitudinal velocity** parameter.
- Lateral velocity is zero.
- Steering angle is zero.
- Yaw angle rate is zero.

If the initial conditions in your model do not match these conditions, the **Steering angle** output can exhibit an initial bump at the start of the simulation.

To modify the controller initial conditions to match your simulation, create a custom lane-keeping control system by, on the **Block** tab, clicking **Create LKA subsystem**.

Version History
Introduced in R2018a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
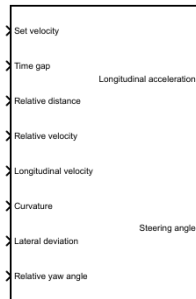Generate Structured Text code using Simulink® PLC Coder™.

See Also

Blocks
Adaptive MPC Controller | Adaptive Cruise Control System | Path Following Control System

Topics
"Automated Driving Using Model Predictive Control"

Path Following Control System

Simulate path-following control using adaptive model predictive controller

Libraries:
Model Predictive Control Toolbox / Automated Driving

Description

The Path Following Control System block simulates a path-following control (PFC) system that keeps an ego vehicle traveling along the center of a straight or curved road while tracking a set velocity and maintaining a safe distance from a lead vehicle. To do so, the controller adjusts both the longitudinal acceleration and front steering angle of the ego vehicle. The block computes optimal control actions while satisfying safe distance, velocity, acceleration, and steering angle constraints using adaptive model predictive control (MPC).

This block combines the capabilities of the Lane Keeping Assist System and Adaptive Cruise Control System blocks into a single controller.
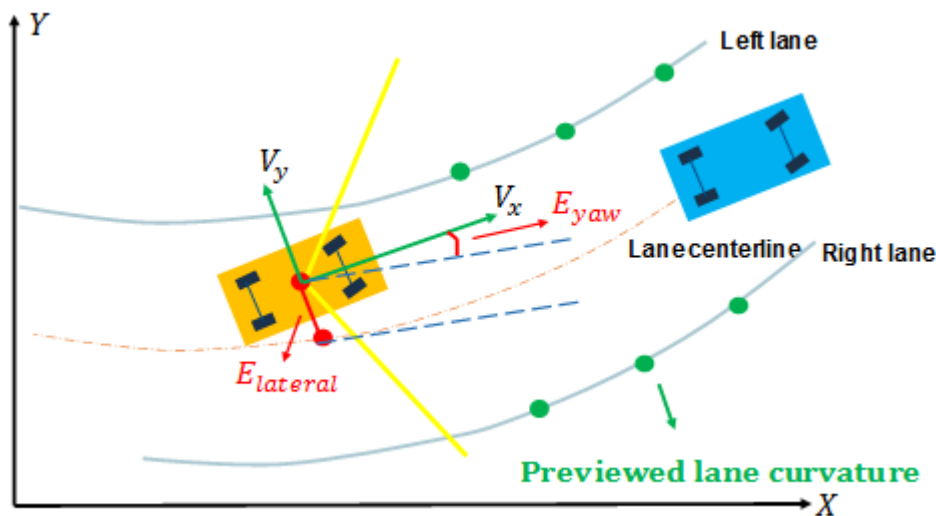
To customize your controller, for example to use advanced MPC features or modify controller initial conditions, click **Create PFC subsystem**.

Ports

Input

Set velocity — Ego vehicle velocity setpoint
nonnegative scalar

Ego vehicle velocity setpoint in m/s. When there is no lead vehicle, the controller tracks this velocity.

Time gap — Safe time gap
nonnegative scalar

Safe time gap in seconds between the lead vehicle and the ego vehicle. This time gap is used to calculate the minimum safe following distance constraint. For more information, see "Safe Following Distance" on page 4-152.

Relative distance — Distance between lead vehicle and ego vehicle
positive scalar

Distance in meters between lead vehicle and ego vehicle. To calculate this signal, subtract the ego vehicle position from the lead vehicle position.

Relative velocity — Velocity difference between lead vehicle and ego vehicle
scalar

Velocity difference in meters per second between lead vehicle and ego vehicle. To calculate this signal, subtract the ego vehicle velocity from the lead vehicle velocity.

Longitudinal velocity — Ego vehicle velocity
nonnegative scalar

Ego vehicle velocity in m/s.

Curvature — Road curvature
scalar | vector

Road curvature, specified as $1/R$, where R is the radius of the curve in meters.

The road curvature is:

- Positive when the road curves toward the positive Y axis of the global coordinate system.
- Negative when the road curves toward the negative Y axis of the global coordinate system.
- Zero for a straight road.

The controller models the road curvature as a measured disturbance with previewing. You can specify the curvature as a:

- Scalar signal — Specify the curvature for the current control interval. The controller uses this curvature value across the prediction horizon.
- Vector signal with length less than or equal to the **Prediction Horizon** parameter — Specify the current and predicted curvature values across the prediction horizon. If the length of the vector is less than the prediction horizon, then the controller uses the final curvature value in the vector for the remainder of the prediction horizon.

Lateral deviation — Ego vehicle lateral deviation
scalar

Ego vehicle lateral deviation in meters from the centerline of the lane. The lateral deviation e_1 is positive when the ego vehicle is to the right of the centerline and negative when the ego vehicle is to the left of the centerline.

Relative yaw angle — Angle from lane centerline
scalar

Ego vehicle longitudinal axis angle in radians from the centerline of the lane, defined as:

$$e_2 = \theta_e - \theta_c$$

Here, θ_e is the ego vehicle angle and θ_c is the centerline angle, with both angles defined in the global coordinate frame.

Minimum longitudinal acceleration — Minimum ego vehicle acceleration
scalar

Minimum ego vehicle longitudinal acceleration constraint in m/s^2. Use this input port when the minimum acceleration varies at run time.

Dependencies

To enable this port, select corresponding **Use external source** parameter.

Maximum longitudinal acceleration — Maximum ego vehicle acceleration
scalar

Maximum ego vehicle longitudinal acceleration constraint in m/s^2. Use this input port when the maximum acceleration varies at run time.

Dependencies

To enable this port, select corresponding **Use external source** parameter.

Minimum steering angle — Minimum front steering angle
scalar

Minimum front steering angle constraint in radians. Use this input port when the minimum steering angle varies at run time.

Dependencies

To enable this port, select corresponding **Use external source** parameter.

Maximum steering angle — Maximum front steering angle
scalar

Maximum front steering angle constraint in radians. Use this input port when the maximum steering angle varies at run time.

Dependencies

To enable this port, select corresponding **Use external source** parameter.

Enable optimization — Controller optimization enable signal
scalar

Controller optimization enable signal. When this signal is:

- Nonzero, the controller performs optimization calculations and generates the **Longitudinal acceleration** and **Steering angle** control signals.
- Zero, the controller does not perform optimization calculations. In this case, the **Longitudinal acceleration** and **Steering angle** output signals remain at the values they had when the optimization was disabled. The controller continues to update its internal state estimates.

Dependencies

To enable this port, select the **Use external signal to enable or disable optimization** parameter.

External control signal — Control signals applied to ego vehicle
vector of length two

Actual control signals applied to the ego vehicle. The first element of this signal is the longitudinal acceleration in m/s^2, and the second element is the steering angle in radians. The controller uses these signals to estimate the ego vehicle model states. Use this input port when the control signals applied to the ego vehicle do not match the optimal control signals computed by the model predictive controller. This mismatch can occur when, for example:

- The Path Following Control System is not the active controller. Maintaining an accurate state estimate when the controller is not active prevents bumps in the control signals when the controller becomes active.
- The steering or acceleration actuator fails and does not provide the correct control signal to the ego vehicle.

Dependencies

To enable this port, select the **Use external control signal for bumpless transfer between PFC and other controllers** parameter.

Vehicle dynamics matrix A — State matrix of ego vehicle predictive model
square matrix

State matrix of ego vehicle predictive model. The number of rows in the state matrix corresponds to the number of states in the predictive model. This matrix must be square.

The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Dependencies

To enable this port, select the **Use vehicle model** parameter.

Vehicle dynamics matrix B — Input-to-state matrix of ego vehicle predictive model
matrix with two columns

Input-to-state matrix of ego vehicle predictive model. The number of rows in this signal must match the number of rows in **Vehicle dynamics matrix A**.

The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Dependencies

To enable this port, select the **Use vehicle model** parameter.

Vehicle dynamics matrix C — State-to-output matrix of ego vehicle predictive model
matrix with three rows

State-to-output matrix of ego vehicle predictive model. The number of columns in this signal must match the number of rows in **Vehicle dynamics matrix A**.

The ego vehicle predictive model defined by **Vehicle dynamics matrix A**, **Vehicle dynamics matrix B**, and **Vehicle dynamics matrix C** must be minimal.

Dependencies

To enable this port, select the **Use vehicle model** parameter.

Output

Longitudinal acceleration — Acceleration control signal
scalar

Acceleration control signal in m/s^2 generated by the controller.

Steering angle — Front steering angle control signal
scalar

Front steering angle control signal in radians generated by the controller. The front steering angle is the angle of the front tires from the longitudinal axis of the vehicle. The steering angle is positive towards the positive lateral axis of the ego vehicle.

Parameters

Parameters Tab

Ego Vehicle

Use vehicle parameters — Define ego vehicle model using vehicle properties
on (default) | off

Select this parameter to define the ego vehicle model used by the MPC controller by specifying properties of the ego vehicle. The ego vehicle model is the linear model from the longitudinal acceleration and front steering angle to the longitudinal velocity, lateral velocity, and yaw angle rate.

To define the vehicle model, specify the following block parameters:

- **Total mass**
- **Yaw moment of inertia**
- **Longitudinal distance from center of gravity to front tires**
- **Longitudinal distance from center of gravity to rear tires**
- **Cornering stiffness of front tires**
- **Cornering stiffness of rear tires**
- **Longitudinal acceleration tracking time constant**

For more information on the ego vehicle model, see "Ego Vehicle Predictive Model" on page 4-150

Selecting this parameter clears the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: ModelType
Type: string, character vector
Default: "Use vehicle parameters"

Use vehicle model — Define ego vehicle model using state-space matrices
off (default) | on

Select this parameter to define the state-space matrices of the ego vehicle model used by the MPC controller. The ego vehicle model is the linear model from the longitudinal acceleration and front steering angle to the longitudinal velocity, lateral velocity, and yaw angle rate.

To define the initial internal model, specify the **A**, **B**, and **C** state-space matrices. The internal model must be a minimal realization with no direct feedthrough, and the dimensions of **A**, **B**, and **C** must be consistent.

Typically, the ego vehicle model is velocity-dependent, and therefore, it varies over time. To update the internal model at run time, use the **Vehicle dynamics A**, **Vehicle dynamics B**, and **Vehicle dynamics C** input ports.

For more information on the ego vehicle model, see "Ego Vehicle Predictive Model" on page 4-150

Selecting this parameter clears the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: ModelType
Type: string, character vector
Default: "Use vehicle parameters"

Total mass — Ego vehicle mass
1575 (default) | positive scalar

Ego vehicle mass in kg.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: VehicleMass
Type: string, character vector
Default: "1575"

Yaw moment of inertia — Moment of inertia about the ego vehicle vertical axis
2875 (default) | positive scalar

Moment of inertia about the ego vehicle vertical axis in Kg·m^2.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `VehicleYawInertia`
Type: string, character vector
Default: `"2875"`

Longitudinal distance from center of gravity to front tires — Distance from the ego vehicle center of mass to its front tires
1.2 (default) | positive scalar

Distance from the ego vehicle center of mass to its front tires in meters, measured along the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `LengthToFront`
Type: string, character vector
Default: `"1.2"`

Longitudinal distance from center of gravity to rear tires — Distance from the ego vehicle center of mass to its rear tires
1.6 (default) | positive scalar

Distance from the ego vehicle center of mass to its rear tires in meters, measured along the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `LengthToRear`
Type: string, character vector
Default: `"1.6"`

Cornering stiffness of front tires — Front tire stiffness
19000 (default) | positive scalar

Front tire stiffness in N/rad, defined as the relationship between the side force on the front tires and the angle of the tires to the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `FrontTireStiffness`
Type: string, character vector
Default: `"19000"`

Cornering stiffness of rear tires — Rear tire stiffness
33000 (default) | positive scalar

Rear tire stiffness in N/rad, defined as the relationship between the side force on the rear tires and the angle of the tires to the longitudinal axis of the vehicle.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `RearTireStiffness`
Type: string, character vector
Default: `"33000"`

Longitudinal acceleration tracking time constant — Time constant for acceleration tracking
`0.5` (default) | positive scalar

Time constant for tracking longitudinal acceleration, specified in seconds.

Dependencies

To enable this parameter, select the **Use vehicle parameters** parameter.

Programmatic Use
Block Parameter: `AccelTimeConstant`
Type: string, character vector
Default: `"0.5"`

A — Initial state matrix of ego vehicle predictive model
square matrix

Initial state matrix of ego vehicle predictive model. The number of rows in the state matrix corresponds to the number of states in the predictive model. This matrix must be square.

The initial ego vehicle predictive model defined by **A**, **B**, and **C** must be minimal.

Typically, the ego vehicle model varies over time. To update the state matrix at run time, use the **Vehicle dynamics A** input port.

Dependencies

To enable this parameter, select the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: `EgoModelMatrixA`
Type: string, character vector
Default: `"[-4.4021 ,-12.4603;1.3913,-5.1868]"`

B — Initial input-to-state matrix of ego vehicle predictive model
matrix with two columns

Initial input-to-state matrix of ego vehicle predictive model. The number of rows in this parameter must match the number of rows in **A**.

The initial ego vehicle predictive model defined by **A**, **B**, and **C** must be minimal.

Typically, the ego vehicle model varies over time. To update the input-to-state matrix at run time, use the **Vehicle dynamics B** input port.

Dependencies

To enable this parameter, select the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: `EgoModelMatrixB`
Type: string, character vector
Default: `"[24.1270;15.8609]"`

C — Initial state-to-output matrix of ego vehicle predictive model
matrix with three rows

Initial state-to-output matrix of ego vehicle predictive model. The number of columns in this parameter must match the number of rows in **A**.

The initial ego vehicle predictive model defined by **A**, **B**, and **C** must be minimal.

Typically, the ego vehicle model varies over time. To update the state-to-output matrix at run time, use the **Vehicle dynamics C** input port.

Dependencies

To enable this parameter, select the **Use vehicle model** parameter.

Programmatic Use
Block Parameter: `EgoModelMatrixC`
Type: string, character vector
Default: `"[1,0;0,1]"`

Initial longitudinal velocity — Initial velocity of the ego vehicle model
15 (default) | nonnegative scalar

Initial velocity of the ego vehicle model in m/s, which can differ from the actual ego vehicle initial velocity.

This value is used to configure the initial conditions of the model predictive controller. For more information, see "Initial Conditions" on page 4-152.

Note A very small initial velocity, for example `eps`, can produce a nonminimal realization for the controller plant model, causing an error. To prevent this error, set the initial velocity to a larger value, for example `1e-3`.

Programmatic Use
Block Parameter: `InitialLongVel`
Type: string, character vector
Default: `"15"`

Transport lag between model inputs and outputs — Total transport lag in ego vehicle model
0 (default) | nonnegative scalar

Total transport lag, τ, in the ego vehicle model in seconds. This lag includes actuator, sensor, and communication lags. For each input-output channel, the transport lag model is:

$$\frac{1}{\tau s + 1}$$

Programmatic Use
Block Parameter: `TransportLag`

Type: string, character vector
Default: "0"

Spacing Control

Maintain safe distance between lead vehicle and ego vehicle — Enable spacing control
on (default) | off

To configure the safe following distance, set the **Default spacing** parameter. For more information on the safe following distance used by the controller, see "Safe Following Distance" on page 4-152.

Programmatic Use
Block Parameter: spaceCtrl
Type: string, character vector
Default: "on"

Default spacing — Minimum spacing to lead vehicle
10 (default) | nonnegative scalar

Minimum spacing in meters between the lead vehicle and the ego vehicle. This value corresponds to the target relative distance between the ego and lead vehicles when the ego vehicle velocity is zero.

This value is used to calculate the:

- Minimum safe following distance. For more information, see "Safe Following Distance" on page 4-152.
- Controller initial conditions. For more information, see "Initial Conditions" on page 4-152.

Dependencies

To enable this parameter, select the **Maintain safe distance between lead vehicle and ego vehicle** parameter.

Programmatic Use
Block Parameter: DefaultSpacing
Type: string, character vector
Default: "10"

Controller Tab

Path Following Controller Constraints

Minimum steering angle — Minimum front steering angle
-0.26 (default) | scalar between -pi/2 and pi/2

Minimum front steering angle constraint in radians.

If the minimum steering angle varies over time, add the **Minimum steering angle** input port to the block by selecting **Use external source**.

Dependencies

This parameter must be less than the **Maximum steering angle** parameter.

Programmatic Use
Block Parameter: MinSteering
Type: string, character vector

Default: "-0.26"

Maximum steering angle — Maximum front steering angle
0.26 (default) | scalar between -pi/2 and pi/2

Maximum front steering angle constraint in radians.

If the maximum steering angle varies over time, add the **Maximum steering angle** input port to the block by selecting **Use external source**.

Dependencies

This parameter must be greater than the **Minimum steering angle** parameter.

Programmatic Use
Block Parameter: MaxSteering
Type: string, character vector
Default: "0.26"

Minimum longitudinal acceleration — Minimum ego vehicle acceleration
-3 (default) | scalar

Minimum ego vehicle longitudinal acceleration constraint in m/s^2.

If the minimum acceleration varies over time, add the **Minimum longitudinal acceleration** input port to the block by selecting **Use external source**.

Programmatic Use
Block Parameter: MinAcceleration
Type: string, character vector
Default: "-3"

Maximum longitudinal acceleration — Maximum ego vehicle acceleration
2 (default) | scalar

Maximum ego vehicle longitudinal acceleration constraint in m/s^2.

If the maximum acceleration varies over time, add the **Maximum longitudinal acceleration** input port to the block by selecting **Use external source**.

Programmatic Use
Block Parameter: MaxAcceleration
Type: string, character vector
Default: "2"

Model Predictive Controller Settings

Sample time — Controller sample time
0.1 (default) | positive scalar

Controller sample time in seconds.

Programmatic Use
Block Parameter: Ts
Type: string, character vector
Default: "0.1"

Prediction horizon — Controller prediction horizon
10 (default) | positive integer

Controller prediction horizon steps. The controller prediction time is the product of the sample time and the prediction horizon.

Programmatic Use
Block Parameter: PredictionHorizon
Type: string, character vector
Default: "30"

Control horizon — Controller control horizon
3 (default) | positive integer | vector of positive integers

Controller control horizon, specified as one of the following:

- Positive integer less than or equal to the **Prediction horizon** parameter. In this case, the controller computes m free control moves occurring at times k through $k+m$-1, and holds the controller output constant for the remaining prediction horizon steps from $k+m$ through $k+p$-1. Here, k is the current control interval.

- Vector of positive integers, $[m_1, m_2, ...]$, where the sum of the integers equals the **Prediction horizon** parameter. In this case, the controller computes M blocks of free moves, where M is the length of the control horizon vector. The first free move applies to times k through $k+m_1$-1, the second free move applies from time $k+m_1$ through $k+m_1+m_2$-1, and so on. Using block moves can improve the robustness of your controller.

Programmatic Use
Block Parameter: PredictionHorizon
Type: string, character vector
Default: "30"

Controller Behavior

Weight on velocity tracking — Tuning weight for longitudinal velocity tracking
0.1 (default) | positive scalar

Tuning weight for longitudinal velocity tracking. To produce smaller velocity-tracking errors, increase this weight.

Programmatic Use
Block Parameter: LongWeight
Type: string, character vector
Default: "0.1"

Weight on lateral error — Tuning weight for lateral error
1 (default) | positive scalar

Tuning weight for lateral error. To produce smaller lateral errors, increase this weight.

Programmatic Use
Block Parameter: LateralWeight
Type: string, character vector
Default: "1"

Weight on change of longitudinal acceleration — Tuning weight for change in longitudinal acceleration
`0.1` (default) | positive scalar

Tuning weight for changes in longitudinal acceleration. To produce less-aggressive vehicle acceleration, increase this weight.

Programmatic Use
Block Parameter: `AccelRateWeight`
Type: string, character vector
Default: `"0.1"`

Weight on change of steering angle — Tuning weight for change in steering angle
`0.1` (default) | positive scalar

Tuning weight for changes in steering angle. To produce less-aggressive steering angle changes, increase this weight.

Programmatic Use
Block Parameter: `SteerRateWeight`
Type: string, character vector
Default: `"0.1"`

Block Tab

Use suboptimal solution — Apply suboptimal solution after specified number of iterations
`off` (default) | on

Configure the controller to apply a suboptimal solution after a specified maximum number of iterations, which guarantees the worst-case execution time for your controller.

For more information, see "Suboptimal QP Solution".

Dependencies

After selecting this parameter, specify the **Maximum iteration number** parameter.

Programmatic Use
Block Parameter: `suboptimal`
Type: string, character vector
Default: `"off"`

Maximum iteration number — Maximum optimization iterations
`10` (default) | positive integer

Maximum number of controller optimization iterations.

Dependencies

To enable this parameter, select the **Use suboptimal solution** parameter.

Programmatic Use
Block Parameter: `maxiter`
Type: string, character vector
Default: `"10"`

Use external signal to enable or disable optimization — Add port for enabling optimization
`off` (default) | on

To add the **Enable optimization** input port to the block, select this parameter.

Programmatic Use
Block Parameter: optmode
Type: string, character vector
Default: "off"

Use external signal for bumpless transfer between PFC and other controllers — Add external control signal input port
off (default) | on

To add the **External control signal** input port to the block, select this parameter.

Programmatic Use
Block Parameter: trackmode
Type: string, character vector
Default: "off"

Create PFC subsystem — Create custom controller
button

Generate a custom PFC subsystem, which you can modify for your application. The configuration data for the custom controller is exported to the MATLAB workspace as a structure.

You can modify the custom controller subsystem to:

- Modify default MPC settings or use advanced MPC features.
- Modify the default controller initial conditions.
- Use different application settings, such as a custom safe following distance definition.

Algorithms

Ego Vehicle Predictive Model

The default ego vehicle predictive model for path-following control is the combination of two state-space models, one for adaptive cruise control and one for lane keeping.

Adaptive Cruise Control Predictive Model

The predictive state-space model for adaptive cruise control is:

$$A_1 = \begin{bmatrix} -1/\tau & 0 \\ 1 & 0 \end{bmatrix}$$
$$B_1 = \begin{bmatrix} 1/\tau \\ 0 \end{bmatrix}$$
$$C_1 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$D_1 = 0$$

Here, τ is the **Longitudinal acceleration tracking time constant** parameter.

The input to this model is the longitudinal acceleration in m/s^2, and the output is the longitudinal velocity in meters per second.

Lane-Keeping Predictive Model

The predictive state-space model for lane keeping is:

$$A_2 = \begin{bmatrix} -2(C_F + C_R)/m/V_X & -V_X - 2(C_F L_F - C_R L_R)/m/V_X \\ -2(C_F L_F - C_R L_R)/I_Z/V_X & -2\left(C_F L_F^2 + C_R L_R^2\right)/I_Z/V_X \end{bmatrix}$$

$$B_2 = 2C_F \begin{bmatrix} 1/m \\ L_F/I_Z \end{bmatrix}$$

$$C_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$D_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Here:

- V_X is the longitudinal velocity of the car. At the start of the simulation, this velocity is equal to the **Initial condition for longitudinal velocity** parameter. At run time, this velocity is equal to the **Longitudinal velocity** input signal.
- m is the **Total mass** parameter.
- I_Z is the **Yaw moment of inertia** parameter.
- L_F is the **Longitudinal distance from center of gravity to front tires** parameter.
- L_R is the **Longitudinal distance from center of gravity to rear tires** parameter.
- C_F is the **Cornering stiffness of front tires** parameter.
- C_R is the **Cornering stiffness of rear tires** parameter.

The input to this model is the steering angle in radians. The outputs are the lateral velocity in meters per second and yaw angle rate in radians per second.

Combined Path-Following Predictive Model

The Path Following Control System block combines these models as follows:

$$A = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}$$

$$B = \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix}$$

$$C = \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix}$$

$$D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}$$

The inputs to this combined model are the longitudinal acceleration in m/s^2 and steering angle in radians. The outputs are the longitudinal velocity in meters per second, lateral velocity in meters per second, and yaw angle rate in radians per second.

The controller creates its internal predictive model by augmenting the ego vehicle dynamic model. The augmented model includes the road curvature as a measured disturbance input signal.

Define a Custom Model

To define a different ego vehicle predictive model, select the **Use vehicle model** parameter, and specify the initial state-space model. Then, specify the run-time values of the state-space matrices using the **Vehicle dynamics A**, **Vehicle dynamics B**, and **Vehicle dynamics C** input signals.

Safe Following Distance

When the **Maintain safe distance between lead vehicle and ego vehicle** parameter is selected, the model predictive controller computes the safe following distance constraint; that is, the minimum relative distance between the lead and ego vehicle, as:

$$D_R = D_S + G_T * V_E$$

Here:

- D_S is the **Default spacing** parameter.
- G_T is the **Time gap** input signal.
- V_E is the **Longitudinal velocity** input signal.

To define a different safe following distance constraint, create a custom path-following control system by, on the **Block** tab, clicking **Create PFC subsystem**.

Initial Conditions

By default, the model predictive controller assumes the following initial conditions for the ego vehicle:

- Longitudinal velocity is equal to the **Initial longitudinal velocity** parameter.
- Longitudinal acceleration is zero.
- Lateral velocity is zero.
- Steering angle is zero.
- Yaw angle rate is zero.

When the **Maintain safe distance between lead vehicle and ego vehicle** parameter is selected, the controller assumes the following additional initial conditions:

- The lead vehicle longitudinal velocity is equal to the **Initial longitudinal velocity** parameter.
- Relative distance between the lead vehicle and ego vehicle is:

$$D_R = D_S + G_T * V_E$$

Here:

- D_S is the **Default spacing** parameter.
- G_T is the time gap and is assumed to be `1.4`.
- V_E is the **Initial longitudinal velocity** parameter.

If the initial conditions in your model do not match these conditions, the **Steering angle** and **Longitudinal acceleration** outputs can exhibit initial bumps at the start of the simulation.

To modify the controller initial conditions to match your simulation, create a custom path-following control system by, on the **Block** tab, clicking **Create PFC subsystem**.

Version History

Introduced in R2019a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
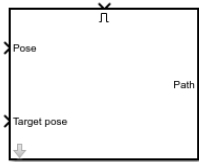
See Also

Blocks
Vehicle Path Planner System | Adaptive Cruise Control System | Lane Keeping Assist System

Topics
"Automated Driving Using Model Predictive Control"

Vehicle Path Planner System

Plan collision-free path from initial pose to target pose

Libraries:
Model Predictive Control Toolbox

Description

The Vehicle Path Planner System block simulates a vehicle path planner (VPP) system that plans a collision-free trajectory from an initial pose to a target pose using a multistage nonlinear MPC controller. You can use this block to plan a path for a vehicle, such as a vehicle entering and exiting a parking spot or a robot moving in a warehouse.

Once you plan a path, you can have the vehicle track the path using a path-following control system.

To customize your controller, for example to modify the controller cost function or change the controller initial conditions, click **Create VPP subsystem**.

Ports

Input

Pose — Initial vehicle pose
three-element row vector

Specify the initial vehicle pose as the vector signal $[x_0 \; y_0 \; \theta_0]$, where:

- (x_0, y_0) is the initial position for the center of the vehicle rear axle.
- θ_0 is the initial vehicle yaw angle.

Target Pose — Target vehicle pose
three-element row vector

Specify the target vehicle pose as the vector signal $[x_t \; y_t \; \theta_t]$, where:

- (x_t, y_t) is the target position for the center of the vehicle rear axle.
- θ_t is the initial vehicle yaw angle.

Enable — Enable planning
logical scalar

To plan a path using the current pose, target pose, and obstacles, enable the path planner by setting this signal to a positive value.

When planning is complete, set the enable signal to zero, which resets the nonlinear MPC controller.

The time for which the path planner block is enabled must be:

- Longer than the worst-case execution time of the planner. To guarantee a worst-case execution time, configure the planning controller to apply a suboptimal solution after a specified maximum number of iterations by selecting the **Use suboptimal solution** parameter.

- Shorter than the sample time of the controller, as specified using the **Sample time** parameter.

Obstacles — List of obstacles
five-column matrix

Specify obstacle information as a matrix with *N* rows, where *N* is equal to the **Number of obstacles** parameter. Each row contains the following information for an obstacle in the order [*X Y Φ L W*]:

- (*X*, *Y*) — Position, specified in meters
- *Φ* — Yaw angle, specified in radians
- *L* — Length, specified in meters
- *W* — Width, specified in meters

Dependencies

To enable this port, select the corresponding **Use external source** parameter.

Velocity range — Velocity range
two-element vector

Vehicle velocity range in meters per second, specified as the two-element row vector [V_{min} V_{max}], where:

- V_{min} is the minimum vehicle velocity.
- V_{max} is the maximum vehicle velocity.

Dependencies

To enable this port, select the corresponding **Use external source** parameter.

Steering angle range — Front steering angle range
two-element vector

Front steering angle range in radians, specified as the two-element row vector [ψ_{min} ψ_{max}], where:

- ψ_{min} is the minimum steering angle.
- ψ_{max} is the maximum steering angle.

The front steering angle is the angle between the front tires and the longitudinal axis of the vehicle. The steering angle is positive towards the positive lateral axis of the ego vehicle.

Dependencies

To enable this port, select the corresponding **Use external source** parameter.

Output

Path — Planned vehicle path
three-column matrix

Planned vehicle path, output as a three column matrix. Each row contains one vehicle pose [*x y θ*] along the vehicle path, where:

- (*x*, *y*) is the position for the center of the vehicle rear axle.
- *θ* is the vehicle yaw angle.

Exitflag — Optimization status
integer

The optimization status of the nonlinear MPC controller, output as an integer as follows:

- Positive — Optimal solution found
- 0 — Maximum number of optimization iterations reached, solution might not be optimal
- Negative — Infeasible optimization problem

Dependencies

To enable this port, select the **Optimization status** parameter.

Parameters

Parameters Tab

Length — Vehicle length
4.7 (default) | nonnegative scalar

Vehicle length, specified in meters.

Programmatic Use
Block Parameter: egoLength
Type: string, character vector
Default: "4.7"

Width — Vehicle width
1.8 (default) | nonnegative scalar

Vehicle width, specified in meters.

Programmatic Use
Block Parameter: egoWidth
Type: string, character vector
Default: "1.8"

Wheelbase — Distance between front and rear axles
2.8 (default) | nonnegative scalar

The distance between the front and rear axles, specified in meters.

Programmatic Use
Block Parameter: wheelbase
Type: string, character vector
Default: "2.8"

Center to rear axle — Distance from center of vehicle to rear axle
1.4 (default) | nonnegative scalar

The longitudinal distance from the center of the vehicle to the rear axle, specified in meters.

Programmatic Use
Block Parameter: `dist2Center`
Type: string, character vector
Default: `"1.4"`

Number of obstacles — Number of obstacles in environment
0 (default) | nonnegative integer

The number of obstacles in the environment.

Programmatic Use
Block Parameter: `numObs`
Type: string, character vector
Default: `"0"`

Obstacle list — Obstacles in environment
`zeros(1,5)` (default) | five-column matrix

Specify obstacle information as a matrix with *N* rows, where *N* is equal to the **Number of obstacles** parameter. Each row contains the following information for an obstacle in the order [*X Y Φ L W*]:

- (*X*, *Y*) — Position, specified in meters
- *Φ* — Yaw angle, specified in radians
- *L* — Length, specified in meters
- *W* — Width, specified in meters

If the obstacle positions and orientations vary over time, add the **Obstacles** input port to the block by selecting **Use external source**.

Dependencies

To enable this parameter, set the **Number of obstacles** parameter to a nonzero value.

Programmatic Use
Block Parameter: `obsMat`
Type: string, character vector
Default: `"zeros(1,5)"`

Programmatic Use: Use external source
Block Parameter: `obs_ext`
Type: string, character vector
Values: `"off"` | `"on"`
Default: `"off"`

Minimum distance from ego vehicle to obstacles — Minimum safe distance between vehicle and any obstacle
0.1 (default) | positive scalar

Minimum safe distance between the ego vehicle and any obstacle in the environment, specified in meters.

Dependencies

To enable this parameter, set the **Number of obstacles** parameter to a nonzero value.

Programmatic Use
Block Parameter: minDist
Type: string, character vector
Default: "0.1"

Sample time — Controller sample time
1 (default) | positive scalar

Controller sample time, specified in seconds.

Programmatic Use
Block Parameter: Ts
Type: string, character vector
Default: "1"

Prediction horizon — Prediction horizon
20 (default) | positive integer

Prediction horizon steps for planning. The total prediction time is the product of the sample time and the prediction horizon.

Programmatic Use
Block Parameter: pPlanning
Type: string, character vector
Default: "20"

Velocity range — Velocity range
[-6.5 6.5] (default) | two-element row vector

Vehicle velocity range in meters per second, specified as the two-element row vector [V_{min} V_{max}], where:

- V_{min} is the minimum vehicle velocity.
- V_{max} is the maximum vehicle velocity.

If the velocity range varies over time, add the **Velocity range** input port to the block by selecting **Use external source**.

Programmatic Use
Block Parameter: v_range
Type: string, character vector
Default: "[-6.5,6.5]"

Programmatic Use: Use external source
Block Parameter: v_ext
Type: string, character vector
Values: "off" | "on"
Default: "off"

Steering angle range — Front steering angle range
[-pi/4 pi/4] | two-element row vector

Front steering angle range in radians, specified as the two-element row vector [ψ_{min} ψ_{max}], where:

- ψ_{min} is the minimum steering angle.

- ψ_{max} is the maximum steering angle.

The front steering angle is the angle of the front tires from the longitudinal axis of the vehicle. The steering angle is positive towards the positive lateral axis of the ego vehicle.

If the steering angle range varies over time, add the **Steering angle range** input port to the block by selecting **Use external source**.

Programmatic Use
Block Parameter: steer_range
Type: string, character vector
Default: "[-pi/4,pi/4]"

Programmatic Use: Use external source
Block Parameter: steer_ext
Type: string, character vector
Values: "off" | "on"
Default: "off"

Block Tab

Use suboptimal solution — Apply suboptimal solution after specified number of iterations
off (default) | on

Configure the controller to apply a suboptimal solution after a specified maximum number of iterations, which guarantees the worst-case execution time for your planning controller. To set the maximum number of iterations, use the **Maximum iteration number** parameter.

For more information, see "Suboptimal QP Solution".

Programmatic Use
Block Parameter: suboptimal
Type: string, character vector
Values: "off" | "on"
Default: "off"

Maximum iteration number — Maximum optimization iterations
200 (default) | positive integer

Maximum number of optimization iterations.

Dependencies

To enable this parameter, select the **Use suboptimal solution** parameter.

Programmatic Use
Block Parameter: maxIter
Type: string, character vector
Default: "200"

Optimization status — Add optimization status output port
off (default) | on

Select this parameter to add the **Exitflag** output port to the block.

Programmatic Use
Block Parameter: output_exitflag

Type: string, character vector
Values: "off" | "on"
Default: "off"

Create VPP subsystem — Create custom planner
button

Generate a custom VPP subsystem, which you can modify for your application. The configuration data for the custom controller is exported to the MATLAB workspace as a structure.

Version History
Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Code generation for single-precision or fixed-point computations is not supported.
- In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, you must select the following parameters.

  - **Support non-finite numbers** — This parameter is selected by default.
  - **Support variable-size signals** when using Embedded Coder — This parameter is not selected by default.

See Also

Blocks
Multistage Nonlinear MPC Controller | Path Following Control System

Topics
"Automated Driving Using Model Predictive Control"