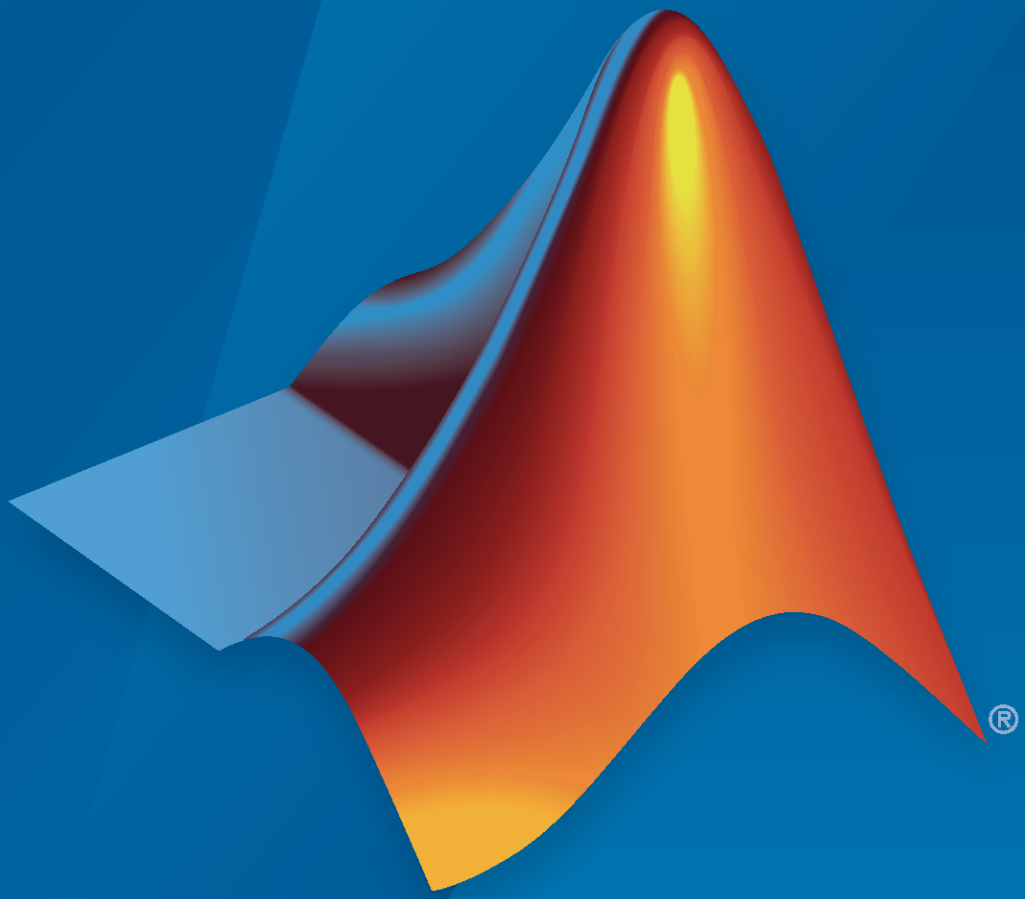
Model Predictive Control Toolbox™

Getting Started Guide

Alberto Bemporad
N. Lawrence Ricker
Manfred Morari



MATLAB®

R2023a

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

Model Predictive Control Toolbox™ Getting Started Guide

Revision History

October 2004	First printing	New for Version 2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.2.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.2.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.2.4 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.1.1 (Release 2009b)
March 2010	Online only	Revised for Version 3.2 (Release 2010a)
September 2010	Online only	Revised for Version 3.2.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.1.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.1.2 (Release 2013a)
September 2013	Online only	Revised for Version 4.1.3 (Release 2013b)
March 2014	Online only	Revised for Version 4.2 (Release R2014a)
October 2014	Online only	Revised for Version 5.0 (Release 2014b)
March 2015	Online only	Revised for Version 5.0.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.1 (Release 2015b)
March 2016	Online only	Revised for Version 5.2 (Release 2016a)
September 2016	Online only	Revised for Version 5.2.1 (Release 2016b)
March 2017	Online only	Revised for Version 5.2.2 (Release 2017a)
September 2017	Online only	Revised for Version 6.0 (Release 2017b)
March 2018	Online only	Revised for Version 6.1 (Release 2018a)
September 2018	Online only	Revised for Version 6.2 (Release 2018b)
March 2019	Online only	Revised for Version 6.3 (Release 2019a)
September 2019	Online only	Revised for Version 6.3.1 (Release 2019b)
March 2020	Online only	Revised for Version 6.4 (Release 2020a)
September 2020	Online only	Revised for Version 7.0 (Release 2020b)
March 2021	Online only	Revised for Version 7.1 (Release 2021a)
September 2021	Online only	Revised for Version 7.2 (Release 2021b)
March 2022	Online only	Revised for Version 7.3 (Release 2022a)
September 2022	Online only	Revised for Version 8.0 (Release 2022b)
March 2023	Online only	Revised for Version 8.1 (Release 2023a)

Contents

1 Introduction

2 Building Models

3 Design MPC Controllers

Introduction

Model Predictive Control Toolbox Product Description

Design and simulate model predictive controllers

Model Predictive Control Toolbox provides functions, an app, Simulink® blocks, and reference examples for developing model predictive control (MPC). For linear problems, the toolbox supports the design of implicit, explicit, adaptive, and gain-scheduled MPC. For nonlinear problems, you can implement single- and multi-stage nonlinear MPC. The toolbox provides deployable optimization solvers and also enables you to use a custom solver.

You can evaluate controller performance in MATLAB® and Simulink by running closed-loop simulations. For automated driving, you can also use the provided MISRA C™- and ISO 26262-compliant blocks and examples to quickly get started with lane keep assist, path planning, path following, and adaptive cruise control applications.

The toolbox supports C and CUDA® code and IEC 61131-3 Structured Text generation.

What is Model Predictive Control?

Model predictive control (MPC) is an optimal control technique in which the calculated control actions minimize a cost function for a constrained dynamical system over a finite, receding, horizon.

At each time step, an MPC controller receives or estimates the current state of the plant. It then calculates the sequence of control actions that minimizes the cost over the horizon by solving a constrained optimization problem that relies on an internal plant model and depends on the current system state. The controller then applies to the plant only the first computed control action, disregarding the following ones. In the following time step the process repeats.

MPC Basic Control Loop

When the cost function is quadratic, the plant is linear and without constraints, and the horizon tends to infinity, MPC is equivalent to linear-quadratic regulator (LQR) control, or linear-quadratic Gaussian (LQG) control if a Kalman filter estimates the plant state from its inputs and outputs.

In practice, despite the finite horizon, MPC often inherits many useful characteristics of traditional optimal control, such as the ability to naturally handle multi-input multi-output (MIMO) plants, the capability of dealing with time delays (possibly of different durations in different channels), and built-in robustness properties against modeling errors. Nominal stability can also be guaranteed by using specific terminal constraints. Other additional important MPC features are its ability to explicitly handle constraints and the possibility of making use of information on future reference and disturbance signals, when available.

For an introduction on the subject, see first two books in the bibliography sections. For an explanation of the controller internal model and its estimator, see “MPC Prediction Models” on page 2-3 and “Controller State Estimation”, respectively. For an overview of the optimization problem, see “Optimization Problem”. For more information on the solvers, see “QP Solvers”.

Solving a constrained optimal control online at each time step can require substantial computational resources. However in some cases, such as for linear constrained plants, you can precompute and store the control law across the entire state space rather than solve the optimization in real time. This approach is known as explicit MPC.

MPC Design Workflow

In the simplest case (also known as traditional, or linear, MPC), in which both plant and constraints are linear and the cost function is quadratic, the general workflow to develop an MPC controller includes the following steps.

1. **Specify plant** — Define the internal plant model that the MPC controller uses to forecast plant behavior across the prediction horizon. Typically, you obtain this plant model by linearizing a nonlinear plant at a given operating point and specifying it as an LTI object, such as `ss`, `tf`, and `zpk`. You can also identify a plant using System Identification Toolbox™ software. Note that one limitation is that the plant cannot have a direct feedthrough between its control input and any output. For more information on this step, see “Construct Linear Time Invariant Models” on page 2-15, “Specify Multi-Input Multi-Output Plants” on page 2-20, “Linearize Simulink Models” on page 2-22, “Linearize Simulink Models Using MPC Designer” on page 2-31, and “Identify Plant from Data” on page 2-49.

2. **Define signal types** — For MPC design purposes, plant signals are usually categorized into different input and output types. You typically use `setmpcsignals` to specify, in the plant object defined in the previous step, whether each plant output is measured or unmeasured, and whether each plant input is a manipulated variable (that is, a control input) or a measured or unmeasured disturbance. Alternatively, you can specify signal types in **MPC Designer**. For more information, see “MPC Signal Types” on page 2-2.
3. **Create MPC object** — After specifying the signal types in the plant object, you create an `mpc` object in the MATLAB workspace (or in the **MPC Designer**), and specify, in the object, controller parameters such as the sample time, prediction and control horizons, cost function weights, constraints, and disturbance models. The following is an overview of the most important parameters that you need to select.
   a. Sample time — A typical starting guess consists of setting the controller sample time so that 10 to 20 samples cover the rise time of the plant.
   b. Prediction horizon — The number of future samples over which the controller tries to minimize the cost. It should be long enough to capture the transient response and cover the significant dynamics of the system. A longer horizon increases both performance and computational requirements. A typical prediction horizon is 10 to 20 samples.
   c. Control horizon — The number of free control moves that the controller uses to minimize the cost over the prediction horizon. Similarly to the prediction horizon, a longer control horizon increases both performance and computational requirements. A good rule of thumb for the control horizon is to set it from 10% to 20% of the prediction horizon while having a minimum of two to three steps. For more information on sample time and horizon, see “Choose Sample Time and Horizons”.
   d. Nominal Values — If your plant is derived from the linearization of a nonlinear model around an operating point, a good practice is to set the nominal values for input, state, state derivative (if nonzero), and output. Doing so allows you to specify constraints on the actual inputs and outputs (instead of doing so on the deviations from their nominal values), and allows you to simulate the closed loop and visualize signals more easily when using Simulink or the `sim` command.
   e. Scale factors — Good practice is to specify scale factors for each plant input and output, especially when their range and magnitude is very different. Appropriate scale factors improve the numerical condition of the underlying optimization problem and make weight tuning easier. A good recommendation is to set a scale factor approximatively equal to the span (the difference between the maximum and minimum value in engineering units) of the related signal. For more information, see “Specify Scale Factors”.
   f. Constraints — Constraints typically reflect physical limits. You can specify constraints as either hard (cannot be violated in the optimization) or soft (can be violated to a small extent). A good recommendation is to set hard constraints, if necessary, on the inputs or their rate of change, while setting output constraints, if necessary, as soft. Setting hard constraints on both input and outputs can lead to infeasibility and is in general not recommended. For more information, see “Specify Constraints”.
   g. Weights — You can prioritize the performance goals of your controller by adjusting the cost function tuning weights. Typically, larger output weights provide aggressive reference tracking performance, while larger weights on the manipulated variable rates promote smoother control moves that improve robustness. For more information, see “Tune Weights”.
   h. Disturbance and noise models — The internal prediction model that the controller uses to calculate the control action typically consists of the plant model augmented with models for disturbances and measurement noise affecting the plant. Disturbance models specify the dynamic characteristics of the unmeasured disturbances on the inputs and outputs,

respectively, so they can be better rejected. By default, these disturbance models are assumed to be integrators (therefore allowing the controller to reject step-like disturbances) unless you specify them otherwise. Measurement noise is typically assumed to be white. For more information on plant and disturbance models see “MPC Prediction Models” on page 2-3, and “Adjust Disturbance and Noise Models”.

After creating the `mpc` object, good practice is to use functions such as `cloffset` to calculate the closed loop steady state output sensitivities, therefore checking whether the controller can reject constant output disturbances. The more general `review` also inspects the object for potential problems. To perform a deeper sensitivity and robustness analysis for the time frames in which you expect no constraint to be active, you can also convert the unconstrained controller to an LTI system object using `ss`, `zpk`, or `tf`. For related examples, see “Review Model Predictive Controller for Stability and Robustness Issues”, “Test MPC Controller Robustness Using MPC Designer”, “Compute Steady-State Output Sensitivity Gain”, and “Extract Controller”.

Note that many of the recommended parameter choices are incorporated in the default values of the `mpc` object; however, since each of these parameter is normally the result of several problem-dependent trade offs, you have to select the parameters that make sense for your particular plant and requirements.

4 **Simulate closed loop** — After you create an MPC controller, you typically evaluate the performance of your controller by simulating it in closed loop with your plant using one of the following options.

- Using MATLAB, you can simulate the closed loop using `sim` (more convenient for linear plant models) or `mpcmove` (more flexible, allowing for more general discrete time plants or disturbance signals and for a custom state estimator).
- Using Simulink, you can use the MPC Controller block (which takes your `mpc` object as a parameter) in closed loop with your plant model built in Simulink. This option allows for the greatest flexibility in simulating more complex systems and for easy generation of production code from your controller.
- Using **MPC Designer**, you can simulate the linear closed loop response while at the same time tuning the controller parameters.

Note that any of these options allows you to also simulate model mismatches (cases in which the actual plant is slightly different from the internal plant model that the controller uses for prediction). For a related example, see “Simulating MPC Controller with Plant Model Mismatch”. When reference and measured disturbances are known ahead of time, MPC can use this information (also known as look-ahead, or previewing) to improve the controller performance. See “Signal Previewing” for more information and “Improving Control Performance with Look-Ahead (Previewing)” for a related example. Similarly, you can specify tuning weights and constraints that vary over the prediction horizon. For related examples, see “Update Constraints at Run Time”, “Vary Input and Output Bounds at Run Time”, “Tune Weights at Run Time”, and “Adjust Horizons at Run Time”.

5 **Refine design** — After an initial evaluation of the closed loop you typically need to refine the design by adjusting the controller parameters and evaluating different simulation scenarios. In addition to the parameters described in step 3, you can consider:

- Using manipulated variable blocking. For more information, see “Manipulated Variable Blocking”.
- For over-actuated systems, setting reference targets for the manipulated variables. For a related example, see “Setting Targets for Manipulated Variables”.

- Tuning the gains of the Kalman state estimator (or designing a custom state estimator). For more information and related examples, see “Controller State Estimation”, “Custom State Estimation”, and “Implement Custom State Estimator Equivalent to Built-In Kalman Filter”.
- Specifying terminal constraints. For more information and a related example, see “Terminal Weights and Constraints” and “Provide LQR Performance Using Terminal Penalty Weights”.
- Specifying custom constraints. For related examples, see “Constraints on Linear Combinations of Inputs and Outputs” and “Use Custom Constraints in Blending Process”.
- Specifying off-diagonal cost function weights. For an example, see “Specifying Alternative Cost Function with Off-Diagonal Weight Matrices”.

6 **Speed up execution** — See “MPC Controller Deployment” on page 1-8.

7 **Deploy controller** — See “MPC Controller Deployment” on page 1-8.

Control Nonlinear and Time-Varying Plants

Often the plant to be controlled can be accurately approximated by a linear plant only locally, around a given operating point. This approximation might no longer be accurate as time passes and the plant operating point changes.

You can use several approaches to deal with these cases, from the simpler to more general and complicated.

1 Adaptive MPC — If the order (and the number of time delays) of the plant does not change, you can design a single MPC controller (for example for the initial operating point), and then at run-time you can update the controller prediction model at each time step (while the controller still assumes that the prediction model stays constant in the future, across its prediction horizon).

   Note that while this approach is the simplest, it requires you to continuously (that is, at each time step) calculate the linearized plant that has to be supplied to the controller. You can do so in three main ways.

   a If you have a reliable plant model, you can extract the local linear plant online by linearizing the equations, assuming this process is not too computationally expensive. If you have simple symbolic equations for your plant model, you might be able to derive, offline, a symbolic expression of the linearized plant matrices at any given operating condition. Online, you can then calculate these matrices and supply them to the adaptive MPC controller without having to perform a numerical linearization at each time step. For an example using this strategy, see “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization”.

   b Alternatively, you can extract an array of linearized plant models offline, covering the relevant regions of the state-input space, and then online you can use a linear parameter-varying (LPV) plant that obtains, by interpolation, the linear plant at the current operating point. For an example using this strategy, see “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter-Varying System”.

   c If the plant is not accurately represented by a mathematical model, but you can assume a known structure with some estimates of its parameters, stability, and a minimal amount of input noise, you can use the past plant inputs and outputs to estimate a model of the plant online, although this can be somewhat computationally intensive. For an example using this strategy, see “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation”.

This approach requires an `mpc` object and either the `mpcmoveAdaptive` function or the Adaptive MPC Controller block. For more information, see “Adaptive MPC” and “Model Updating Strategy”.

2. Linear Time Varying MPC — This approach is a kind of adaptive MPC in which the controller knows in advance how its internal plant model changes in the future, and therefore uses this information when calculating the optimal control across the prediction horizon. Here, at every time step, you supply to the controller not only the current plant model but also the plant models for all the future steps, across the whole prediction horizon. To calculate the plant models for the future steps, you can use the manipulated variables and plant states predicted by the MPC controller at each step as operating points around which a nonlinear plant model can be linearized.

   This approach is particularly useful when the plant model changes considerably (but predictably) within the prediction horizon. It requires an `mpc` object and using `mpcmoveAdaptive` or the Adaptive MPC Controller block. For more information, see “Time-Varying MPC”.

3. Gain-Scheduled MPC — In this approach you design multiple MPC controllers offline, one for each relevant operating point. Then, online, you switch the active controller as the plant operating point changes. While switching the controller is computationally simple, this approach requires more online memory (and in general more design effort) than adaptive MPC. It should be reserved for cases in which the linearized plant models have different orders or time delays (and the switching variable changes slowly, with respect to the plant dynamics). To use gain-scheduled MPC. you create an array of `mpc` objects and then use the `mpcmoveMultiple` function or the Multiple MPC Controllers block for simulation. For more information, see “Gain-Scheduled MPC”. For an example, see “Gain-Scheduled MPC Control of Nonlinear Chemical Reactor”.

4. Nonlinear MPC — You can use this strategy to control highly nonlinear plants when all the previous approaches are unsuitable, or when you need to use nonlinear constraints or non-quadratic cost functions. This approach is more computationally intensive than the previous ones, and it also requires you to design an implement a nonlinear state estimator if the plant state is not completely available. Two nonlinear MPC approaches are available.

   a. Multistage Nonlinear MPC — For a multistage MPC controller, each future step in the horizon (stage) has its own decision variables and parameters, as well as its own nonlinear cost and constraints. Crucially, cost and constraint functions at a specific stage are functions only of the decision variables and parameters at that stage. While specifying multiple costs and constraint functions can require more design time, it also allows for an efficient formulation of the underlying optimization problem and a smaller data structure, which significantly reduces computation times compared to generic nonlinear MPC. Use this approach if your nonlinear MPC problem has cost and constraint functions that do not involve cross-stage terms, as is often the case. To use multistage nonlinear MPC you need to create an `nlmpcMultistage` object, and then use the `nlmpcmove` function or the Multistage Nonlinear MPC Controller block for simulation. For more information, see “Multistage Nonlinear MPC”.

   b. Generic Nonlinear MPC — This method is the most general, and computationally expensive, form of MPC. As it explicitly provides standard weights and linear bounds settings, it can be a good starting point for a design where the only nonlinearity comes from the plant model. Furthermore, you can use the `RunAsLinearMPC` option in the `nlmpc` object to evaluate whether linear, time varying, or adaptive MPC can achieve the same performance. If so, use these design options, and possibly evaluate gain scheduled MPC; otherwise, consider multistage nonlinear MPC. Use generic nonlinear MPC only as an initial design, or when all the previous design options are not viable. To use generic nonlinear MPC, you need to create

an `nlmpc` object, and then use the `nlmpcmove` function or the Nonlinear MPC Controller block for simulation. For more information, see “Generic Nonlinear MPC”.

MPC Controller Deployment

When you are satisfied with the simulation performance of your controller design, you typically look for ways to speed up the execution, in an effort to both optimize the design for future simulations and to meet the stricter computational requirements of embedded applications.

You can use several strategies to improve the computational performance of MPC controllers.

1. Try to increase the sample time — The sampling frequency must be high enough to cover the significant bandwidth of the system. However, if the sample time is too small, not only do you reduce the available computation time for the controller but you must also use a larger prediction horizon to cover the system response, which increases computational requirements.
2. Try to shorten prediction and control horizons — Since both horizons directly impact the total number of decision variables and constraints in the resulting optimization problem, they heavily affect both memory usage and the number of required calculations. Therefore, check whether you can obtain similar tracking responses and robustness to uncertainties with shorter horizons. Note that sample time plays a role too. The sampling frequency needs to be high enough (equivalently the sample time small enough) to cover the significant bandwidth of the system. However, if the sample time is too small, not only you have a shorter available execution time on the hardware, but you also need a larger number of prediction steps to cover the system response, which results in a more computationally expensive optimization problem to be solved at each time step.
3. Use varying parameters only when needed — Normally Model Predictive Control Toolbox allows you to vary some parameters (such as weights or constraints coefficients) at run-rime. While this capability is useful in some cases, it considerably increases the complexity of the software. Therefore, unless specifically needed, for deployment, consider explicitly specifying such parameters as constants, thus preventing the possibility of changing them online. For related examples, see “Update Constraints at Run Time”, “Vary Input and Output Bounds at Run Time”, “Tune Weights at Run Time”, and “Adjust Horizons at Run Time”.
4. Limit the maximum number of iterations that your controller can use to solve the quadratic optimization problem, and configure it to use the current suboptimal solution when the maximum number of iterations is reached. Using a suboptimal solution shortens the time needed by the controller to calculate the control action, and in some cases it does not significantly decrease performance. In any case, since the number of iterations can change dramatically from one control interval to the next, for real time applications, it is recommended to limit the maximum number of iterations. Doing so helps ensuring that the worst-case execution time does not exceed the total computation time allowed on the hardware platform, which is determined by the controller sample time. For a related example, see “Use Suboptimal Solution in Fast MPC Applications”.
5. Tune the solver and its options — The default Model Predictive Control Toolbox solver is a "dense," "active set" solver based on the KWIK algorithm, and it typically performs well in many cases. However, if the total number of manipulated variables, outputs, and constraints across the whole horizon is large, you might consider using an interior point solver. If the internal plant is highly open-loop unstable, consider using a sparse solver. For an overview of the optimization problem, see “Optimization Problem”. For more information on the solvers, see “QP Solvers” and “Configure Optimization Solver for Nonlinear MPC”. For related examples, see “Simulate MPC Controller with a Custom QP Solver” and “Optimizing Tuberculosis Treatment Using Nonlinear MPC with a Custom Solver”.

For application with extremely fast sample time, consider using explicit MPC. It can be proven that the solution to the linear MPC problem (quadratic cost function, linear plant and constraints) is piecewise affine (PWA) on polyhedra. In other words, the constraints divide the state space into polyhedral "critical" regions in which the optimal control action is an affine (linear plus a constant) function of the state. The idea behind explicit MPC is to precalculate, offline and once for all, these functions of the state for every region. These functions can then be stored in your controller. At run time, the controller then selects and applies the appropriate state feedback law, depending on the critical region that the current operating point is in. Since explicit MPC controllers do not solve an optimization problem online, they require much fewer computations and are therefore useful for applications requiring small sample times. On the other hand, they also have a much larger memory footprint. Indeed, excessive memory requirements can render this approach no longer viable for medium to large problems. Also, since explicit MPC pre-computes the controller offline, it does not support runtime updates of parameters such as weights, constraints or horizons.

To use explicit MPC, you need to generate an `explicitMPC` object from an existing `mpc` object and then use the `mpcmoveExplicit` function or the Explicit MPC Controller block for simulation. For more information, see “Explicit MPC”.

A final option to consider to improve computational performance of both implicit and explicit MPC is to simplify the problem. Some parameters, such as the number of constraints and the number state variables, greatly increase the complexity of the resulting optimization problem. Therefore, if the previous options are not satisfying, consider retuning these parameters (and potentially use a simpler lower-fidelity prediction model) to simplify the problem.

Once you are satisfied with the computational performance of your design, you can generate code for deployment to real-time applications from MATLAB or Simulink. For more information, see “Generate Code and Deploy Controller to Real-Time Targets”.

Acknowledgments

MathWorks would like to acknowledge the following contributors to Model Predictive Control Toolbox.

Alberto Bemporad

Professor of Control Systems, IMT Institute for Advanced Studies Lucca, Italy. Research interests include model predictive control, hybrid systems, optimization algorithms, and applications to automotive, aerospace, and energy systems. Fellow of the IEEE®. Author of the Model Predictive Control Simulink library and commands.

N. Lawrence Ricker

Professor of Chemical Engineering, University of Washington, Seattle, USA. Research interests include model predictive control and process optimization. Author of the Model Predictive Control Simulink library and commands.

Manfred Morari

Professor at the Automatic Control Laboratory and former Head of Department of Information Technology and Electrical Engineering, ETH Zurich, Switzerland. Research interests include model predictive control, hybrid systems, and robust control. Fellow of the IEEE, AIChE, and IFAC. Co-author of the first version of the toolbox.

Bibliography

[1] Allgower, F., and A. Zheng, *Nonlinear Model Predictive Control*, Springer-Verlag, 2000.

[2] Maciejowski, J. M., *Predictive Control with Constraints*, Pearson Education POD, 2002.

[3] Kouvaritakis, B., and M. Cannon, *Non-Linear Predictive Control: Theory & Practice*, IEE Publishing, 2001.

[4] Prett, D., and C. Garcia, *Fundamental Process Control*, Butterworths, 1988.

[5] Camacho, E. F., and C. Bordons, *Model Predictive Control*, Springer-Verlag, 1999.

[6] Rossiter, J. A., *Model-Based Predictive Control: A Practical Approach*, CRC Press, 2003.

[7] Borrelli, F., A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*, Cambridge University Press, 2017.

Building Models

MPC Signal Types

For MPC design purposes, signals are usually categorized into different input and output types. Typically you use `setmpcsignals` to specify, in your plant model, whether an input or output signal belongs to one of the following categories.

Inputs

The *plant inputs* are the independent variables affecting the plant. As shown in “MPC Prediction Models” on page 2-3, there are three types:

Measured disturbances

The controller cannot adjust them, but uses them for feedforward compensation.

Manipulated variables

The controller adjusts these in order to achieve its goals.

Unmeasured disturbances

These are independent inputs of which the controller has no direct knowledge, and for which it must compensate.

Outputs

The *plant outputs* are the dependent variables (outcomes) you wish to control or monitor. As shown in “MPC Prediction Models” on page 2-3, there are two types:

Measured outputs

The controller uses these to estimate unmeasured quantities and as feedback on the success of its adjustments.

Unmeasured outputs

The controller estimates these based on available measurements and the plant model. The controller can also hold unmeasured outputs at setpoints or within constraint boundaries.

See Also

Functions
`setmpcsignals`

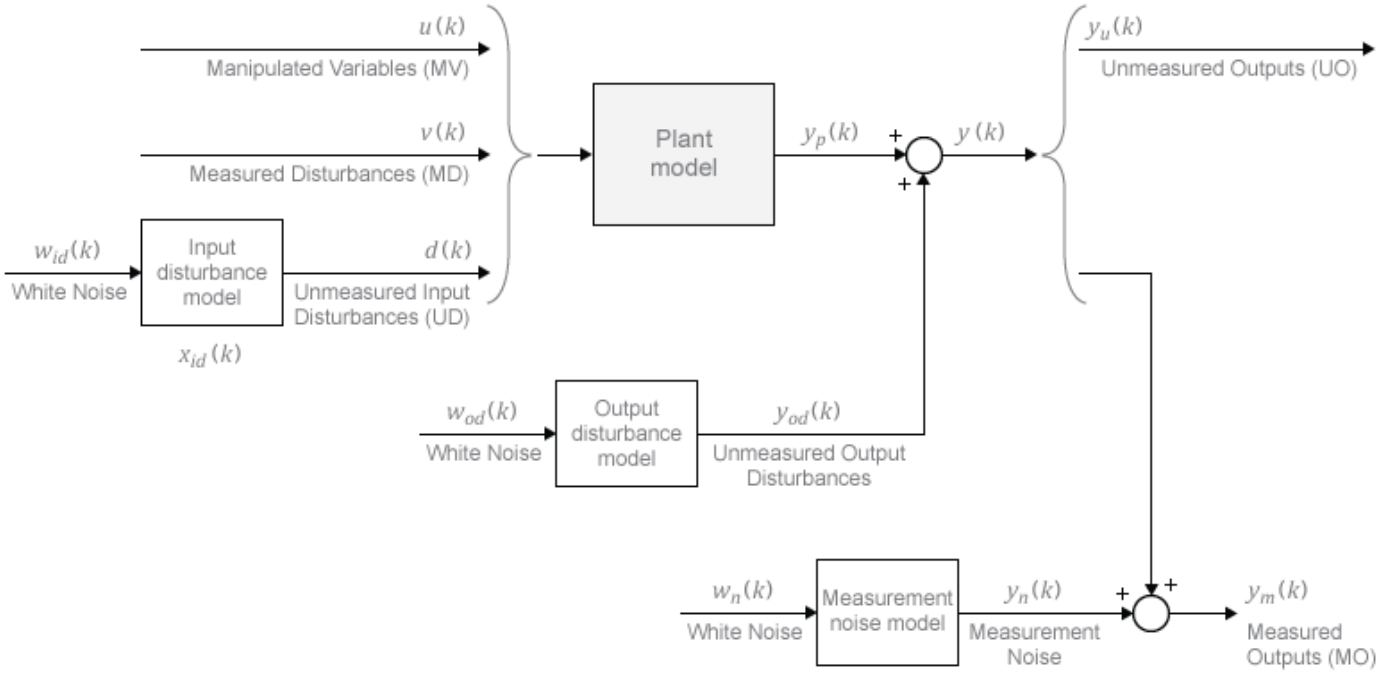
Objects
`mpc`

More About

- “MPC Prediction Models” on page 2-3
- “What is Model Predictive Control?” on page 1-3

MPC Prediction Models

Model predictive controllers use plant, disturbance, and noise models for prediction and state estimation. The different signal types are described in “MPC Signal Types” on page 2-2. The model structure used in an MPC controller appears in the following illustration.



Plant Model

You can specify the plant model in one of the following linear-time-invariant (LTI) formats:

- Numeric LTI models — Transfer function (`tf`), state space (`ss`), zero-pole-gain (`zpk`)
- Identified models (requires System Identification Toolbox) — `idss`, `idtf`, `idproc`, and `idpoly`

The MPC controller performs all estimation and optimization calculations using a discrete-time, delay-free, state-space system with dimensionless input and output variables. Therefore, when you specify a plant model in the MPC controller, the software performs the following, if needed:

1. Conversion to state space — The `ss` command converts the supplied model to an LTI state-space model.
2. Discretization or resampling — If the model sample time differs from the MPC controller sample time (defined in the `Ts` property), one of the following occurs:
   - If the model is continuous time, the `c2d` command converts it to a discrete-time LTI object using the controller sample time.
   - If the model is discrete time, the `d2d` command resamples it to generate a discrete-time LTI object using the controller sample time.
3. Delay removal — If the discrete-time model includes any input, output, or internal delays, the `absorbDelay` command replaces them with the appropriate number of poles at $z = 0$, increasing

the total number of discrete states. The `InputDelay`, `OutputDelay`, and `InternalDelay` properties of the resulting state-space model are all zero.

4 Conversion to dimensionless input and output variables — The MPC controller enables you to specify a scale factor for each plant input and output variable. If you do not specify scale factors, they default to `1`. The software converts the plant input and output variables to dimensionless form as follows:

$$x_p(k+1) = A_p x_p(k) + BS_i u_p(k)$$
$$y_p(k) = S_o^{-1} C x_p(k) + S_o^{-1} DS_i u_p(k)\,.$$

where A_p, B, C, and D are the constant zero-delay state-space matrices from step 3, and:

- S_i is a diagonal matrix of input scale factors in engineering units.
- S_o is a diagonal matrix of output scale factors in engineering units.
- x_p is the state vector from step 3 in engineering units (including any absorbed delay states). No scaling is performed on state variables.
- u_p is a vector of dimensionless plant input variables, including manipulated variables, measured disturbances, and unmeasured input disturbances.
- y_p is a vector of dimensionless plant output variables.

The resulting plant model has the following equivalent form:

$$x_p(k+1) = A_p x_p(k) + B_{pu} u(k) + B_{pv} v(k) + B_{pd} d(k)$$
$$y_p(k) = C_p x_p(k) + D_{pu} u(k) + D_{pv} v(k) + D_{pd} d(k)\,.$$

Here, $C_p = S_o^{-1} C$, B_{pu}, B_{pv}, and B_{pd} are the corresponding columns of BS_i. Also, D_{pu}, D_{pv}, and D_{pd} are the corresponding columns of $S_o^{-1} DS_i$. Finally, $u(k)$, $v(k)$, and $d(k)$ are the dimensionless manipulated variables, measured disturbances, and unmeasured input disturbances, respectively.

The MPC controller enforces the restriction of $D_{pu} = 0$, which means that the controller does not allow direct feedthrough from any manipulated variable to any plant output.

Input Disturbance Model

If your plant model includes unmeasured input disturbances, $d(k)$, the input disturbance model specifies the signal type and characteristics of $d(k)$. See “Controller State Estimation” for more information about the model.

The `getindist` command provides access to the model in use.

The input disturbance model is a key factor that influences the following controller performance attributes:

- Dynamic response to apparent disturbances — The character of the controller response when the measured plant output deviates from its predicted trajectory, due to an unknown disturbance or modeling error.
- Asymptotic rejection of sustained disturbances — If the disturbance model predicts a sustained disturbance, controller adjustments continue until the plant output returns to its desired trajectory, emulating a classical integral feedback controller.

You can provide the input disturbance model as an LTI state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) object using `setindist`. The MPC controller converts the input disturbance model to a discrete-time, delay-free, LTI state-space system using the same steps used to convert the plant model on page 2-3. The result is:

$$x_{id}(k+1) = A_{id}x_{id}(k) + B_{id}w_{id}(k)$$
$$d(k) = C_{id}x_{id}(k) + D_{id}w_{id}(k)\,.$$

where A_{id}, B_{id}, C_{id}, and D_{id} are constant state-space matrices, and:

- $x_{id}(k)$ is a vector of $n_{xid} \geq 0$ input disturbance model states.
- $d_k(k)$ is a vector of n_d dimensionless unmeasured input disturbances.
- $w_{id}(k)$ is a vector of $n_{id} \geq 1$ dimensionless white noise inputs, assumed to have zero mean and unit variance.

If you do not provide an input disturbance model, then the controller uses a default model, which has integrators with dimensionless unity gain added to its outputs. An integrator is added for each unmeasured input disturbance, unless doing so would cause a violation of state observability. In this case, a static system with dimensionless unity gain is used instead.

Output Disturbance Model

The output disturbance model is a special case of the more general input disturbance model. Its output, $y_{od}(k)$, is directly added to the plant output rather than affecting the plant states. The output disturbance model specifies the signal type and characteristics of $y_{od}(k)$, and it is often used in practice. See “Controller State Estimation” for more details about the model.

The `getoutdist` command provides access to the output disturbance model in use.

You can specify a custom output disturbance model as an LTI state-space (`ss`), transfer function (`tf`), or zero-pole-gain (`zpk`) object using `setoutdist`. Using the same steps as for the plant model on page 2-3, the MPC controller converts the specified output disturbance model to a discrete-time, delay-free, LTI state-space system. The result is:

$$x_{od}(k+1) = A_{od}x_{od}(k) + B_{od}w_{od}(k)$$
$$y_{od}(k) = C_{od}x_{od}(k) + D_{od}w_{od}(k)\,.$$

where A_{od}, B_{od}, C_{od}, and D_{od} are constant state-space matrices, and:

- $x_{od}(k)$ is a vector of $n_{xod} \geq 1$ output disturbance model states.
- $y_{od}(k)$ is a vector of n_y dimensionless output disturbances to be added to the dimensionless plant outputs.
- $w_{od}(k)$ is a vector of n_{od} dimensionless white noise inputs, assumed to have zero mean and unit variance.

If you do not specify an output disturbance model, then the controller uses a default model, which has integrators with dimensionless unity gain added to some or all of its outputs. These integrators are added according to the following rules:

- No disturbances are estimated, that is no integrators are added, for unmeasured plant outputs.
- An integrator is added for each measured output in order of decreasing output weight.

- For time-varying weights, the sum of the absolute values over time is considered for each output channel.
- For equal output weights, the order within the output vector is followed.

- For each measured output, an integrator is not added if doing so would cause a violation of state observability. Instead, a gain with a value of zero is used instead.

If there is an input disturbance model, then the controller adds any default integrators to that model before constructing the default output disturbance model.

Measurement Noise Model

One controller design objective is to distinguish disturbances, which require a response, from measurement noise, which should be ignored. The measurement noise model specifies the expected noise type and characteristics. See “Controller State Estimation” for more details about the model.

Using the same steps as for the plant model on page 2-3, the MPC controller converts the measurement noise model to a discrete-time, delay-free, LTI state-space system. The result is:

$$x_n(k+1) = A_n x_n(k) + B_n w_n(k)$$
$$y_n(k) = C_n x_n(k) + D_n w_n(k)\,.$$

Here, A_n, B_n, C_n, and D_n are constant state space matrices, and:

- $x_n(k)$ is a vector of $n_{xn} \geq 0$ noise model states.
- $y_n(k)$ is a vector of n_{ym} dimensionless noise signals to be added to the dimensionless measured plant outputs.
- $w_n(k)$ is a vector of $n_n \geq 1$ dimensionless white noise inputs, assumed to have zero mean and unit variance.

If you do not supply a noise model, the default is a unity static gain: $n_{xn} = 0$, D_n is an n_{ym}-by-n_{ym} identity matrix, and A_n, B_n, and C_n are empty.

For an `mpc` controller object, `mpcobj`, the property `mpcobj.Model.Noise` provides access to the measurement noise model.

Note If the minimum eigenvalue of $D_n D_n^T$ is less than $1\text{x}10^{-8}$, the MPC controller adds $1\text{x}10^{-4}$ to each diagonal element of D_n. This adjustment makes a successful default Kalman gain calculation more likely.

See Also

Functions
`setmpcsignals`

Objects
`mpc`

Related Examples

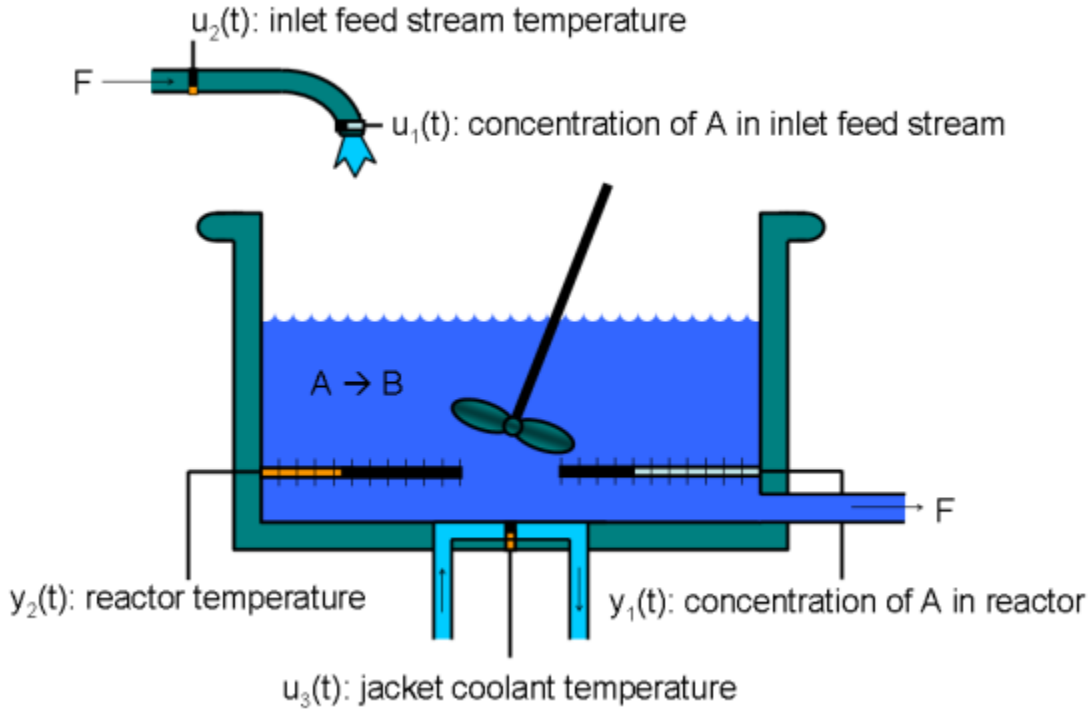
- “Adjust Disturbance and Noise Models”

More About

- “MPC Signal Types” on page 2-2
- “Controller State Estimation”

CSTR Model

The adiabatic continuous stirred tank reactor (CSTR) is a common chemical system in the process industry, and it is described extensively in [1]. A single first-order exothermic and irreversible reaction, A → B, takes place in the vessel, which is assumed to be always perfectly mixed. The inlet stream of reagent A enters the tank at a constant volumetric rate. The product stream B exits continuously at the same volumetric rate, and liquid density is constant. Thus, the volume of reacting liquid is constant. The following figure shows a schematic diagram of the vessel and the surrounding cooling jacket.



The inputs of the CSTR model are arranged in the vector $u(t)$ and are as follows.

- u_1 — C_{Af}, the concentration of reagent A in the inlet feed stream, measured in kmol/m^3
- u_2 — T_f, the temperature of the inlet feed stream, measured in K
- u_3 — T_c, the temperature of the jacket coolant, measured in K

The first two inputs (concentration and temperature of the inlet reagent feed stream, sometimes also indicated as C_{Ai} and T_i, respectively) are normally assumed to be constant unmeasured disturbances, while the third (temperature of the coolant) is the control input used to control the process. Note that the diagram is a simplified sketch; in reality the coolant flow surrounds the whole reactor jacket, and not just the bottom of it.

The states of the model are arranged in the vector $x(t)$.

- x_1 — C_A, the concentration of reagent A in the reactor, measured in kmol/m^3
- x_2 — T, the temperature in the reactor, measured in K

Nonlinear Model

The CSTR system is modeled using basic mass balance and energy conservation principles. The change of the concentration of reagent A in the vessel per time unit can be modeled as follows.

$$\frac{dC_A}{dt} = \frac{F}{V}\left(C_{Af}(t) - C_A(t)\right) - r(t)$$

The first term, where V is the reactor volume and F is the volumetric flow rate, expresses the concentration difference between the inlet and the stream. The second term is the reaction rate per unit of volume, and it is described by the Arrhenius rate law, as follows.

$$r(t) = k_0 e^{\frac{-E}{RT(t)}} C_A(t)$$

Here:

- E is the activation energy.
- R is the Boltzmann ideal gas constant.
- T is the temperature in the reactor.
- k_0 is an unknown nonthermal constant.

The rate law states that the reaction rate increases exponentially with the absolute temperature.

Similarly, using the energy balance principle, and assuming constant volume in the reactor, the temperature change per unit of time can be modeled as follows.

$$\frac{dT(t)}{dt} = \frac{F}{V}\left(T_f(t) - T(t)\right) - \frac{\Delta H}{\rho C_p} r(t) - \frac{UA}{\rho C_p V}(T(t) - T_c(t))$$

Here, the first and third terms describe changes due to the inlet feed stream temperature T_f and jacket coolant temperature T_c, respectively. The second term represents the influence on the reactor temperature caused by the chemical reaction in the vessel.

In this equation:

- ΔH is the heat of the reaction, per mole.
- C_p is a heat capacity coefficient.
- ρ is a density coefficient.
- U is an overall heat transfer coefficient.
- A is the area for the heat exchange (coolant/vessel interface area).

A Simulink representation of this nonlinear reactor model is available in the models `mpc_cstr_plant`, `CSTR_OpenLoop`, and `CSTR_INOUT`. It is used in several examples illustrating how to linearize nonlinear models and how to use linear, adaptive, gain-scheduled, and nonlinear MPC to control a nonlinear plant.

Parameters of the Nonlinear CSTR Simulink Model

Parameter	Value	Unit	Description
F	1	m^3/h	Volumetric flow rate
V	1	m^3	Reactor volume
R	1.985875	kcal/(kmol·K)	Boltzmann's ideal gas constant
ΔH	-5,960	kcal/kmol	Heat of reaction per mole
E	11,843	kcal/kmol	Activation energy per mole
k_0	34,930,800	1/h	Pre-exponential nonthermal factor
ρC_p	500	kcal/(m^3·K)	Density multiplied by heat capacity
UA	150	kcal/(K·h)	Overall heat transfer coefficient multiplied by tank area

In the model, the initial value of C_A is 8.5698 kmol/m^3 and the initial value for T is 311.2639 K. This operating point is an equilibrium when the inflow feed concentration C_{Af} is 10 kmol/m^3, the inflow feed temperature T_f is 300 K, and the coolant temperature T_c is 292 K.

In the example “Non-Adiabatic Continuous Stirred Tank Reactor: MATLAB File Modeling with Simulations in Simulink” (System Identification Toolbox), you use the above equations to estimate the last four parameters when the disturbance inputs C_{Af} and T_f stay around to 10 kmol/m^3 and 298 K, respectively, and the control input T_c ranges from 273 to 322 K. The first state variable, C_A, ranges from 0 to 10 kmol/m^3 and the second one, T, ranges from 310 to 390 K. The values of the last four parameters are estimated to be 11,854, 35,588,869, 500.7095, and 150.1275, respectively, with the same units as in the table.

Linear Model

A linearized model of the CSTR, in which T_f does not deviate from its nominal condition, can be represented by the following linear differential equations.

$$\frac{dC'_A}{dt} = a_{11}C'_A + a_{12}T' + b_{11}T'_c + b_{12}C'_{Af}$$

$$\frac{dT'}{dt} = a_{21}C'_A + a_{22}T' + b_{21}T'_c + b_{22}C'_{Af}$$

Here, the primes (for example, C'_A) denote a deviation from the nominal steady-state condition at which the model has been linearized. The constants a_{ij} and b_{ij} are the coefficients of the Jacobian matrices (normally indicated as A and B) with respect to state and input, respectively. A symbolic expression of the majority of these coefficients is given in [1].

Since measurement of reactant concentrations is often difficult, a common assumption is that T is the only measured output, while C_A is unmeasured. For similar reasons, C_{Af} is commonly assumed to be an unmeasured disturbance. In general, T_c, is the manipulated variable used to control the reactor.

The linearized model fits the general state-space format

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du,$$

where

$$x = \begin{bmatrix} C'_A \\ T' \end{bmatrix}, u = \begin{bmatrix} T'_c \\ C'_{Af} \end{bmatrix}, y = \begin{bmatrix} T' \\ C'_A \end{bmatrix},$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The following code shows how to define such a model for some specific values of the a_{ij} and b_{ij} constants:

```
A = [   -5  -0.3427;
     47.68    2.785];
B = [    0   1
       0.3   0];
C = flipud(eye(2));
D = zeros(2);
CSTR = ss(A,B,C,D);
```

These values correspond to a linearization around an operating point in which C_A is 2 kmol/m^3, T is 373 K, C_{Af} is 10 kmol/m^3, T_f is 300 K, and T_c is 299 K. See “Linearization Using MATLAB Code” on page 2-22 for more information.

You can specify the input, output, and state names for your CSTR model. Also, you can specify the input and output signals types.

```
CSTR.InputName = {'T_c', 'C_A_f'};  % set names of input signals
CSTR.OutputName = {'T', 'C_A'};     % set names of output signals
CSTR.StateName = {'C_A', 'T'};      % set names of state variables

% assign input and output signals to different MPC categories
CSTR=setmpcsignals(CSTR,'MV',1,'UD',2,'MO',1,'UO',2);
```

Here, `MV`, `UD`, `MO`, and `UO` stand for "Manipulated Variable," "Unmeasured Disturbance," "Measured Output," and "Unmeasured Output," respectively.

View the CSTR model and its properties.

```
CSTR
```

```
CSTR =

  A =
           C_A         T
   C_A      -5  -0.3427
   T     47.68    2.785

  B =
         T_c  C_A_f
   C_A     0      1
   T     0.3      0

  C =
       C_A    T
   T     0    1
   C_A   1    0
```

```
  D =
          T_c  C_A_f
   T        0      0
   C_A      0      0

Input groups:
       Name        Channels
    Manipulated       1
    Unmeasured        2

Output groups:
       Name       Channels
     Measured        1
    Unmeasured       2

Continuous-time state-space model.
```

In summary, in this linearized model, the first two state variables are the concentration of reagent and the temperature of the reactor, while the first two inputs are the coolant temperature and the inflow feed reagent concentration.

For details on how to obtain this linear model, see the two examples in “Linearize Simulink Models” on page 2-22. In the first example the linearization is done in MATLAB, while in the second one it is done using **Model Linearizer** in Simulink.

Examples using a CSTR model

The following examples use the linear CSTR model.

- “Construct Linear Time Invariant Models” on page 2-15 – Create LTI state-space models for MPC design using the linear CSTR model.
- “Design Controller Using MPC Designer” on page 3-2 – Use a linear CSTR model where the reactor temperature is a measured output. Using **MPC Designer**, you design an MPC controller that stabilizes the closed-loop while constraining the CSTR coolant temperature and its rate of change.
- “Design MPC Controller at the Command Line” on page 3-19 – Design the same controller designed in the previous example but using MATLAB instructions.
- “Test MPC Controller Robustness Using MPC Designer” – Test the sensitivity of your MPC controller to prediction errors using simulations in **MPC Designer**.
- “Compute Steady-State Output Sensitivity Gain” – Analyze the steady-state performance of an MPC controller.
- “Compare Multiple Controller Responses Using MPC Designer” – Compare multiple controller responses using **MPC Designer**.

The following examples use the nonlinear CSTR model.

The example “Linearize Simulink Models Using MPC Designer” on page 2-31 shows how to linearize the nonlinear Simulink model of the reactor at different operating points, and using different approaches, in the context of designing an MPC controller, using **MPC Designer**.

Similarly, in the example “Design MPC Controller in Simulink” on page 3-32, the **MPC Designer** is used first to linearize the same nonlinear model around an operating point in which C_A is around 2

(kg·mol)/m^3 and then to design an MPC controller for the linearized plant. In this example C_A is the only measured output (and T_c is the control input).

The example “Simulate Linear MPC Controller with Nonlinear Plant Using Successive Linearizations” uses a `for` loop to successively linearize the nonlinear model using the `linmod` command, redesign a linear MPC controller, calculate the control input, and feed it back into the nonlinear Simulink model at each time step. This approach is no longer recommended, use “Adaptive MPC” or “Gain-Scheduled MPC” instead.

The example “Adaptive MPC Control of Nonlinear Chemical Reactor Using Successive Linearization” uses the Adaptive MPC Controller block to simulate the closed-loop directly in Simulink. Here a linearization block is used to extract a linear plant model from the nonlinear equations at each time step. In general MPC adaptive control is the preferred approach when a linear plant model can be obtained at run time and when all the linearized plant models have the same order and time delay.

Similarly, in the example “Adaptive MPC Control of Nonlinear Chemical Reactor Using Online Model Estimation” a Recursive Polynomial Model Estimator is used (instead of a linearization block) to identify a two-input (T_f and T_c) and one-output (T) discrete time ARX model based on the measured temperatures at each control interval. The estimated model is then converted into state space and fed to an Adaptive MPC Controller block, which provides the control input to the nonlinear plant. Online estimation can be a good approach when the plant is stable and slowly varying, and its equations are not accurately known.

In the example “Adaptive MPC Control of Nonlinear Chemical Reactor Using Linear Parameter-Varying System” a linear parameter varying (LPV) system consisting of three linear plant models (at an initial, intermediate, and final operating point) is constructed offline. At run time, the LPV System block feeds an appropriate interpolation to an Adaptive MPC Controller block, which provides the control input to the nonlinear plant. Plant interpolation can be a good approach when, at a given point, the interpolated plant is a good approximation of the actual one, the scheduling variable varies comparatively slowly, and obtaining a linearized plant at run time might be too computationally expensive or unsafe.

In the example “Gain-Scheduled MPC Control of Nonlinear Chemical Reactor” three different MPC controllers (for the plant at an initial, intermediate, and final operating point) are designed. These controllers are stored in the Multiple MPC Controllers block and are switched at run time at appropriate points along the transition path. Switching controllers is a good approach when the linearized plant models have different order or time delays.

Finally, in the example “Nonlinear Model Predictive Control of an Exothermic Chemical Reactor”, the nonlinear plant is controlled by a single Nonlinear MPC Controller block, using T_c as the control input and C_A as the only measured output. In general, nonlinear MPC control is the remaining strategy to control highly nonlinear plants when all the previous approaches are unsuitable, or you need to use nonlinear constraints or non-quadratic cost functions.

References

[1] Bequette, B., *Process Dynamics: Modeling, Analysis and Simulation*, Prentice-Hall, 1998, Module 8, pp. 641-660.

[2] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp, *Process Dynamics and Control*, 2nd Edition, Wiley, 2004, pp. 34–36 and 94–95.

See Also

Apps
MPC Designer

Functions
`setmpcsignals` | `ss`

Objects
`mpc`

Related Examples

- “Construct Linear Time Invariant Models” on page 2-15

More About

- “What is Model Predictive Control?” on page 1-3
- “MPC Prediction Models” on page 2-3
- “Model-Based Design with Simulink” (Simulink)

Construct Linear Time Invariant Models

Model Predictive Control Toolbox software supports the same LTI model formats as does Control System Toolbox software. You can use whichever is most convenient for your application and convert from one format to another. For more details, see “Basic Models”.

Transfer Function Models

A transfer function (TF) relates a particular input/output pair of (possibly vector) signals. For example, if $u(t)$ is a plant input and $y(t)$ is an output, the transfer function relating them might be:

$$\frac{Y(s)}{U(s)} = G(s) = \frac{s+2}{s^2+s+10}e^{-1.5s}$$

This TF consists of a *numerator* polynomial, s+2, a *denominator* polynomial, s^2+s+10, and a delay, which is 1.5 time units here. You can define G using Control System Toolbox `tf` function:

```
Gtf1 = tf([1 2], [1 1 10],'OutputDelay',1.5)
```

```
Transfer function:
                 s + 2
exp(-1.5*s) * ------------
              s^2 + s + 10
```

Zero/Pole/Gain Models

Like the TF format, the zero/pole/gain (ZPK) format relates an input/output pair of (possibly vector) signals. The difference is that the ZPK numerator and denominator polynomials are factored, as in

$$G(s) = 2.5\frac{s+0.45}{(s+0.3)(s+0.1+0.7i)(s+0.1-0.7i)}$$

(zeros and/or poles are complex numbers in general).

You define the ZPK model by specifying the zero(s), pole(s), and gain as in

```
poles = [-0.3, -0.1+0.7*i, -0.1-0.7*i];
Gzpk1 = zpk(-0.45,poles,2.5);
```

State-Space Models

The state-space format is convenient if your model is a set of LTI differential and algebraic equations.

The linearized model of a Continuously Stirred Tank Reactor (CSTR) is shown in “CSTR Model” on page 2-8. In the model, the first two state variables are the concentration of reagent (here referred to as C_A and measured in kmol/m^3) and the temperature of the reactor (here referred to as T, measured in K), while the first two inputs are the coolant temperature (T_c, measured in K, used to control the plant), and the inflow feed reagent concentration C_{Af}, measured in kmol/m^3, (often considered as unmeasured disturbance).

A state-space model can be defined as follows:

```
A = [   -5  -0.3427;
     47.68    2.785];
```

```
B = [    0   1
       0.3   0];
C = [0 1
     1 0];
D = zeros(2,2);
CSTR = ss(A,B,C,D);
```

This defines a *continuous-time* state-space model stored in the variable CSTR. The model is continuous time because no sampling time was specified, and therefore a default sampling value of zero (which means that the model is continuous time) is assumed. You can also specify discrete-time state-space models. You can specify delays in both continuous-time and discrete-time models.

LTI Object Properties

The ss function in the last line of the above code creates a state-space model, CSTR, which is an *LTI object*. The tf and zpk commands described in "Transfer Function Models" on page 2-15 and "Zero/Pole/Gain Models" on page 2-15 also create LTI objects. Such objects contain the model parameters as well as optional properties.

Additional LTI Input and Output Properties

The following code sets some optional input and outputs names and properties for the CSTR state-space object:

```
CSTR.InputName = {'T_c', 'C_A_f'};  % set names of input signals
CSTR.OutputName = {'T', 'C_A'};     % set names of output signals
CSTR.StateName = {'C_A', 'T'};      % set names of state variables

% assign input and output signals to different MPC categories
CSTR=setmpcsignals(CSTR,'MV',1,'UD',2,'MO',1,'UO',2)
```

The first three lines specify labels for the input, output and state variables. The next four specify the signal type for each input and output. The designations MV, UD, MO, and UO mean *manipulated variable*, *unmeasured disturbance*, *measured output*, and *unmeasured output*. (See "MPC Signal Types" on page 2-2 for definitions.) For example, the code specifies that input 2 of model CSTR is an unmeasured disturbance. The last line causes the LTI object to be displayed, generating the following lines in the MATLAB Command Window:

```
CSTR =

  A =
            C_A         T
   C_A       -5  -0.3427
   T      47.68    2.785

  B =
          T_c  C_A_f
   C_A      0      1
   T      0.3      0

  C =
        C_A    T
   T      0    1
   C_A    1    0

  D =
```

```
          T_c  C_A_f
   T        0      0
   C_A      0      0

Input groups:
       Name        Channels
    Manipulated       1
    Unmeasured        2

Output groups:
       Name       Channels
     Measured        1
    Unmeasured       2

Continuous-time state-space model.
```

Input and Output Names

The optional `InputName` and `OutputName` properties affect the model displays, as in the above example. The software also uses the `InputName` and `OutputName` properties to label plots and tables. In that context, the underscore character causes the next character to be displayed as a subscript.

Input and Output Types

As mentioned in “MPC Signal Types” on page 2-2, Model Predictive Control Toolbox software supports three input types and two output types. In a Model Predictive Control Toolbox design, designation of the input and output types determines the controller dimensions and has other important consequences.

For example, suppose your plant structure were as follows:

Plant Inputs	Plant Outputs
Two manipulated variables (MVs)	Three measured outputs (MOs)
One measured disturbance (MD)	Two unmeasured outputs (UOs)
Two unmeasured disturbances (UDs)	

The resulting controller has four inputs (the three MOs and the MD) and two outputs (the MVs). It includes feedforward compensation for the measured disturbance, and assumes that you wanted to include the unmeasured disturbances and outputs as part of the regulator design.

If you didn't want a particular signal to be treated as one of the above types, you could do one of the following:

- Eliminate the signal before using the model in controller design.
- For an output, designate it as unmeasured, then set its weight to zero.
- For an input, designate it as an unmeasured disturbance, then define a custom state estimator that ignores the input.

Note By default, the software assumes that unspecified plant inputs are manipulated variables, and unspecified outputs are measured. Thus, if you didn't specify signal types in the above example, the controller would have four inputs (assuming all plant outputs were measured) and five outputs (assuming all plant inputs were manipulated variables).

Note Since the D matrix is zero, the output does not instantly respond to change in the input. The Model Predictive Control Toolbox software prohibits direct (instantaneous) feedthrough from a manipulated variable to an output. For example, the CSTR state-space model could include direct feedthrough from the unmeasured disturbance, C_{Af}, to either C_A or T but direct feedthrough from T_c to either (measured or not) output would violate this restriction. When the model has a direct feedthrough from T_c, you can add a small delay at this input to circumvent the problem.

For `CSTR`, the default Model Predictive Control Toolbox assumptions are incorrect. You must set its `InputGroup` and `OutputGroup` properties, as illustrated in the above code, or modify the default settings when you load the model into **MPC Designer**.

Use `setmpcsignals` to make type definition. For example:

```
CSTR = setmpcsignals(CSTR,'UD',2,'UO',2);
```

sets `InputGroup` and `OutputGroup` to the same values as in the previous example. The `CSTR` display would then include the following lines:

```
Input groups:
       Name        Channels
    Unmeasured        2
    Manipulated       1



Output groups:
       Name       Channels
    Unmeasured       2
     Measured        1
```

Notice that `setmpcsignals` sets unspecified inputs to `Manipulated` and unspecified outputs to `Measured`.

LTI Model Characteristics

Control System Toolbox software provides functions for analyzing LTI models. Some of the more commonly used are listed below. Type the example code at the MATLAB prompt to see how they work for the `CSTR` example.

Example	Intended Result
`damp(CSTR)`	Displays the damping ratio, natural frequency, and time constant of the poles of `CSTR`.
`pzmap(CSTR)`	Plots the poles and zeros of `CSTR`.
`pole(CSTR)`	Calculates the poles of `CSTR` (to check stability, etc.).
`tzero(CSTR)`	Calculates the transmission zeros of `CSTR`.
`dcgain(CSTR)`	Calculates the steady state gain matrix of `CSTR`.
`step(CSTR)`	Plots unit-step responses of `CSTR`.
`stepinfo(CSTR)`	Calculates rise time, settling time, and other step-response characteristics of `CSTR`.
`impulse(CSTR)`	Plots the unit-impulse responses of `CSTR`.

Example	Intended Result
`sigma(CSTR)`	Plots the singular values of the frequency response of `CSTR`.
`bode(CSTR)`	Plots the Bode frequency responses of `CSTR`.
`nyquist(CSTR)`	Plots the Nyquist frequency responses of `CSTR`.
`nichols(CSTR)`	Plots the Nichols frequency responses of `CSTR`.
`linearSystemAnalyzer(CSTR)`	Opens the Linear System Analyzer with the `CSTR` model loaded. You can then display model characteristics by making menu selections.

See Also

Apps
MPC Designer

Functions
`tf` | `zpk` | `ss` | `setmpcsignals`

Objects
`mpc`

Related Examples

- “Model Predictive Control of Multi-Input Single-Output Plant” on page 3-56

More About

- “Specify Multi-Input Multi-Output Plants” on page 2-20

Specify Multi-Input Multi-Output Plants

Most MPC applications involve plants with multiple inputs and outputs. You can use `ss`, `tf`, and `zpk` to represent a MIMO plant model. For example, consider the following model of a distillation column [1], which has been used in many advanced control studies:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{12.8e^{-s}}{16.7s+1} & \frac{-18.9e^{-3s}}{21.0s+1} & \frac{3.8e^{-8.1s}}{14.9s+1} \\ \frac{6.6e^{-7s}}{10.9s+1} & \frac{-19.4e^{-3s}}{14.4s+1} & \frac{4.9e^{-3.4s}}{13.2s+1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

Outputs y_1 and y_2 represent measured product purities. The controller manipulates the inputs, u_1 and u_2, to hold each output at a specified setpoint. These inputs represent the flow rates of reflux and reboiler steam, respectively. Input u_3 is a measured feed flow rate disturbance.

The model consists of six transfer functions, one for each input/output pair. Each transfer function is the first-order-plus-delay form often used by process control engineers.

Specify the individual transfer functions for each input/output pair. For example, `g12` is the transfer function from input u_1 to output y_2.

```
g11 = tf( 12.8, [16.7 1], 'IOdelay', 1.0,'TimeUnit','minutes');
g12 = tf(-18.9, [21.0 1], 'IOdelay', 3.0,'TimeUnit','minutes');
g13 = tf(  3.8, [14.9 1], 'IOdelay', 8.1,'TimeUnit','minutes');
g21 = tf(  6.6, [10.9 1], 'IOdelay', 7.0,'TimeUnit','minutes');
g22 = tf(-19.4, [14.4 1], 'IOdelay', 3.0,'TimeUnit','minutes');
g23 = tf(  4.9, [13.2 1], 'IOdelay', 3.4,'TimeUnit','minutes');
```

Define a MIMO system by creating a matrix of transfer function models.

```
DC = [g11 g12 g13
      g21 g22 g23];
```

Define the input and output signal names and specify the third input as a measured input disturbance.

```
DC.InputName = {'Reflux Rate','Steam Rate','Feed Rate'};
DC.OutputName = {'Distillate Purity','Bottoms Purity'};
DC = setmpcsignals(DC,'MD',3);
```

```
-->Assuming unspecified input signals are manipulated variables.
```

Review the resulting system.

```
DC
```

```
DC =

  From input "Reflux Rate" to output...
                                    12.8
   Distillate Purity:  exp(-1*s) * ----------
                                   16.7 s + 1

                                    6.6
   Bottoms Purity:  exp(-7*s) * ----------
                                  10.9 s + 1
```

```
  From input "Steam Rate" to output...
                                    -18.9
   Distillate Purity:  exp(-3*s) * --------
                                    21 s + 1

                                   -19.4
   Bottoms Purity:  exp(-3*s) * ----------
                                  14.4 s + 1

  From input "Feed Rate" to output...
                                      3.8
   Distillate Purity:  exp(-8.1*s) * ----------
                                     14.9 s + 1

                                     4.9
   Bottoms Purity:  exp(-3.4*s) * ----------
                                   13.2 s + 1

Input groups:
       Name        Channels
     Measured         3
    Manipulated      1,2

Output groups:
      Name      Channels
    Measured      1,2

Continuous-time transfer function.
```

References

[1] Wood, R. K., and M. W. Berry, *Chem. Eng. Sci.*, Vol. 28, pp. 1707, 1973.

See Also

Apps
MPC Designer

Functions
`ss` | `tf` | `zpk` | `setmpcsignals`

Objects
`mpc`

Related Examples

- “Model Predictive Control of Multi-Input Single-Output Plant” on page 3-56

More About

- “Construct Linear Time Invariant Models” on page 2-15

Linearize Simulink Models

Generally, real systems are nonlinear. To design an MPC controller for a nonlinear system, you can model the plant in Simulink.

Although an MPC controller can regulate a nonlinear plant, the model used within the controller must be linear. In other words, the controller employs a linear approximation of the nonlinear plant. The accuracy of this approximation significantly affects controller performance.

To obtain such a linear approximation, you *linearize* the nonlinear plant at a specified *operating point*.

Note The following examples require Simulink Control Design software.

You can linearize a Simulink model:

- From the command line.
- Using the **Model Linearizer**.
- Using **MPC Designer**. For an example, see “Linearize Simulink Models Using MPC Designer” on page 2-31.

Linearization Using MATLAB Code

This example shows how to obtain a linear model of a plant using a MATLAB script.

For this example the CSTR model, `CSTR_OpenLoop`, is linearized. The model inputs are the coolant temperature (manipulated variable of the MPC controller), limiting reactant concentration in the feed stream, and feed temperature. The model states are the temperature and concentration of the limiting reactant in the product stream. Both states are measured and used for feedback control.

Obtain Steady-State Operating Point

The operating point defines the nominal conditions at which you linearize a model. It is usually a steady-state condition.

Suppose that you plan to operate the CSTR with the output concentration, `C_A`, at 2 $kmol/m^3$. The nominal feed concentration is 10 $kmol/m^3$, and the nominal feed temperature is 300 K.

Create and visualize an operating point specification object to define the steady-state conditions.

```
opspec = operspec('CSTR_OpenLoop');
opspec = addoutputspec(opspec,'CSTR_OpenLoop/CSTR',2);
opspec.Outputs(1).Known = true;
opspec.Outputs(1).y = 2;
opspec
```

```
opspec =
 Operating point specification for the Model CSTR_OpenLoop.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
```

```
     x         Known    SteadyState      Min         Max        dxMin       dxMax
_____________ _________ _____________ _________ ___________ _________ ___________

(1.) CSTR_OpenLoop/CSTR/C_A
  8.5695       false       true          0          Inf        -Inf         Inf
(2.) CSTR_OpenLoop/CSTR/T_K
  311.267      false       true          0          Inf        -Inf         Inf

Inputs:
----------
  u   Known  Min   Max
_____ _____ _____ _____

(1.) CSTR_OpenLoop/Coolant Temperature
  0   false -Inf   Inf

Outputs:
----------
  y   Known  Min   Max
_____ _____ _____ _____

(1.) CSTR_OpenLoop/CSTR
  2   true  -Inf   Inf
```

Search for an operating point that satisfies the specifications.

```
op1 = findop('CSTR_OpenLoop',opspec);

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

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

Operating point specifications were successfully met.
States:
----------
    Min          x          Max        dxMin        dx         dxMax
___________ ___________ ___________ _________ ___________ _________

(1.) CSTR_OpenLoop/CSTR/C_A
     0           2          Inf          0      -4.6683e-12      0
(2.) CSTR_OpenLoop/CSTR/T_K
     0       373.1311       Inf          0      5.5678e-11       0

Inputs:
----------
  Min       u       Max
________ ________ ________

(1.) CSTR_OpenLoop/Coolant Temperature
  -Inf   299.0349   Inf

Outputs:
----------
Min  y  Max
___ ___ ___
```

```
(1.) CSTR_OpenLoop/CSTR
 2   2   2
```

The calculated operating point is `C_A` = 2 $kmol/m^3$ and `T_K` = 373 K. Notice that the steady-state coolant temperature is also given as 299 K, which is the nominal value of the input used to control the plant.

To specify:

- Values of known inputs, use the `Input.Known` and `Input.u` fields of `opspec`
- Initial guesses for state values, use the `State.x` field of `opspec`

For example, the following code specifies the coolant temperature as 305 K and initial guess values of the `C_A` and `T_K` states before calculating the steady-state operating point:

```
opspec = operspec('CSTR_OpenLoop');
opspec.States(1).x = 1;
opspec.States(2).x = 400;
opspec.Inputs(1).Known = true;
opspec.Inputs(1).u = 305;

op2 = findop('CSTR_OpenLoop',opspec)

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

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

Operating point specifications were successfully met.
States:
----------
    Min          x          Max        dxMin        dx         dxMax
___________ ___________ ___________ ___________ ___________ ___________

(1.) CSTR_OpenLoop/CSTR/C_A
     0        1.7787        Inf          0      -3.0198e-14      0
(2.) CSTR_OpenLoop/CSTR/T_K
     0       376.5371       Inf          0      3.1264e-13       0

Inputs:
----------
Min  u  Max
___ ___ ___

(1.) CSTR_OpenLoop/Coolant Temperature
305 305 305

Outputs: None
----------

op2 =
 Operating point for the Model CSTR_OpenLoop.
 (Time-Varying Components Evaluated at time t=0)

States:
```

```
----------
   x
________

(1.) CSTR_OpenLoop/CSTR/C_A
 1.7787
(2.) CSTR_OpenLoop/CSTR/T_K
376.5371

Inputs:
----------
 u
___

(1.) CSTR_OpenLoop/Coolant Temperature
305
```

Specify Linearization Inputs and Outputs

If the linearization input and output signals are already defined in the model, as in `CSTR_OpenLoop`, then use the following to obtain the signal set.

```
io = getlinio('CSTR_OpenLoop');
```

Otherwise, specify the input and output signals as shown here.

```
io(1) = linio('CSTR_OpenLoop/Coolant Temperature',1,'input');
io(2) = linio('CSTR_OpenLoop/Feed Concentration',1,'input');
io(3) = linio('CSTR_OpenLoop/Feed Temperature',1,'input');
io(4) = linio('CSTR_OpenLoop/CSTR',1,'output');
io(5) = linio('CSTR_OpenLoop/CSTR',2,'output');
```

Linearize Model

Linearize the model using the specified operating point, `op1`, and input/output signals, `io`.

```
sys = linearize('CSTR_OpenLoop',op1,io)
```

```
sys =

  A =
           C_A      T_K
   C_A       -5  -0.3427
   T_K    47.68    2.785

  B =
        Coolant Temp  Feed Concent  Feed Tempera
   C_A             0             1             0
   T_K           0.3             0             1

  C =
          C_A  T_K
   CSTR/1   0    1
   CSTR/2   1    0

  D =
          Coolant Temp  Feed Concent  Feed Tempera
   CSTR/1             0             0             0
   CSTR/2             0             0             0
```

```
Continuous-time state-space model.
```

Linearize the model also around the operating point, `op2`, using the same input/output signals.

```
sys = linearize('CSTR_OpenLoop',op2,io)
```

```
sys =

  A =
            C_A      T_K
   C_A   -5.622  -0.3458
   T_K     55.1    2.822

  B =
        Coolant Temp  Feed Concent  Feed Tempera
   C_A             0             1             0
   T_K           0.3             0             1

  C =
          C_A  T_K
   CSTR/1   0    1
   CSTR/2   1    0

  D =
          Coolant Temp  Feed Concent  Feed Tempera
   CSTR/1             0             0             0
   CSTR/2             0             0             0

Continuous-time state-space model.
```

Linearization Using Model Linearizer in Simulink Control Design

This example shows how to linearize a Simulink model using the **Model Linearizer**, provided by the Simulink Control Design software.

Open Simulink Model

This example uses the CSTR model, `CSTR_OpenLoop`.

```
open_system('CSTR_OpenLoop')
```

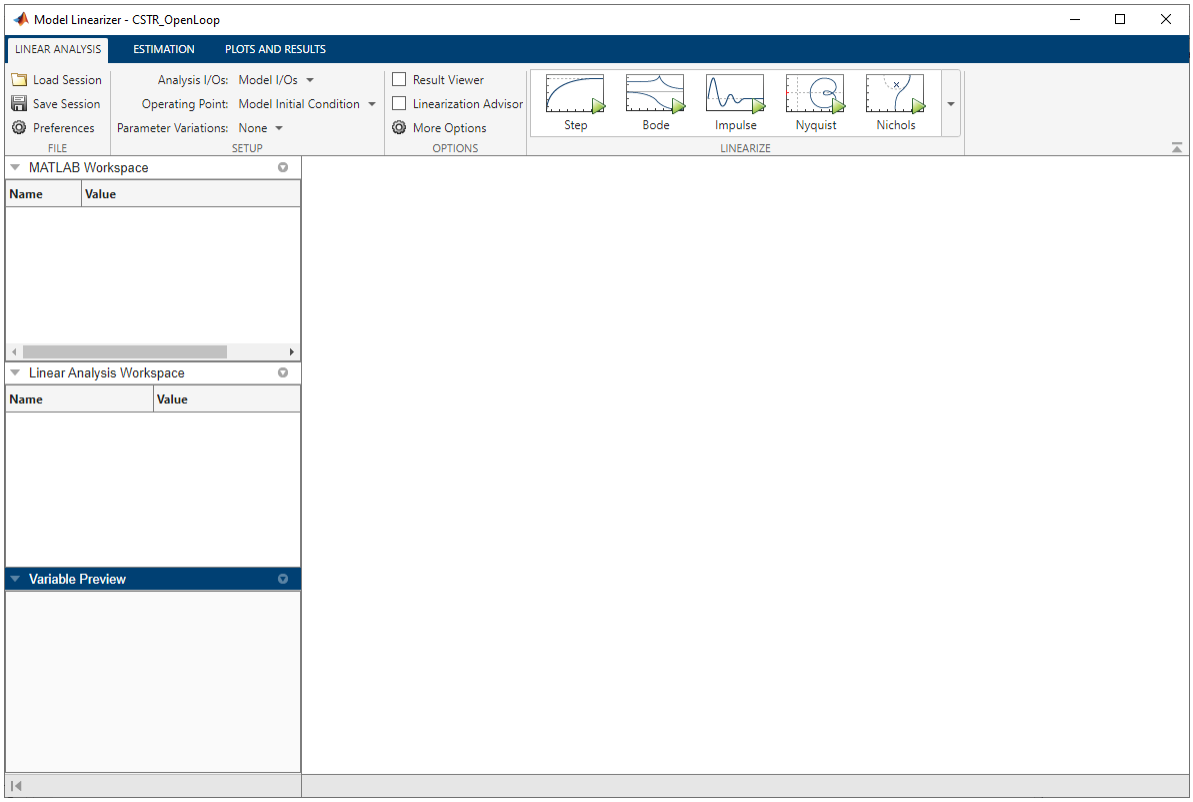
Specify Linearization Inputs and Outputs

The linearization inputs and outputs are already specified for `CSTR_OpenLoop`. The input signals correspond to the outputs from the `Feed Concentration`, `Feed Temperature`, and `Coolant Temperature` blocks. The output signals are the inputs to the `CSTR Temperature` and `Residual Concentration` blocks.

To specify a signal as a linearization input or output, first in the Simulink **Apps** tab, click **Linearization Manager**. Then, in the Simulink model window, click the signal. Finally, in the **Insert Analysis Points** gallery, in the **Closed Loop** section, select either **Input Perturbation** for a linearization input or **Output Measurement** for a linearization output.

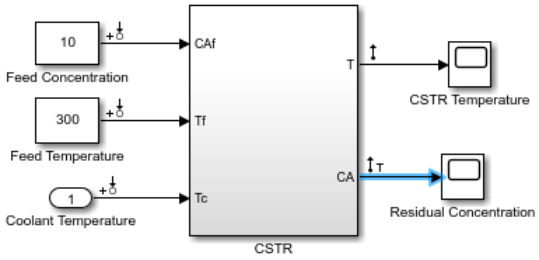
Open Model Linearizer

To open the **Model Linearizer**, in the **Apps** tab, click **Model Linearizer**.



Specify Residual Concentration as Known Trim Constraint

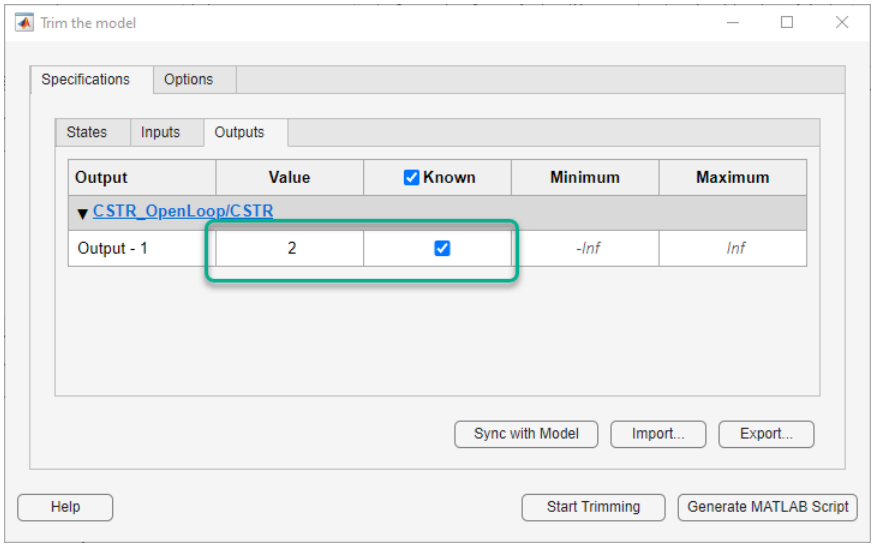
To specify the residual concentration as a known trim constant, first in the Simulink **Apps** tab, click **Linearization Manager**. Then, in the Simulink model window, click the `CA` output signal from the `CSTR` block. Finally, in the **Insert Analysis Points** gallery, in the **Trim** section, select **Trim Output Constraint**.



In the **Model Linearizer**, on the **Linear Analysis** tab, select **Operating Point > Trim Model**.

In the Trim the model dialog box, on the **Outputs** tab:

- Select the **Known** check box for `Channel - 1` under **CSTR_OpenLoop/CSTR**.
- Set the corresponding **Value** to 2 kmol/m^3.

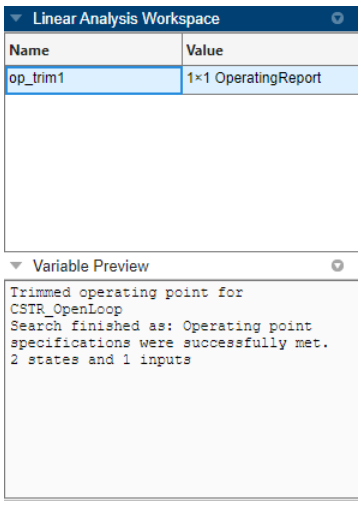


Create and Verify Operating Point

In the Trim the model dialog box, click **Start trimming**.

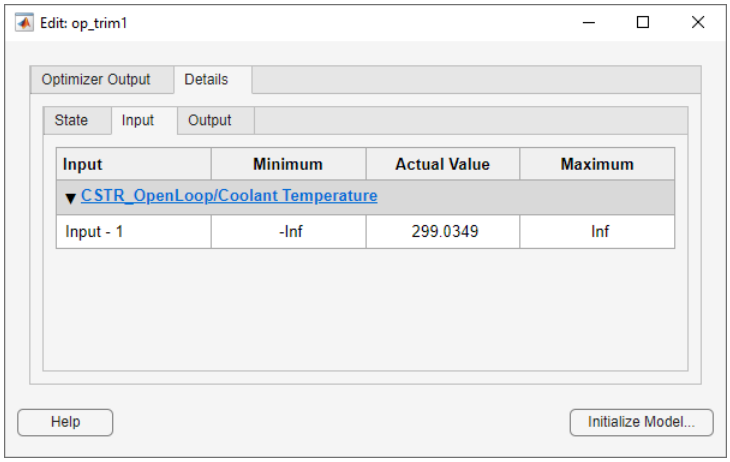
The Trim progress viewer window opens up showing the optimization progress towards finding a point in the state-input space of the model with the characteristics specified in the **States**, **Inputs**, and **Outputs** tabs. After the optimization process terminates, close the trim progress window as well as the Trim the model dialog box.

The operating point `op_trim1` displays in the **Linear Analysis Workspace** of **Model Linearizer**. Select `op_trim1` to display basic information in the **Linear Analysis Workspace** section.



Double click `op_trim1` to view the resulting operating point in the Edit dialog box.

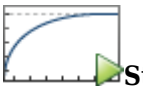
In the Edit dialog box, select the **Input** tab.



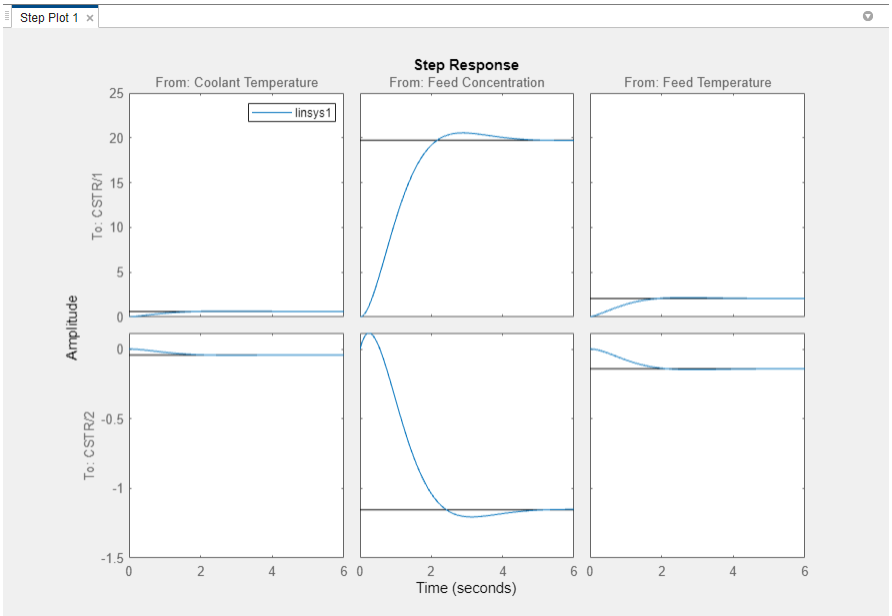
The coolant temperature at steady state is 299 K, as desired. Close the Edit dialog box.

Linearize Model

On the **Linear Analysis** tab, in the **Operating Point** drop-down list, make sure `op_trim1` is selected.

In the **Linearize** section, click  **Step** to linearize the Simulink model and display the step response of the linearized model.

This option creates the linear model `linsys1` in the **Linear Analysis Workspace** and generates a step response for this model. `linsys1` uses `op_trim1` as its operating point.



The step response from feed concentration to output `CSTR/2` displays an interesting inverse response. An examination of the linear model shows that `CSTR/2` is the residual CSTR concentration, `C_A`. When the feed concentration increases, `C_A` increases initially because more reactant is

entering, which increases the reaction rate. This rate increase results in a higher reactor temperature (output `CSTR/1`), which further increases the reaction rate and `C_A` decreases dramatically.

Export Linearization Result

If necessary, you can repeat any of these steps to improve your model performance. Once you are satisfied with your linearization result, in the **Model Linearizer**, drag the linear model from the **Linear Analysis Workspace** section of **Model Linearizer** to the **MATLAB Workspace** section just above it. You can now use your linear model to design an MPC controller.

See Also

Apps
Model Linearizer | **MPC Designer**

Functions
`linearize`

Objects
`mpc`

Related Examples

- “Design MPC Controller in Simulink” on page 3-32
- “Design Controller Using MPC Designer” on page 3-2
- “Design MPC Controller at the Command Line” on page 3-19
- “Linearize Simulink Models Using MPC Designer” on page 2-31

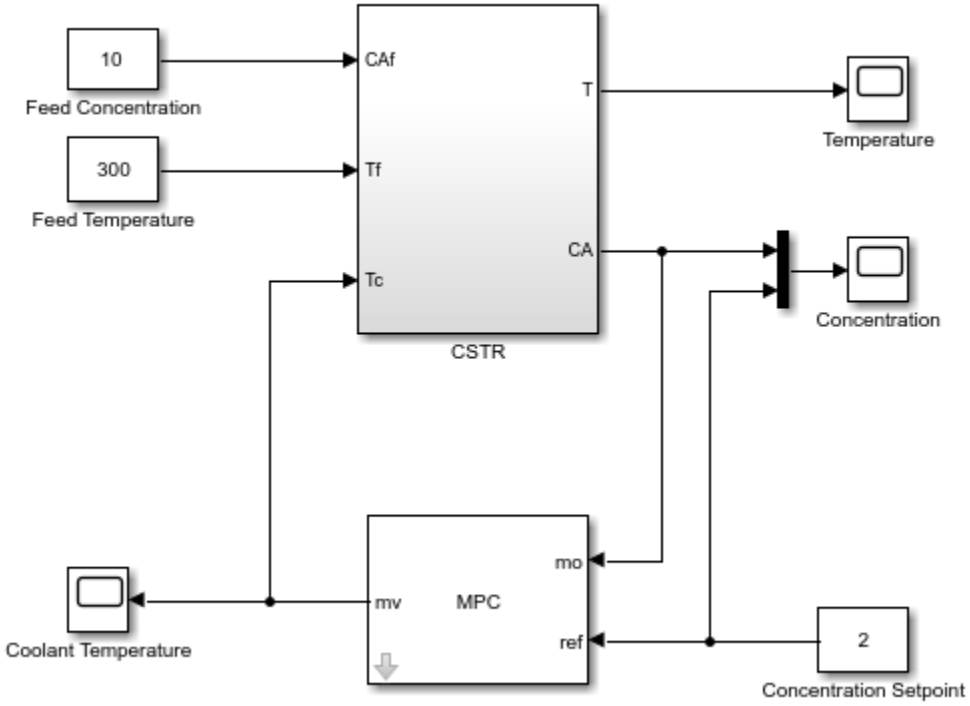
More About

- “What is Model Predictive Control?” on page 1-3

Linearize Simulink Models Using MPC Designer

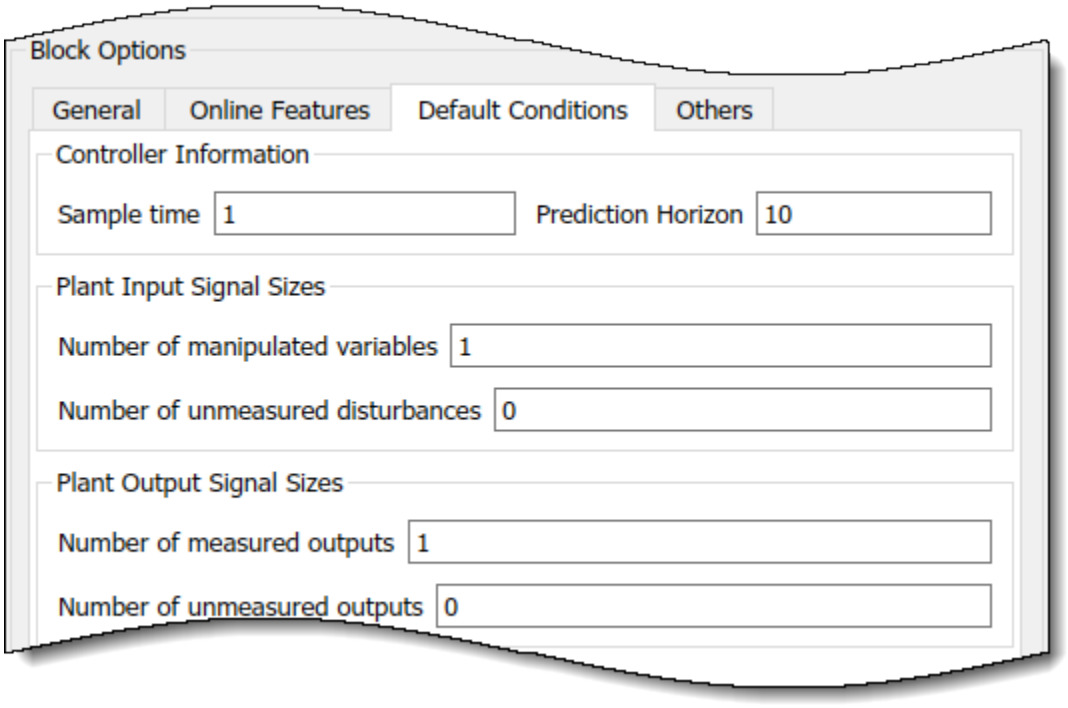
This topic shows how to linearize Simulink models using **MPC Designer**. To do so, open the app from a Simulink model that contains an MPC Controller block. For this example, use the `CSTR_ClosedLoop` model.

```
open_system('CSTR_ClosedLoop')
```

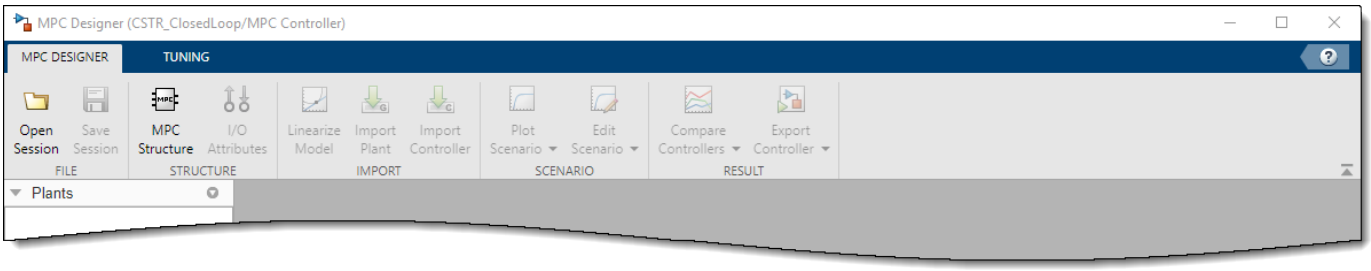


In the model window, double-click the MPC Controller block.

Tip In the MPC Controller Block Parameters dialog box, in the **Default Conditions** tab, you can define the controller sample time and signal dimensions before opening **MPC Designer**.



In the Block Parameters dialog box, ensure that the **MPC Controller** field is empty, and click **Design** to open **MPC Designer**.



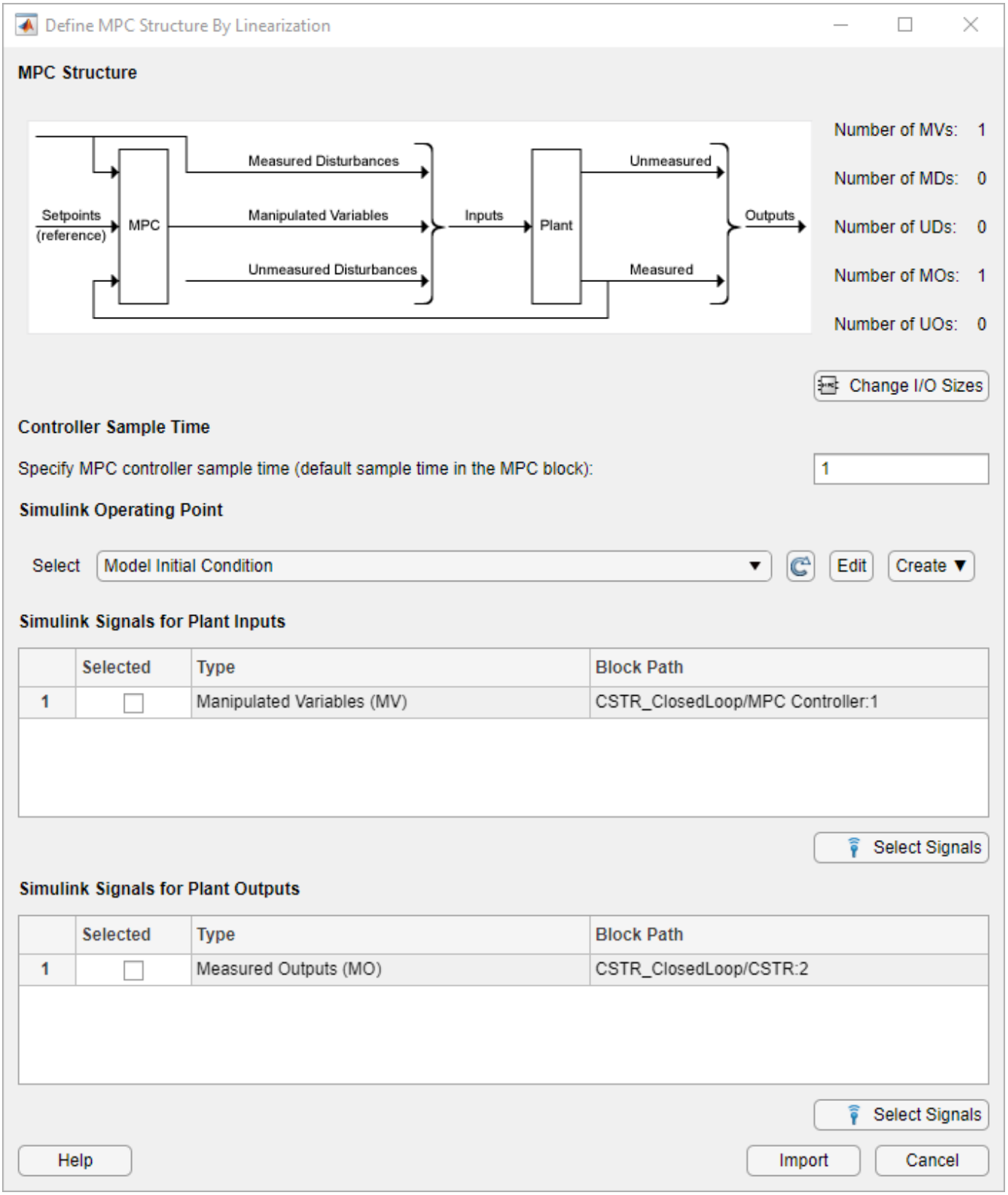
Using **MPC Designer**, you can define the MPC structure by linearizing the Simulink model. After you define the initial MPC structure, you can also linearize the model at different operating points and import the linearized plants.

Note If a controller from the MATLAB workspace is specified in the **MPC Controller** field, the app imports the specified controller. In that case, the MPC structure is derived from the imported controller. However, you can still linearize the Simulink model and import the linearized plants.

Define MPC Structure by Linearization

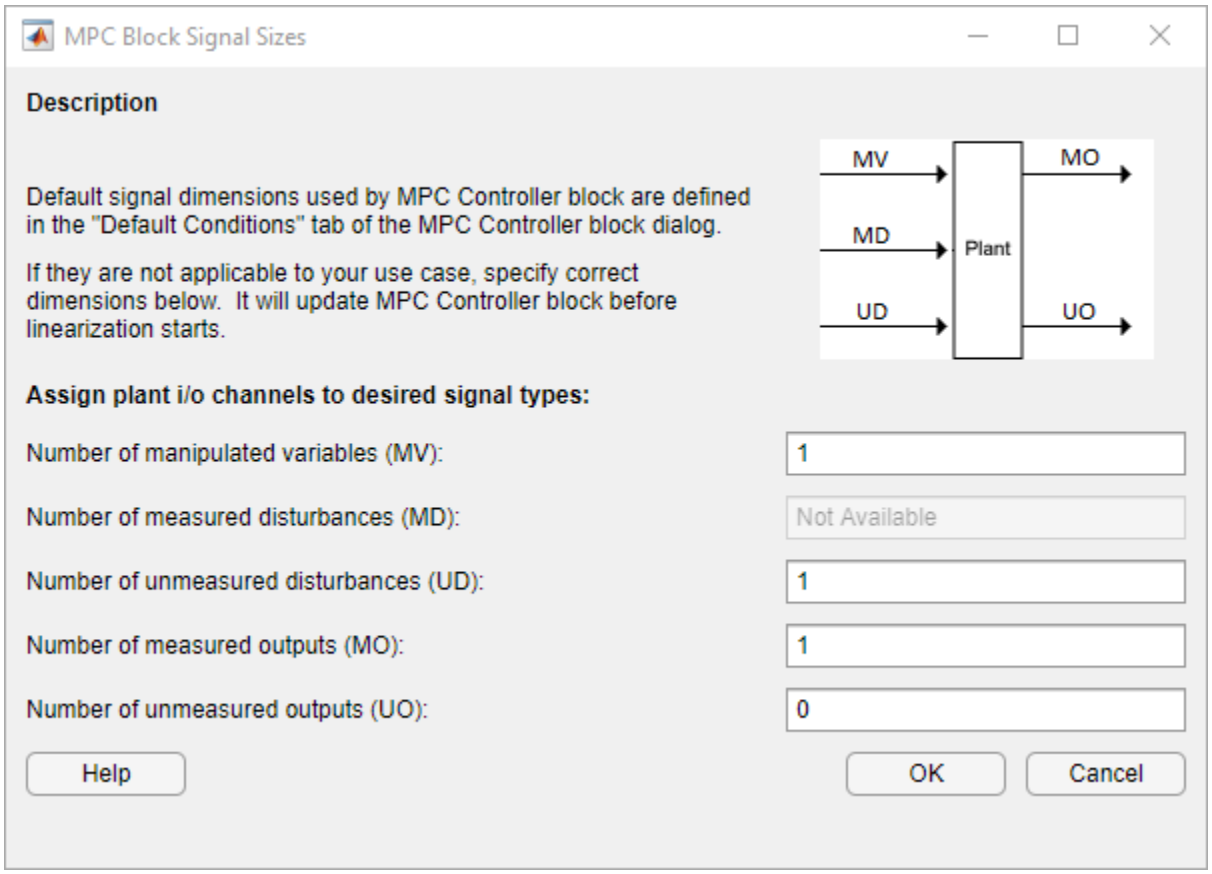
This example shows how to define the plant input/output structure in **MPC Designer** by linearizing a Simulink model.

On the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.


Define MPC Structure By Linearization
MPC Structure
Measured Disturbances
Setpoints
(reference)
MPC
Manipulated Variables
Unmeasured Disturbances
Inputs
Plant
Unmeasured
Measured
Outputs
Number of MVs: 1
Number of MDs: 0
Number of UDs: 0
Number of MOs: 1
Number of UOs: 0
Change I/O Sizes
Controller Sample Time
Specify MPC controller sample time (default sample time in the MPC block):
1
Simulink Operating Point
Select
Model Initial Condition
Edit
Create
Simulink Signals for Plant Inputs
Selected
Type
Block Path
1
Manipulated Variables (MV)
CSTR_ClosedLoop/MPC Controller:1
Select Signals
Simulink Signals for Plant Outputs
Selected
Type
Block Path
1
Measured Outputs (MO)
CSTR_ClosedLoop/CSTR:2
Select Signals
Help
Import
Cancel

Specify Signal Dimensions

In the Define MPC Structure By Linearization dialog box, in the **MPC Structure** section, if the displayed signal dimensions do not match your model, click **Change I/O Sizes** to configure the dimensions. Since unmeasured disturbances or unmeasured outputs in your model do not input to the MPC Controller block, you must specify the dimensions for these signals. For this example, specify one unmeasured disturbance signal.



Click **OK**.

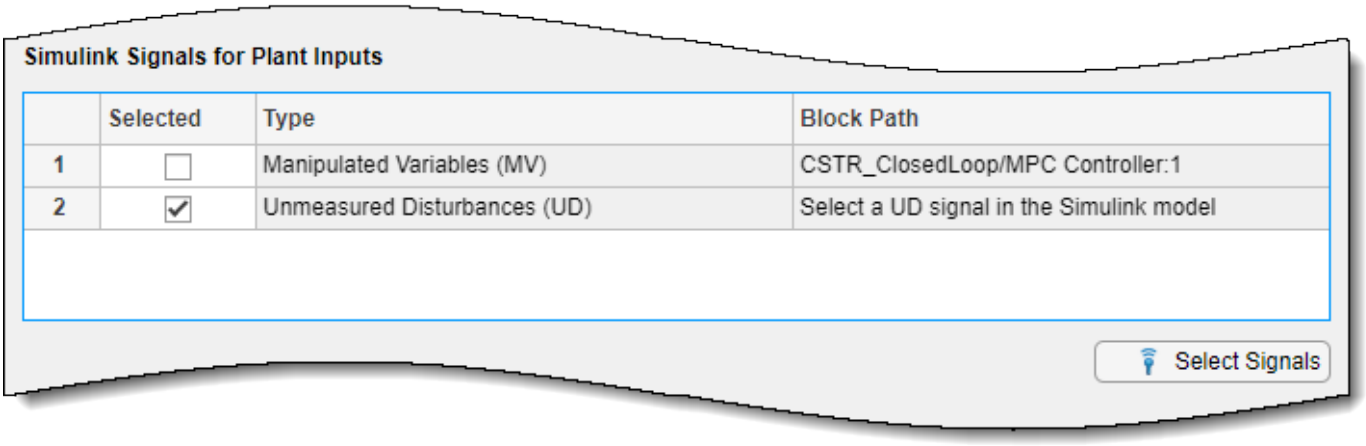
The **Unmeasured Disturbance (UD)** type is added in the **Simulink Signals for Plant Inputs**, without a specified block path.

Select Plant Input/Output Signals

Before linearizing the model, assign Simulink signal lines to each MPC signal type in your model. The app uses these signals as linearization inputs and outputs.

In the **Simulink Signals for Plant Inputs** and **Simulink Signals for Plant Outputs** sections, the **Block Path** is automatically defined for manipulated variables, measured outputs, and measured disturbances. **MPC Designer** detects these signals since they are connected to the MPC Controller block. If your plant has unmeasured disturbances or unmeasured outputs, select their corresponding Simulink signal lines.

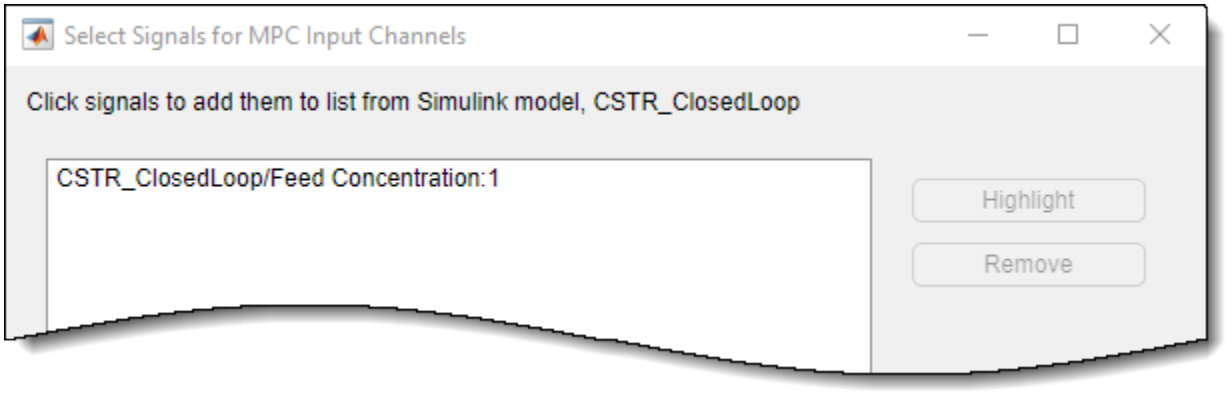
To choose a signal type, use the **Selected** check boxes.



Click **Select Signals**. The Select Signals for MPC Input Channels dialog box opens up.

In the Simulink model window, click the signal line corresponding to the selected signal type.

The signal is highlighted, and its block path is added to the Select signals dialog box.



In the select signals dialog box, click **OK**.

In the Define MPC Structure By Linearization dialog box, the **Block Path** for the selected signal type updates.

Note If your model has measured disturbances, you must connect the `md` input port of the MPC Controller block to the same signal line of the corresponding plant inputs. For more information, see “Connect Measured Disturbances for Linearization” on page 2-46.

Specify Operating Point

In the **Simulink Operating Point** section, in the drop-down list, select an operating point at which to linearize the model. For this example, select **Model Initial Condition**.

For information on the different operating point options, see “Specifying Operating Points” on page 2-39.

Note If you select an option that generates multiple operating points for linearization, **MPC Designer** uses only the first operating point to define the plant structure and linearize the model.

Define Structure and Linearize Model

Click **Import**.

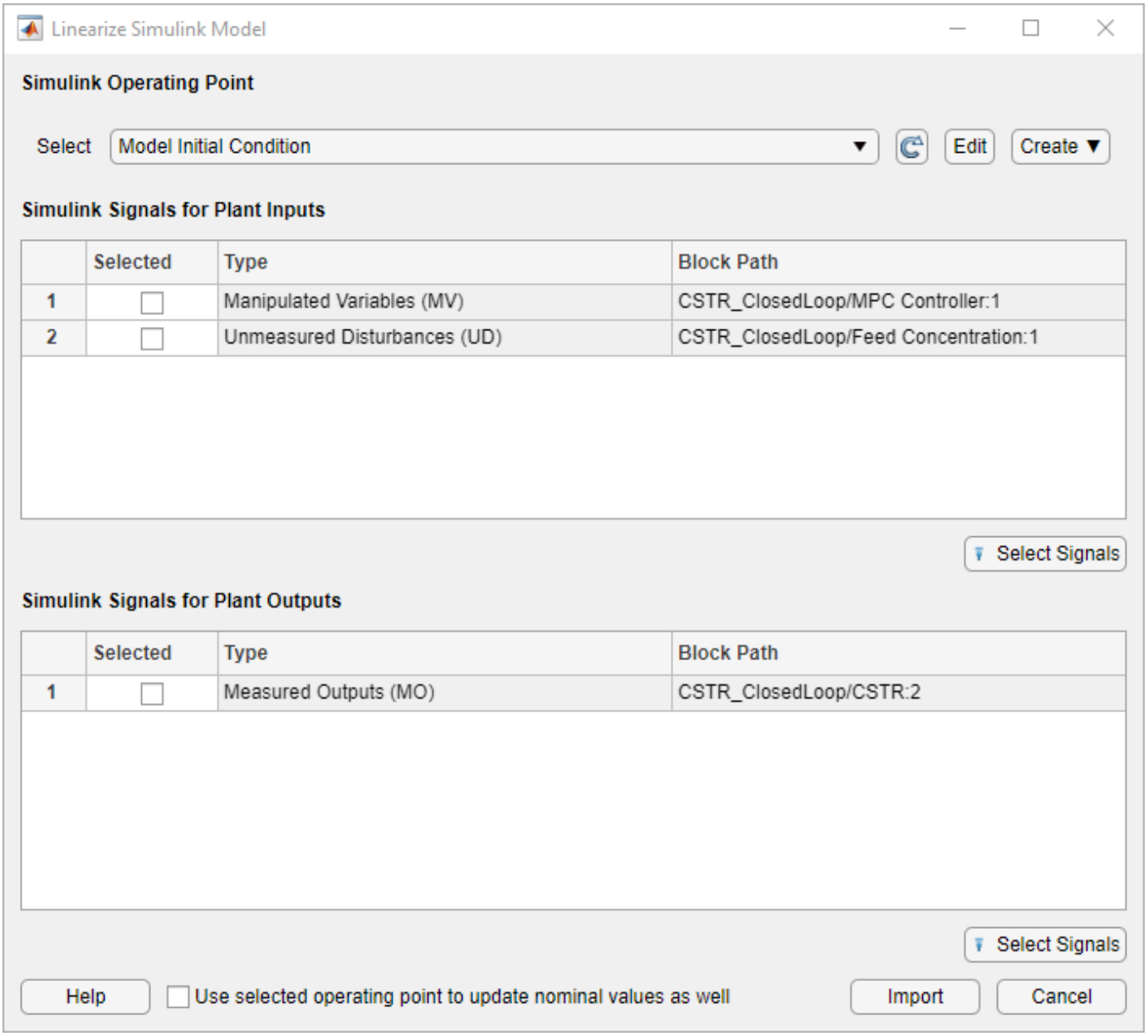
MPC Designer linearizes the Simulink model at the specified operating point using the specified input/output signals, and imports the linearized plant to the **Plants** workspace, on the right hand side of the app. A default controller, which uses the linearized plant as its internal model and input/output signal values at the selected operating point as nominal values, is added to the **Controllers** workspace. A default simulation scenario is also added to the **Scenarios** workspace.

Note All the controller created in the **MPC Designer** share the same nominal values, since otherwise it would not be easy to compare their responses. Therefore, if you update the nominal values, all the controllers are affected.

Linearize Model

After you define the initial MPC structure, you can linearize the Simulink model at different operating points and import the linearized plants. Doing so is useful for validating controller performance against modeling errors.

On the **MPC Designer** tab, in the **Import** section, click **Linearize Model**.



Select Plant Input/Output Signals

In the **Simulink Signals for Plant Inputs** and **Simulink Signals for Plant Outputs** sections, the input/output signal configuration is the same as the one you specify when initially defining the MPC structure.

You cannot change the signal types and dimensions once the structure is defined. However, for each signal type, you can select different signal lines from your Simulink model. The selected lines must have the same dimensions as those defined in the current MPC structure.

Specify Operating Point

In the **Simulink Operating Point** section, in the drop-down list, you can select the operating points at which to linearize the model.

For information on the different operating point options, see “Specifying Operating Points” on page 2-39.

Linearize Model and Import Plant

If you click on **Import**, **MPC Designer** linearizes the Simulink model at the defined operating point and adds the linearized plant, a default controller and a default simulation scenario to the app workspaces on the right hand side, as previously described for the **Import** button of the Define MPC Structure By Linearization dialog box.

If you select the **Use selected operating point to update nominal values as well** option, the nominal values of all the controllers in the **Controllers** workspace of the app are updated using this operating point signal values.

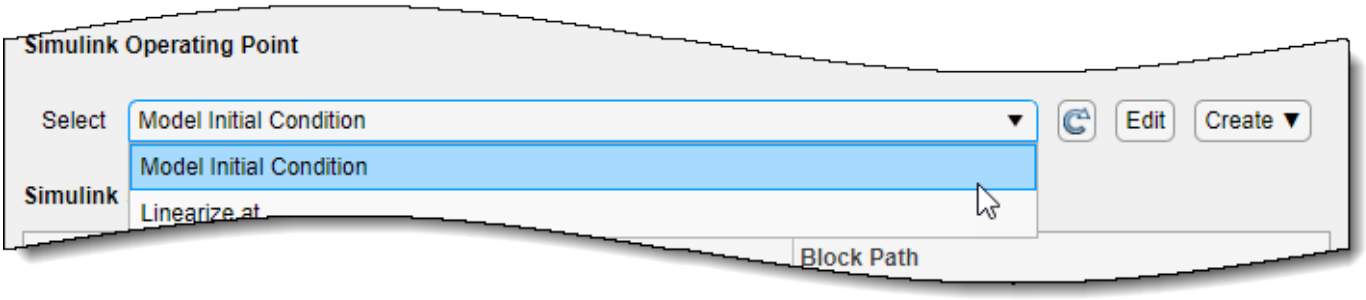
If you select an option that generates multiple operating points for linearization (see “Specifying Operating Points” on page 2-39), the app linearizes the model at all the specified operating points. The linearized plants are added in the **Plants** workspace in the same order in which their corresponding operating points are defined. If you choose to update the nominal values, the app uses the signal values from the first operating point.

Specifying Operating Points

In the **Simulink Operating Point** section of either the Define MPC Structure By Linearization dialog box or the Linearize Simulink Model dialog box, in the drop-down list, you can select or create operating points for model linearization. For more information on finding steady-state operating points, see “About Operating Points” (Simulink Control Design) and “Compute Steady-State Operating Points from Specifications” (Simulink Control Design).

Select Model Initial Condition

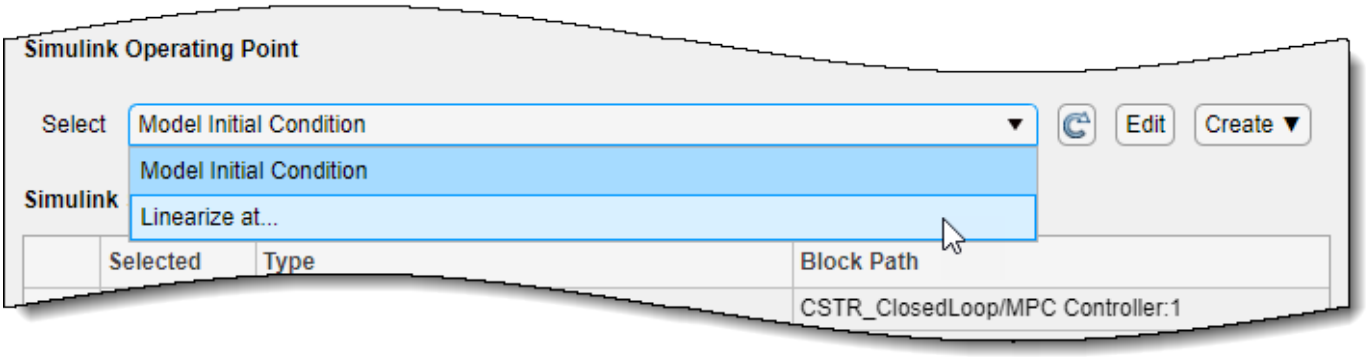
To linearize the model using the initial conditions specified in the Simulink model as the operating point, select **Model Initial Condition**.



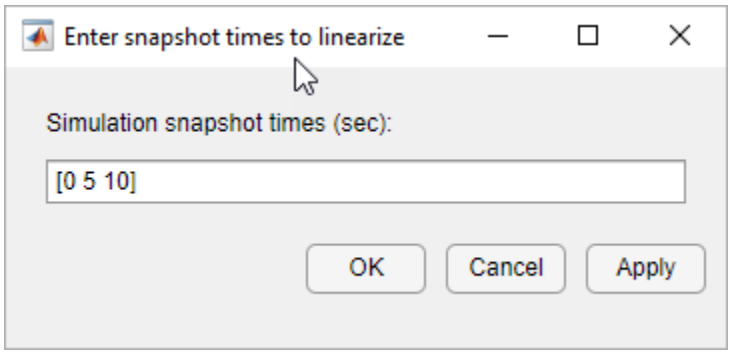
The model initial condition is the default operating point for linearization in **MPC Designer**.

Linearize at Simulation Snapshot Times

To linearize the model at specified simulation snapshot times, select **Linearize at**. Linearizing at snapshot times is useful when you know that your model reaches an equilibrium state after a certain simulation time.



In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



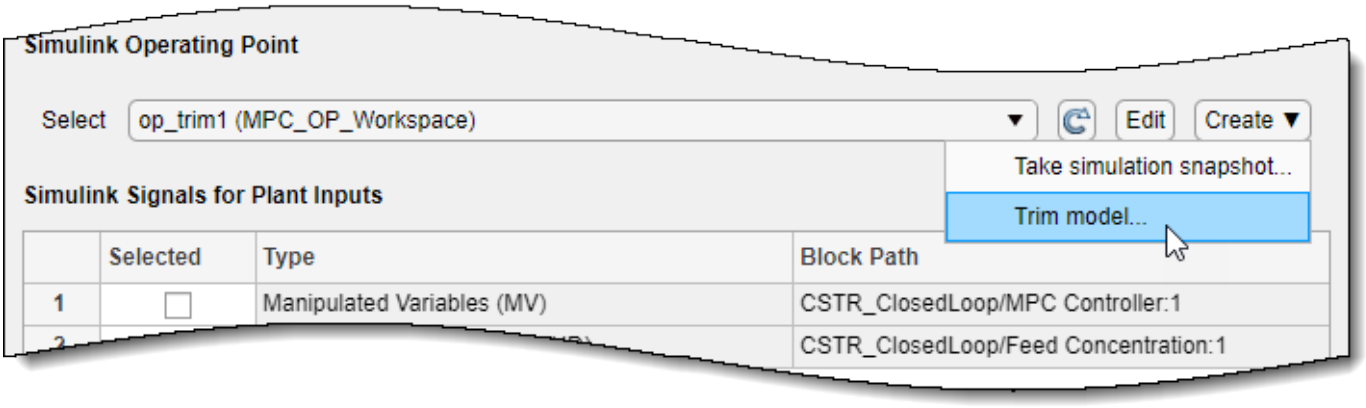
Click **OK**.

If you enter multiple snapshot times, and you previously selected **Linearize at** (and clicked on the **Import** button) from the:

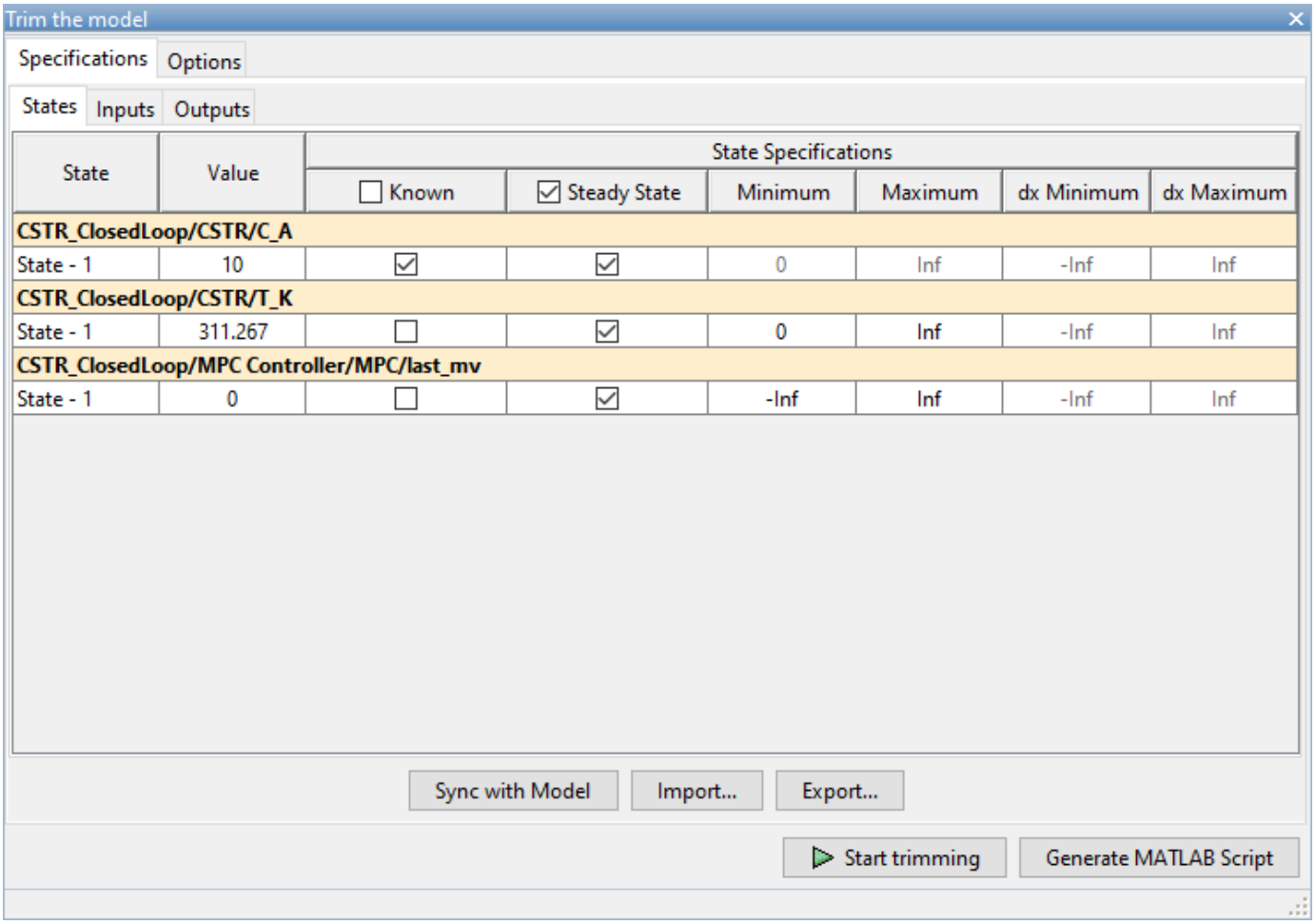
- Define MPC Structure By Linearization dialog box, **MPC Designer** linearizes the Simulink model using only the first snapshot time. The nominal values of the MPC controller are defined using the input/output signal values for this snapshot.
- Linearize Simulink Model dialog box, **MPC Designer** linearizes the model at all the specified snapshot times. The linearized plant models are added to the **Data Browser** in the order specified in the snapshot time array. If you selected the **Use selected operating point to update nominal values as well** option, the nominal values are set using the input/output signal values from the first snapshot.

Compute Steady-State Operating Point

To compute a steady-state operating point using numerical optimization methods to meet your specifications, select **Trim Model** from the **Create** list.



In the Trim the model dialog box, enter the specifications for the steady-state values at which you want to find an operating point. You can specify values for states, input signals, and output signals.



Click **Start Trimming**.

The Trim progress viewer window opens up showing the optimization progress towards finding a point in the state-input space of the model with the characteristics specified in the **States**, **Inputs**,

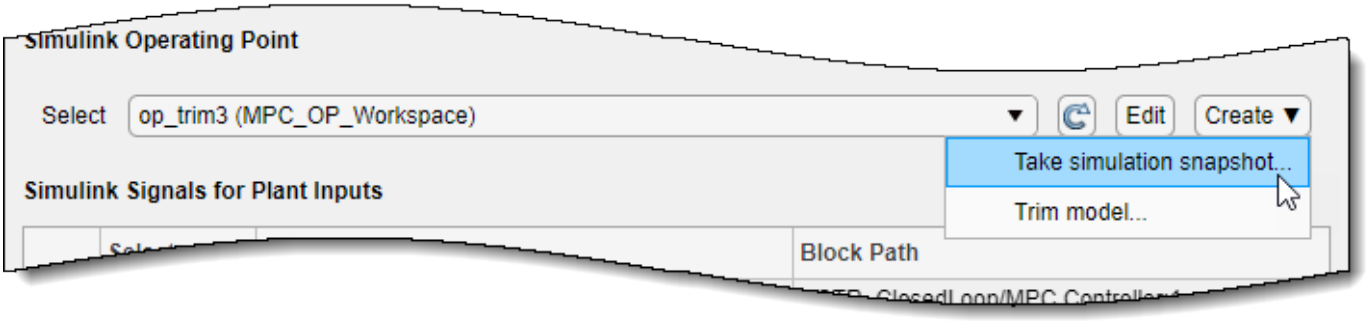
and **Outputs** tabs. After the optimization process terminates, close the trim progress window as well as the Trim the model dialog box.

MPC Designer creates an operating point for the given specifications. The computed operating point is added to the **Simulink Operating Point** drop-down list and is selected.

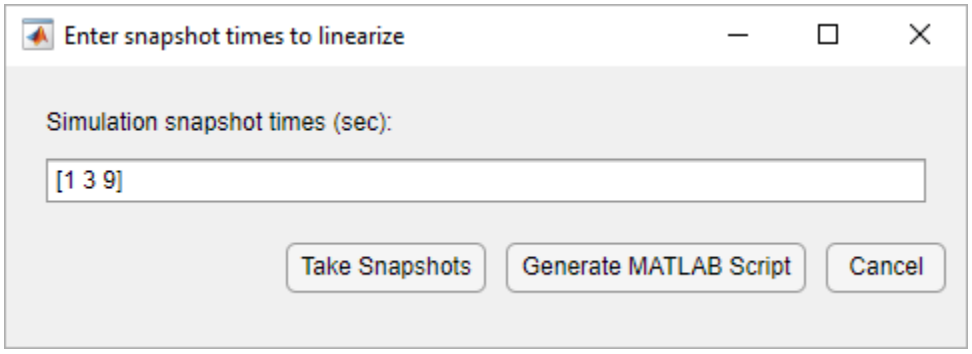
For examples showing how to specify the conditions for a steady-state operating point search, see "Compute Steady-State Operating Points from Specifications" (Simulink Control Design).

Compute Operating Point at Simulation Snapshot Time

To compute operating points using simulation snapshots, select **Take Simulation Snapshot**. Linearizing the model using operating points computed from simulation snapshots is especially useful when you know that your model reaches an equilibrium state after a certain simulation time.



In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



Click **Take Snapshots**.

MPC Designer simulates the Simulink model. At each snapshot time, the current state of the model is used to create an operating point, which is added to the drop-down list and selected.

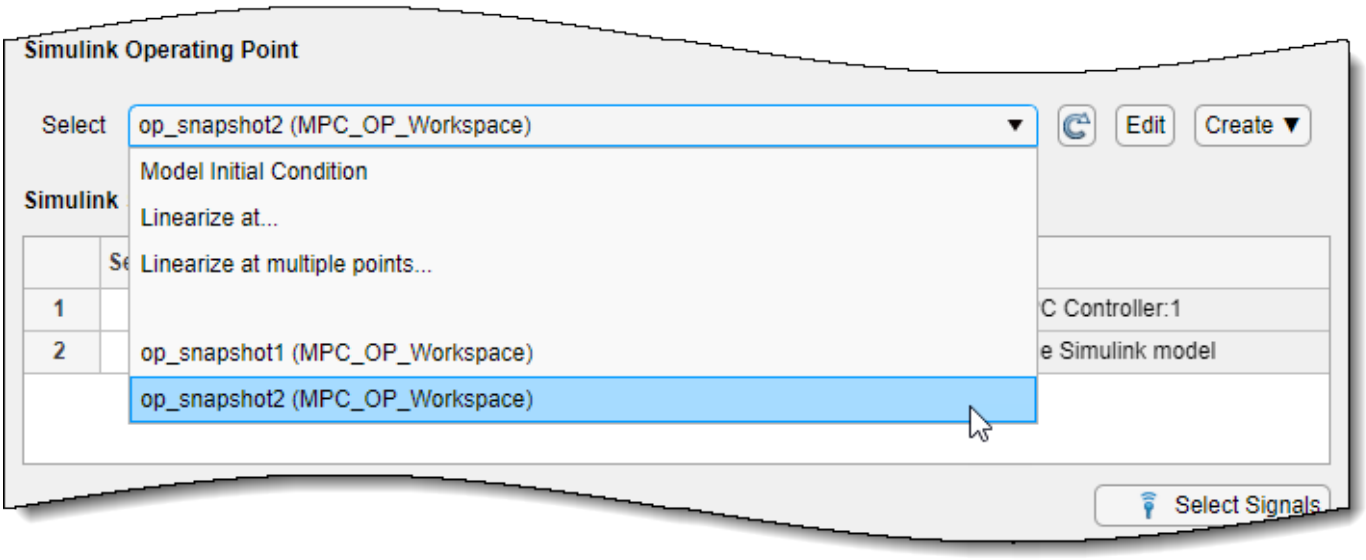
If you enter multiple snapshot times, the operating points are stored together as an array. If you previously selected **Take Simulation Snapshot** from the:

- Define MPC Structure By Linearization dialog box, **MPC Designer** linearizes the model using only the first operating point in the array. The nominal values of the MPC controller are defined using the input/output signal values for this operating point.

- Linearize Simulink Model dialog box, **MPC Designer** linearizes the model at all the operating points in the array. The linearized plant models are added to the **Data Browser** in the same order as the operating point array.

Select Existing Operating Point

Under **Existing Operating Points**, select a previously defined operating point at which to linearize the Simulink model. This option is available if one or more previously created operating points are available in the drop-down list.

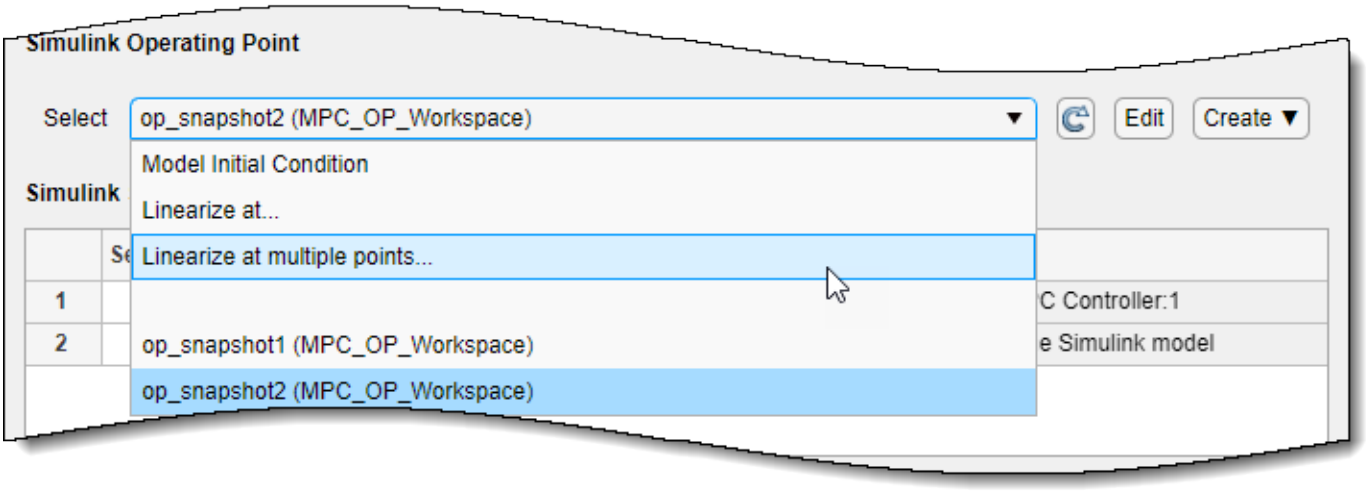


If the selected operating point represents an operating point array created using multiple snapshot times, and you previously selected an operating point from the:

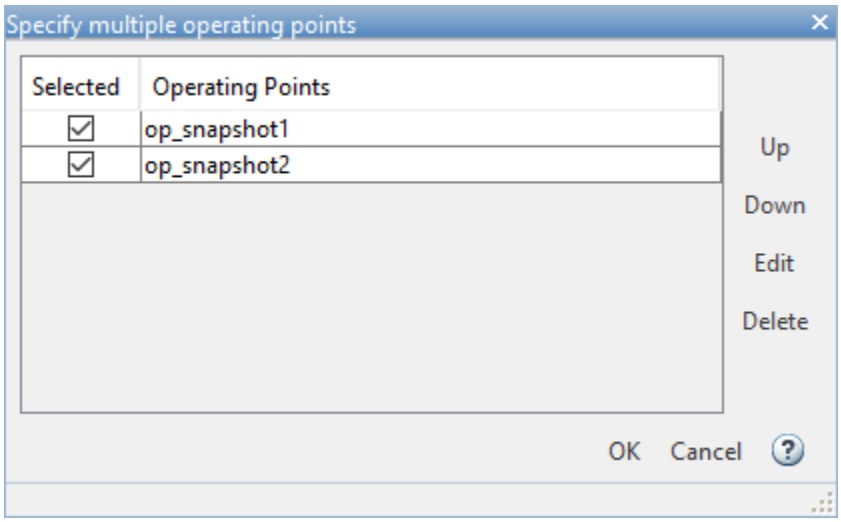
- Define MPC Structure By Linearization dialog box, **MPC Designer** linearizes the model using only the first operating point in the array. The nominal values of the MPC controller are defined using the input/output signal values for this operating point.
- Linearize Simulink Model dialog box, **MPC Designer** linearizes the model at all the operating points in the array. The linearized plant models are added to the **Data Browser** in the same order as the operating point array.

Select Multiple Operating Points

To linearize the Simulink model at multiple existing operating points, select **Linearize at Multiple Points**. This option is available if more than one previously created operating points are in the drop-down list.



In the Specify multiple operating points dialog box, select the operating points at which to linearize the model.



To change the operating point order, click an operating point in the list and click **Up** or **Down** to move the highlighted operating point within the list.

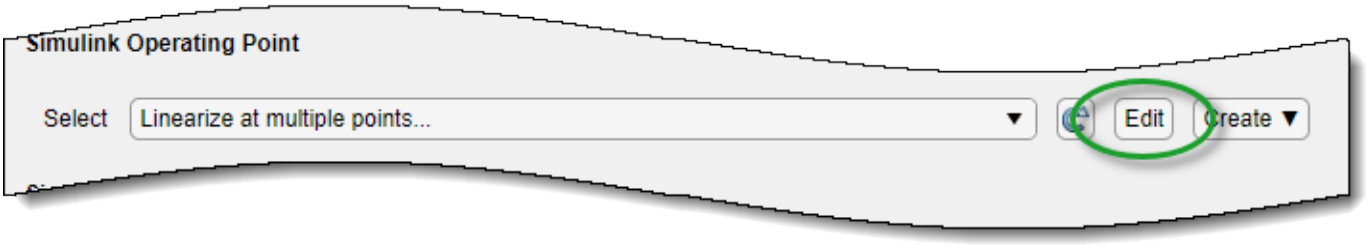
Click **OK**.

If you previously selected **Linearize at Multiple Points** and then clicked **Import** from the:

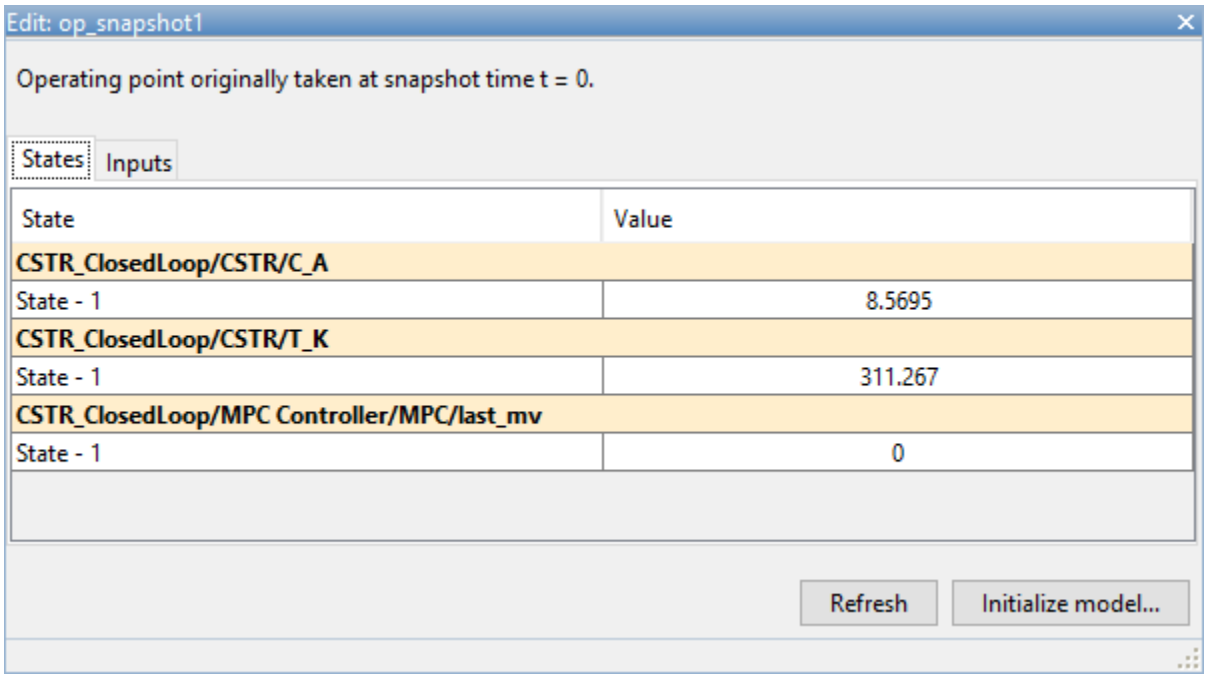
- Define MPC Structure By Linearization dialog box, **MPC Designer** linearizes the model using only the first specified operating point. The nominal values of the MPC controller are defined using the input/output signal values for this operating point.
- Linearize Simulink Model dialog box, **MPC Designer** linearizes the model at all the specified operating points. The linearized plant models are added to the **Data Browser** in the order specified in the Specify multiple operating points dialog box.

View/Edit Operating Point

To view or edit the selected operating point, click the **Edit** button.

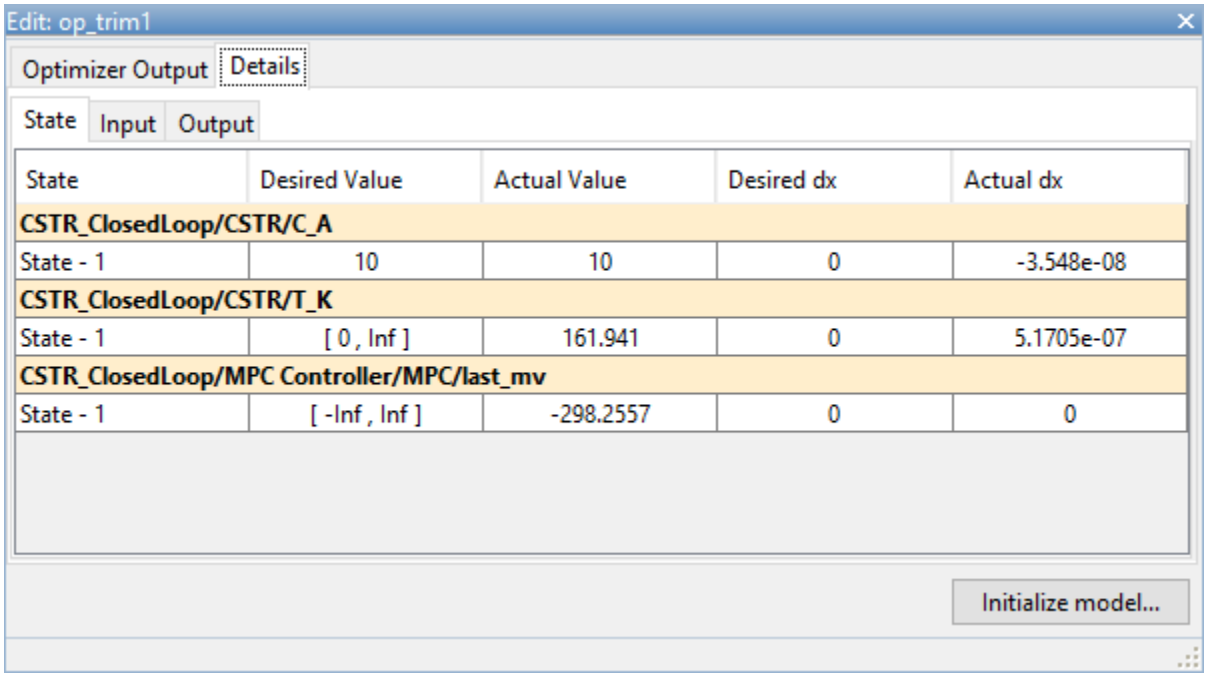


In the Edit dialog box, if you created the selected operating point from a simulation snapshot, you can edit the operating point values.



If the selected operating point represents an operating point array, in the **Select Operating Point** drop-down list, select an operating point to view.

If you obtained the operating point by trimming the model, you can only view the operating point values.



To set the Simulink model initial conditions to the states in the operating point, click **Initialize model**. You can then simulate the model at the specified operating point.

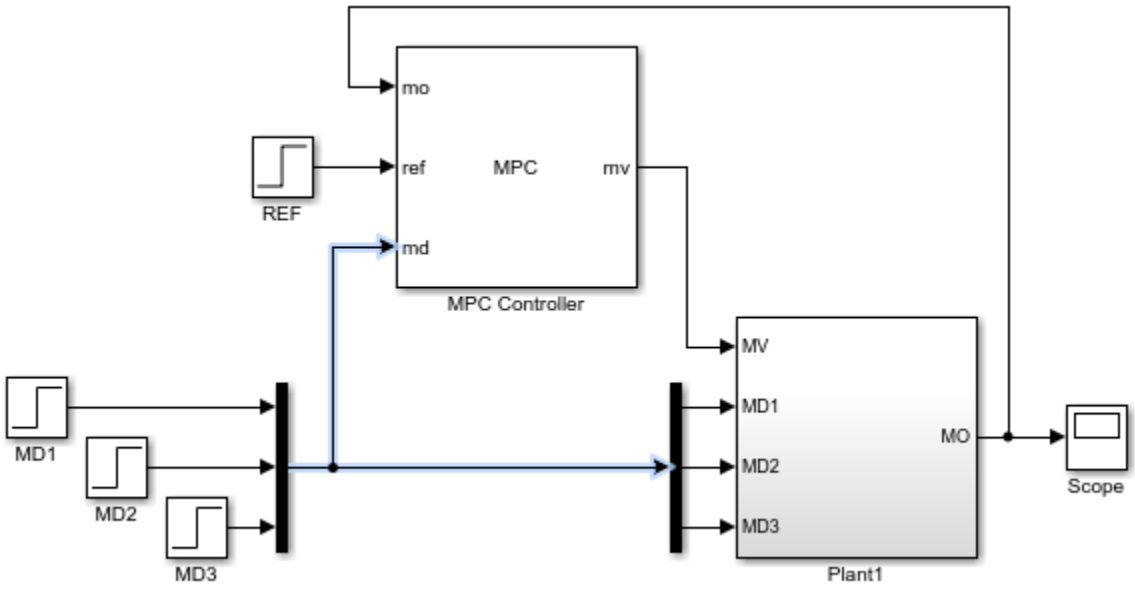
When setting the model initial conditions, **MPC Designer** exports the operating point to the MATLAB workspace. Also, in the Simulink Configuration Parameters dialog box, in the **Data Import/Export** section, it selects the **Input** and **Initial state** parameters and configures them to use the states and inputs in the exported operating point.

To reset the model initial conditions, for example if you delete the exported operating point, clear the **Input** and **Initial state** parameters.

Connect Measured Disturbances for Linearization

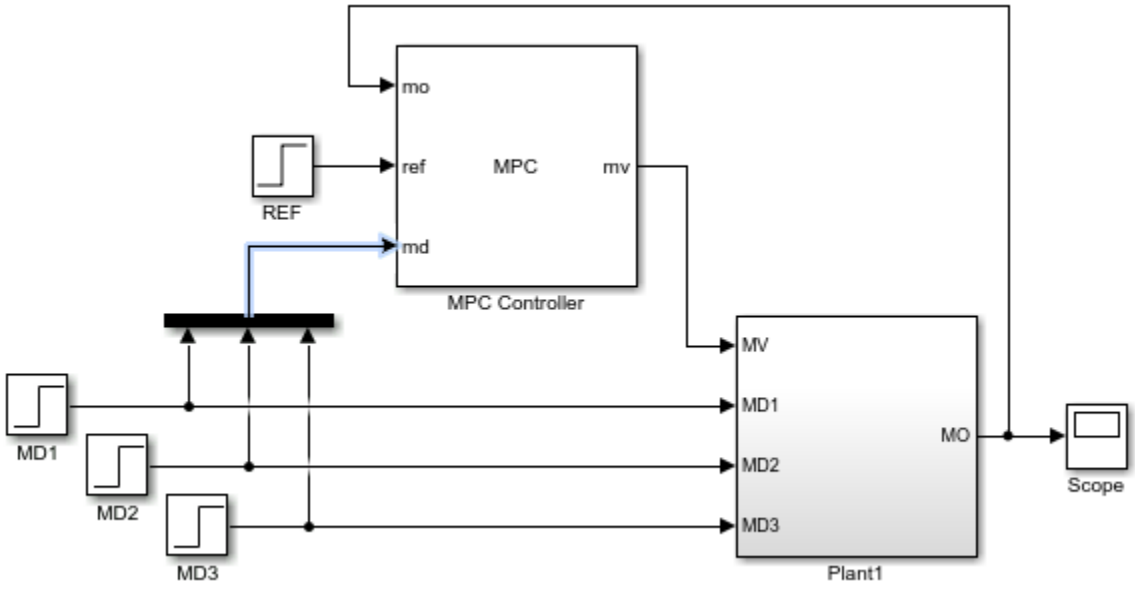
If your Simulink model has measured disturbance signals, connect them to the corresponding plant input ports and to the `md` port of the MPC Controller block. If you have multiple measured disturbances, connect them to the MPC Controller using a vector signal. As discussed in “Define MPC Structure by Linearization” on page 2-32, **MPC Designer** automatically detects the measured disturbances connected to the MPC Controller block and sets them as plant inputs for linearization.

Since the measured disturbances connected to the `md` port are selected as linearization inputs, you must connect the plant measured disturbance input ports to the selected signal line, as shown in the following diagram.



Correct MD Connection

If you connect the plant measured disturbance input ports to the corresponding signals before the Mux block, as shown in the following diagram, there is no linearization path from the signals at the `md` port to the plant. As a result, when you linearize the plant using **MPC Designer**, the measured disturbance channels linearize to zero.



Incorrect MD Connection

See Also

Apps
MPC Designer

Functions
`linearize`

Objects
`mpc`

Related Examples

- “Design MPC Controller in Simulink” on page 3-32

More About

- “Linearize Simulink Models” on page 2-22
- “What is Model Predictive Control?” on page 1-3

Identify Plant from Data

When designing a model predictive controller, you can specify the internal predictive plant model using a linear identified model. You use System Identification Toolbox software to estimate a linear plant model in one of these forms:

- State-space model — `idss`
- Transfer function model — `idtf`
- Polynomial model — `idpoly`
- Process model — `idproc`
- Grey-box model — `idgrey`

You can estimate the plant model programmatically at the command line or interactively using the **System Identification** app.

Identify Plant from Data at the Command Line

This example shows how to identify a plant model at the command line. For information on identifying models using the System Identification app, see “Identify Linear Models Using System Identification App” (System Identification Toolbox).

Load the measured input/output data.

```
load plantIO
```

This command imports the plant input signal, `u`, plant output signal, `y`, and sample time, `Ts` to the MATLAB® workspace.

Create an `iddata` object from the input and output data.

```
mydata = iddata(y,u,Ts);
```

You can optionally assign channel names and units for the input and output signals.

```
mydata.InputName = "Voltage";
mydata.InputUnit = "V";
mydata.OutputName = "Position";
mydata.OutputUnit = "cm";
```

Typically, you must preprocess identification I/O data before estimating a model. For this example, remove the offsets from the input and output signals by detrending the data.

```
mydatad = detrend(mydata);
```

You can also remove offsets by creating an `ssestOptions` object and specifying the `InputOffset` and `OutputOffset` options.

For this example, estimate a second-order, linear state-space model using the detrended data. To estimate a discrete-time model, specify the sample time as `Ts`.

```
ss1 = ssest(mydatad,2,Ts=Ts)
```

```
ss1 =
  Discrete-time identified state-space model:
```

```
    x(t+Ts) = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
            x1       x2
   x1   0.8942  -0.1575
   x2   0.1961   0.7616

  B =
         Voltage
   x1  6.008e-05
   x2   -0.01219

  C =
                  x1       x2
   Position    38.24  -0.3835

  D =
             Voltage
   Position        0

  K =
       Position
   x1   0.03572
   x2    0.0223

Sample time: 0.1 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 10
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "mydatad".
Fit to estimation data: 89.85% (prediction focus)
FPE: 0.0156, MSE: 0.01541
```

You can use this identified plant as the internal prediction model for your MPC controller. When you do so, the controller converts the identified model to a discrete-time, state-space model.

By default, the MPC controller discards any unmeasured noise components from your identified model. To configure noise channels as unmeasured disturbances, you must first create an augmented state-space model from your identified model. For example:

```
ss2 = ss(ss1,"augmented")

ss2 =

  A =
            x1       x2
   x1   0.8942  -0.1575
   x2   0.1961   0.7616

  B =
```

```
          Voltage  v@Position
   x1   6.008e-05    0.004448
   x2    -0.01219    0.002777

  C =
                  x1       x2
   Position    38.24  -0.3835

  D =
                Voltage  v@Position
   Position           0      0.1245

Input groups:
      Name      Channels
    Measured       1
     Noise         2

Sample time: 0.1 seconds
Discrete-time state-space model.
```

This command creates a state-space model, `ss2`, with two input groups, `Measured` and `Noise`, for the measured and noise inputs respectively. When you import the augmented model into your MPC controller, channels in the `Noise` input group are defined as unmeasured disturbances.

Working with Impulse-Response Models

You can use System Identification Toolbox software to estimate finite step-response or finite impulse-response (FIR) plant models using measured data. Such models, also known as nonparametric models, are easy to determine from plant data ([1] and [2]) and have intuitive appeal.

Use the `impulseest` function to estimate an FIR model from measured data. This function generates the FIR coefficients encapsulated as an `idtf` object; that is, a transfer function model with only numerator coefficients. `impulseest` is especially effective in situations where the input signal used for identification has low excitation levels. To design a model predictive controller for this plant, you can convert the identified FIR plant model to a numeric LTI model. However, this conversion usually yields a high-order plant, which can degrade the controller design. For example, the numerical precision issues with high-order plants can affect estimator design. This result is particularly an issue for MIMO systems.

Model predictive controllers work best with low-order parametric models. Therefore, to design a model predictive controller using measured plant data, you can:

- Estimate a low-order parametric model using a parametric estimator, such as `ssest`.
- Initially identify a nonparametric model using `impulseest`, and then estimate a low-order parametric model from the response of the nonparametric model. For an example, see [3].
- Initially identify a nonparametric model using `impulseest`, and then convert the FIR model to a state-space model using `idss`. You can then reduce the order of the state-space model using `balred`. This approach is similar to the method used by `ssregest`.

References

[1] Cutler, C., and F. Yocum, "Experience with the DMC inverse for identification," *Chemical Process Control — CPC IV* (Y. Arkun and W. H. Ray, eds.), CACHE, 1991.

[2] Ricker, N. L., "The use of bias least-squares estimators for parameters in discrete-time pulse response models," *Ind. Eng. Chem. Res.*, Vol. 27, pp. 343, 1988.

[3] Wang, L., P. Gawthrop, C. Chessari, T. Podsiadly, and A. Giles, "Indirect approach to continuous time system identification of food extruder," *J. Process Control*, Vol. 14, Number 6, pp. 603–615, 2004.

See Also

Apps
System Identification

Functions
`iddata` | `detrend` | `ssest`

Related Examples

- “Design MPC Controller for Identified Plant Model”

More About

- “Handling Offsets and Trends in Data” (System Identification Toolbox)
- “Identify Linear Models Using System Identification App” (System Identification Toolbox)

Design MPC Controllers

Design Controller Using MPC Designer

This example shows how to design a model predictive controller for a continuous stirred-tank reactor (CSTR) using **MPC Designer**.

CSTR Model

The linearized model of a Continuously Stirred Tank Reactor (CSTR) is shown in “CSTR Model” on page 2-8. In the model, the first two state variables are the concentration of reagent (here referred to as C_A and measured in kmol/m^3) and the temperature of the reactor (here referred to as T, measured in K), while the first two inputs are the coolant temperature (T_c, measured in K, used to control the plant), and the inflow feed reagent concentration C_{A_f} measured in kmol/m^3, (often considered as unmeasured disturbance).

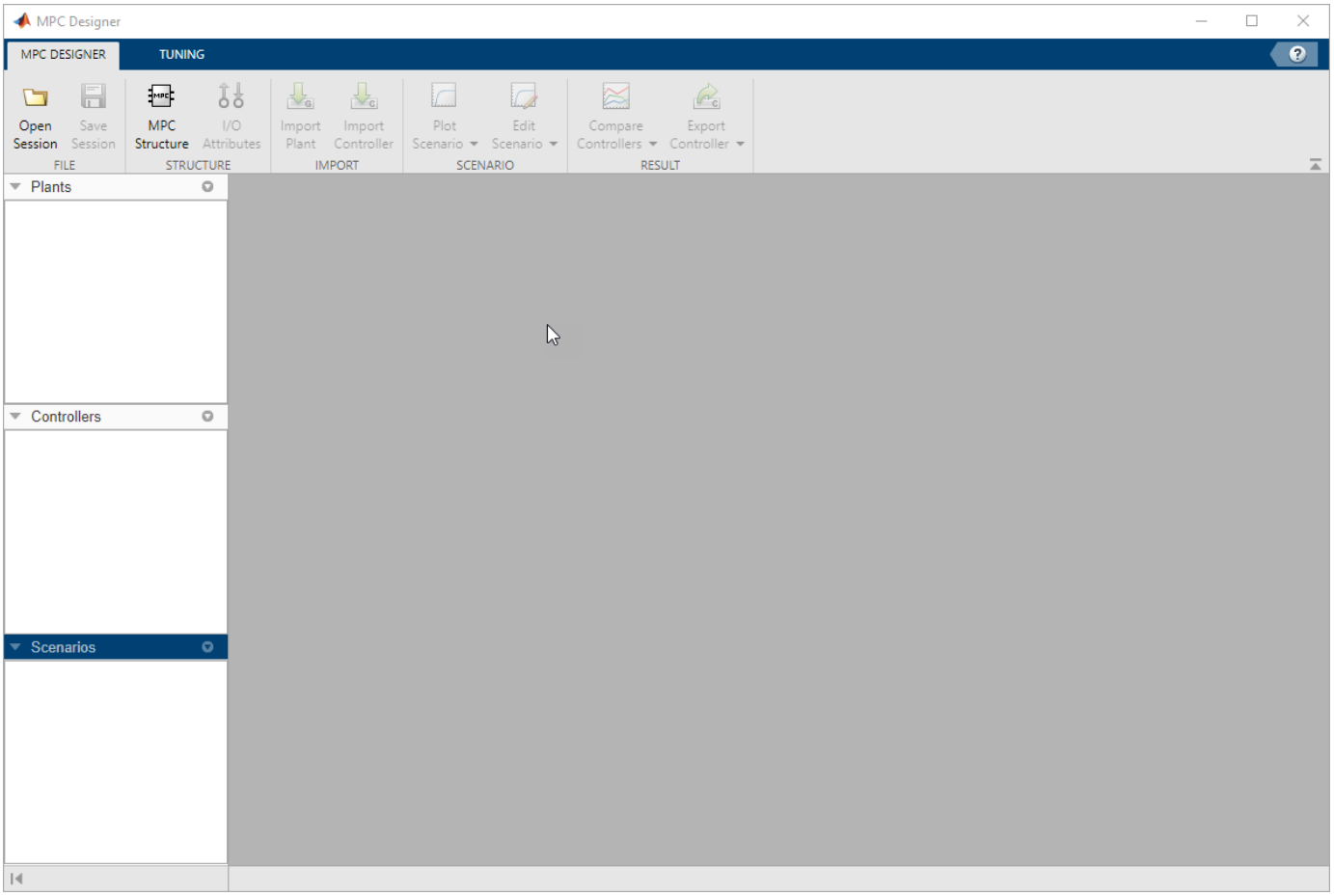
For this example, the coolant temperature has a limited range of ±10 degrees from its nominal value and a limited rate of change of ±2 degrees per second.

Create a state-space model of a CSTR system.

```
A = [   -5  -0.3427;
     47.68    2.785];
B = [    0   1
       0.3   0];
C = flipud(eye(2));
D = zeros(2);
CSTR = ss(A,B,C,D);
```

Import Plant and Define MPC Structure

```
mpcDesigner
```



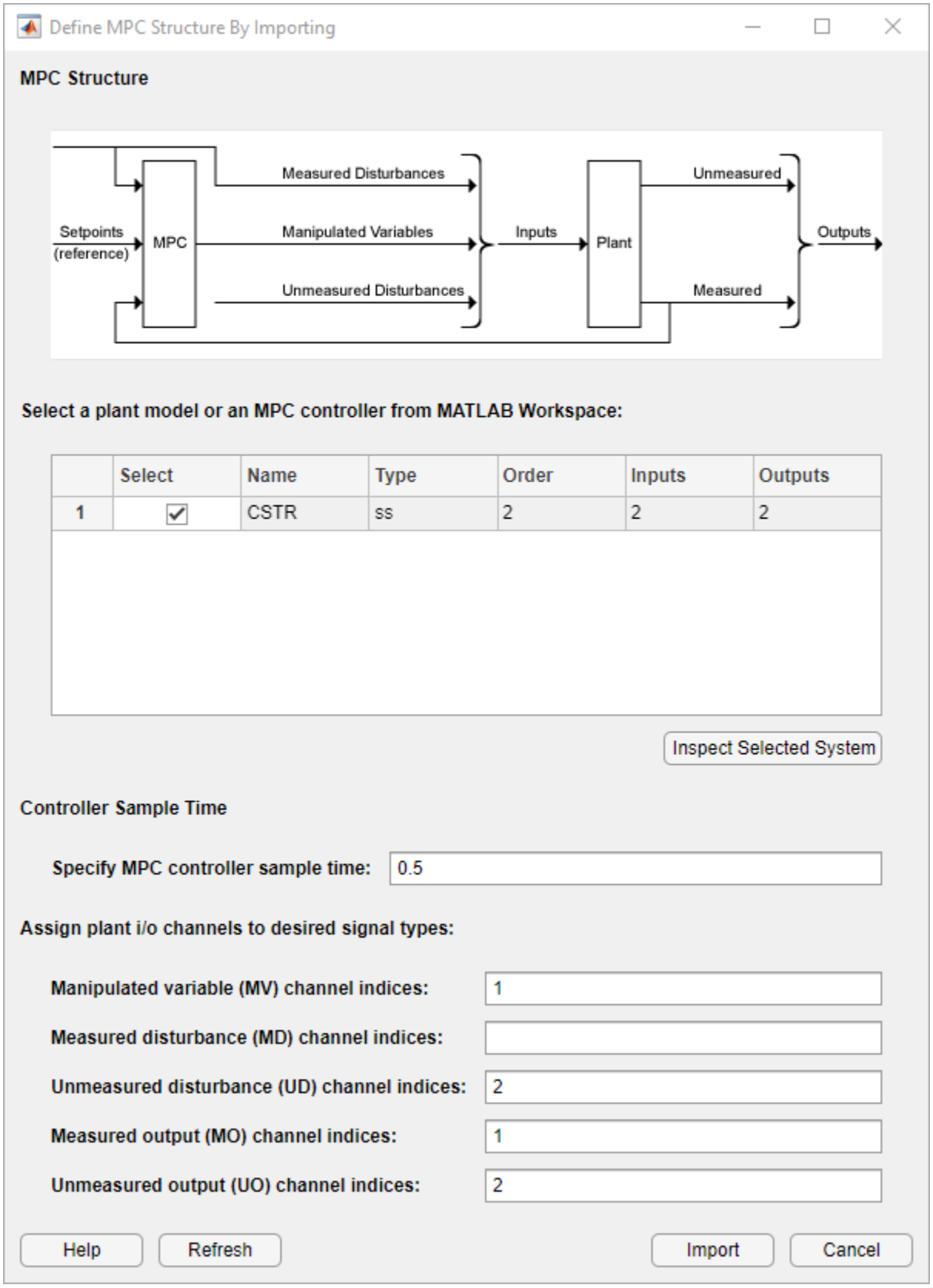
On the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.

In the Define MPC Structure By Importing dialog box, in the **Select a plant model or an MPC controller from MATLAB workspace** table, select the `CSTR` model.

Since `CSTR` 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 `CSTR`. For this example, in the **Specify MPC controller sample time** field, enter a sample time of `0.5` seconds.

By default, all plant inputs are defined as manipulated variables and all plant outputs as measured outputs. In the **Assign plant i/o channels** section, assign the input and output channel indices such that:

- The first input, coolant temperature, is a manipulated variable.
- The second input, feed concentration, is an unmeasured disturbance.
- The first output, reactor temperature, is a measured output.
- The second output, reactant concentration, is an unmeasured output.

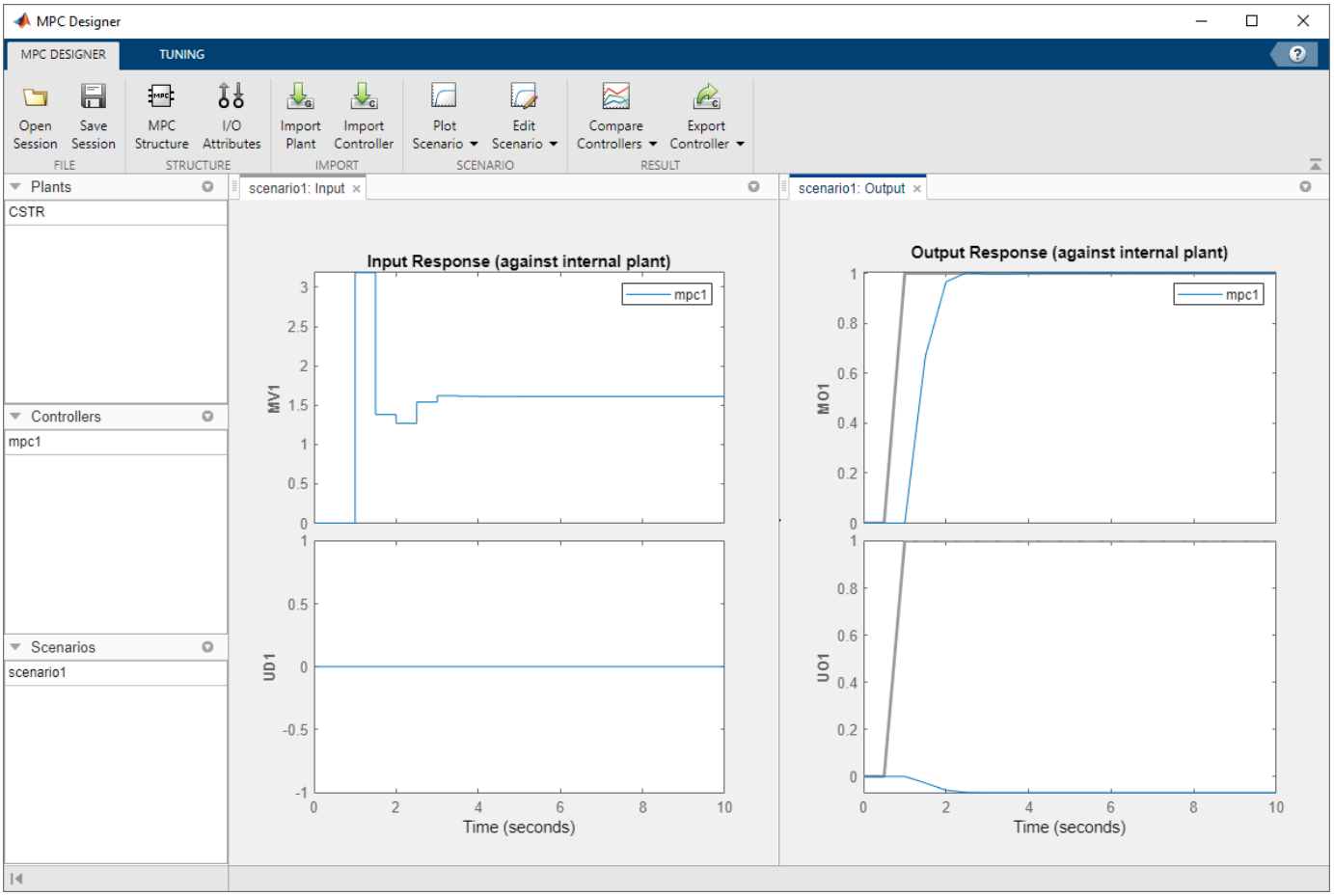

Define MPC Structure By Importing
MPC Structure
Measured Disturbances
Unmeasured
Setpoints
(reference)
MPC
Manipulated Variables
Inputs
Plant
Outputs
Unmeasured Disturbances
Measured
Select a plant model or an MPC controller from MATLAB Workspace:
Select
Name
Type
Order
Inputs
Outputs
1
CSTR
ss
2
2
2
Inspect Selected System
Controller Sample Time
Specify MPC controller sample time:
0.5
Assign plant i/o channels to desired signal types:
Manipulated variable (MV) channel indices:
1
Measured disturbance (MD) channel indices:
Unmeasured disturbance (UD) channel indices:
2
Measured output (MO) channel indices:
1
Unmeasured output (UO) channel indices:
2
Help
Refresh
Import
Cancel

Click **Import**.

The app imports the `CSTR` plant to the **Data Browser**. The following are also added to the **Data Browser**:

- `mpc1` — Default MPC controller created using `CSTR` as its internal model.
- `scenario1` — Default simulation scenario.

The app runs the default simulation scenario and updates the **Input Response** and **Output Response** plots. The closed loop system is able to track the desired measured output successfully, while this is not the case for the unmeasured output. This behavior is expected as the plant has only one manipulated variable.



Once you define the MPC structure, you cannot change it within the current **MPC Designer** session. To use a different channel configuration, start a new session of the app.

Define Input and Output Channel Attributes

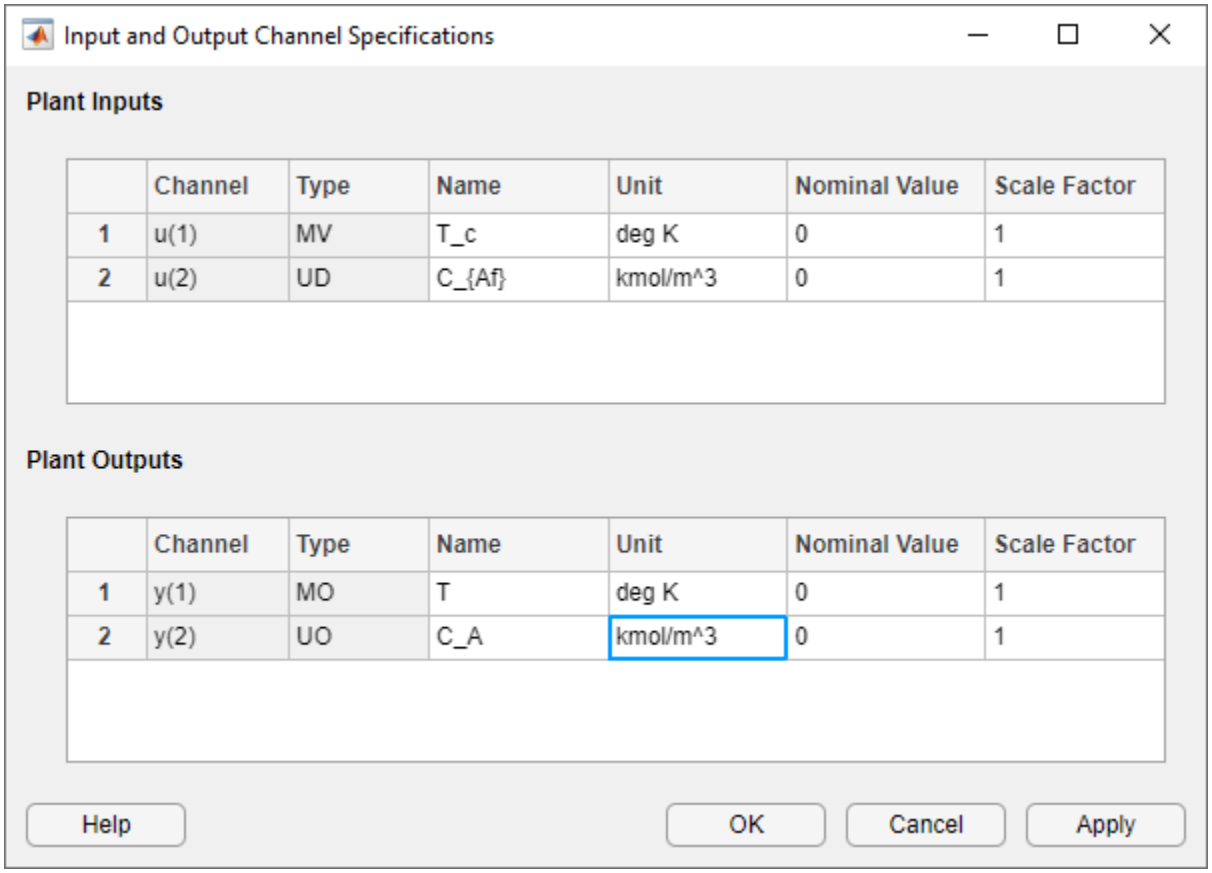
On the **MPC Designer** tab, select **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, in the **Name** column, specify a meaningful name for each input and output channel.

In the **Unit** column, optionally specify the units for each channel.

Since the state-space model is defined using deviations from the nominal operating point, keep the **Nominal Value** for each input and output channel to `0`.

Keep the **Scale Factor** for each channel at the default value of `1`.



Click **OK**.

The **Input Response** and **Output Response** plot labels update to reflect the new signal names and units.

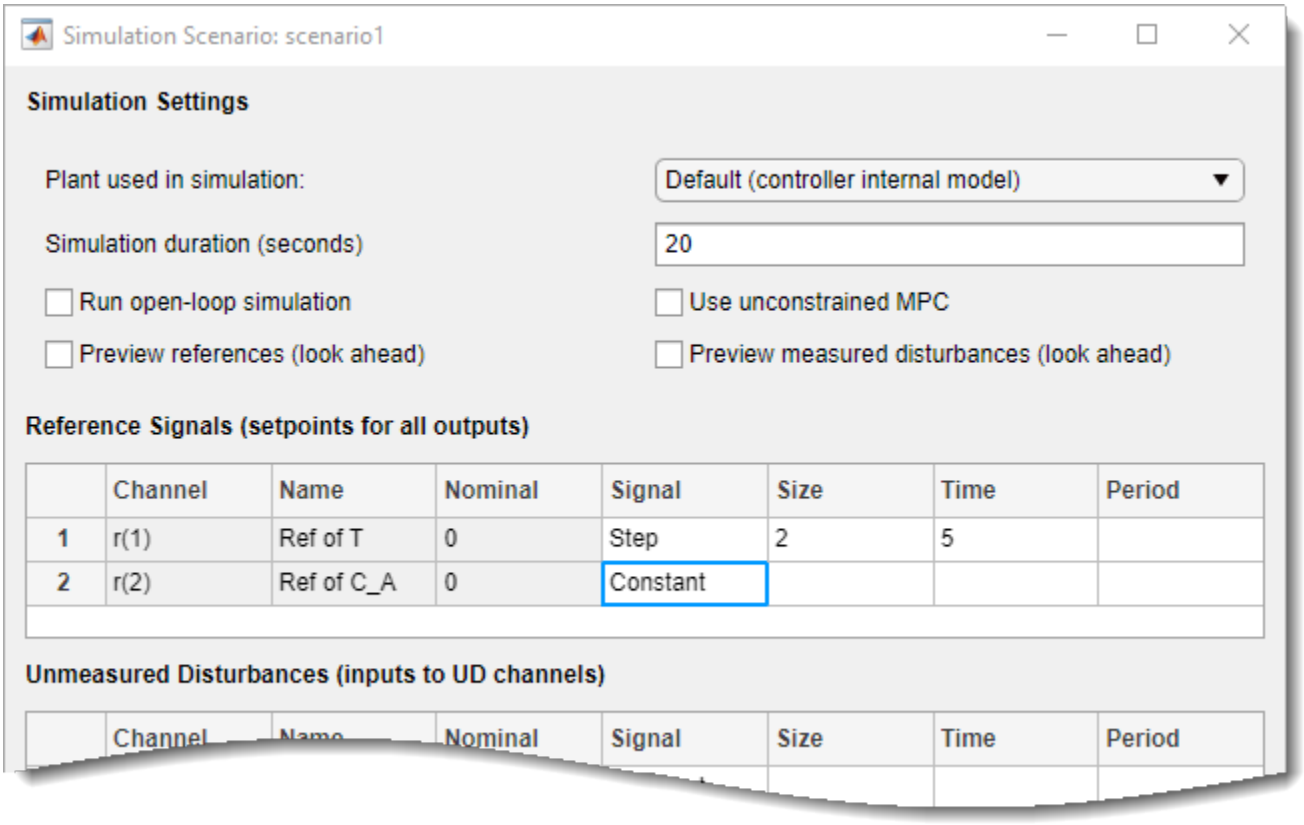
Configure Simulation Scenario

On the **MPC Designer** tab, in the **Scenario** section, click **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, set the **Simulation duration** to 20 seconds.

In the **Reference Signals** table, in the first row, specify a step **Size** of 2 and a **Time** of 5.

In the **Signal** column, in the second row, select a `Constant` reference to hold the concentration setpoint at its nominal value, defined in the Input and Output Channel Specifications dialog box (in this case the nominal value is zero).

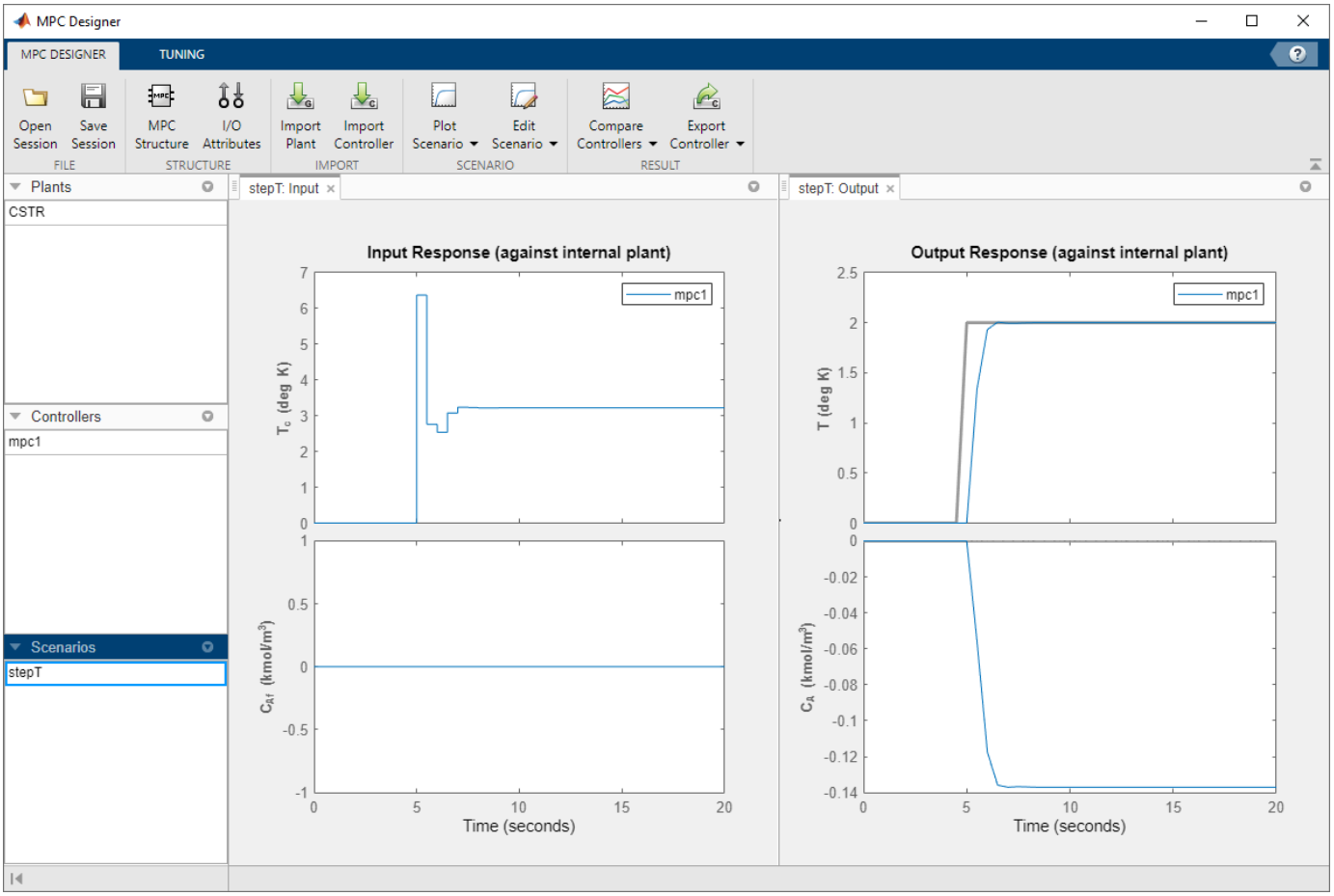


The default scenario is configured to simulate a step change of 2 degrees Kelvin in the reference reactor temperature, T, at a time of 5 seconds.

Click **OK**.

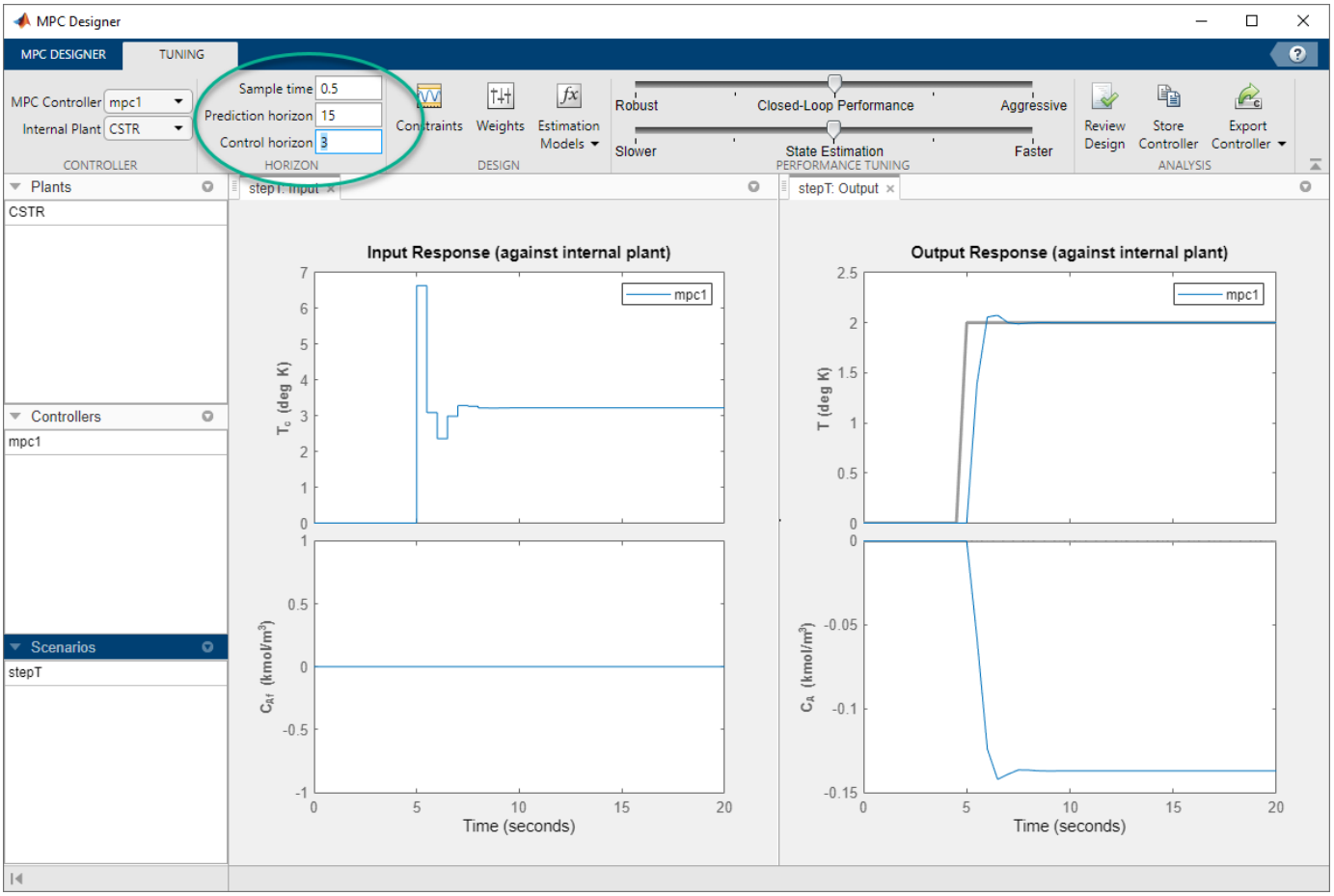
The response plots update to reflect the new simulation scenario configuration. The reference value for C_A is no longer a step but a constant equal to zero.

In the **Scenarios** section in the lower left part of **MPC Designer**, click `scenario1`. Click `scenario1` a second time, and rename the scenario to `stepT`.



Configure Controller Horizons

On the **Tuning** tab, in the **Horizons** section, specify a **Prediction horizon** of 15 and a **Control horizon** of 3.



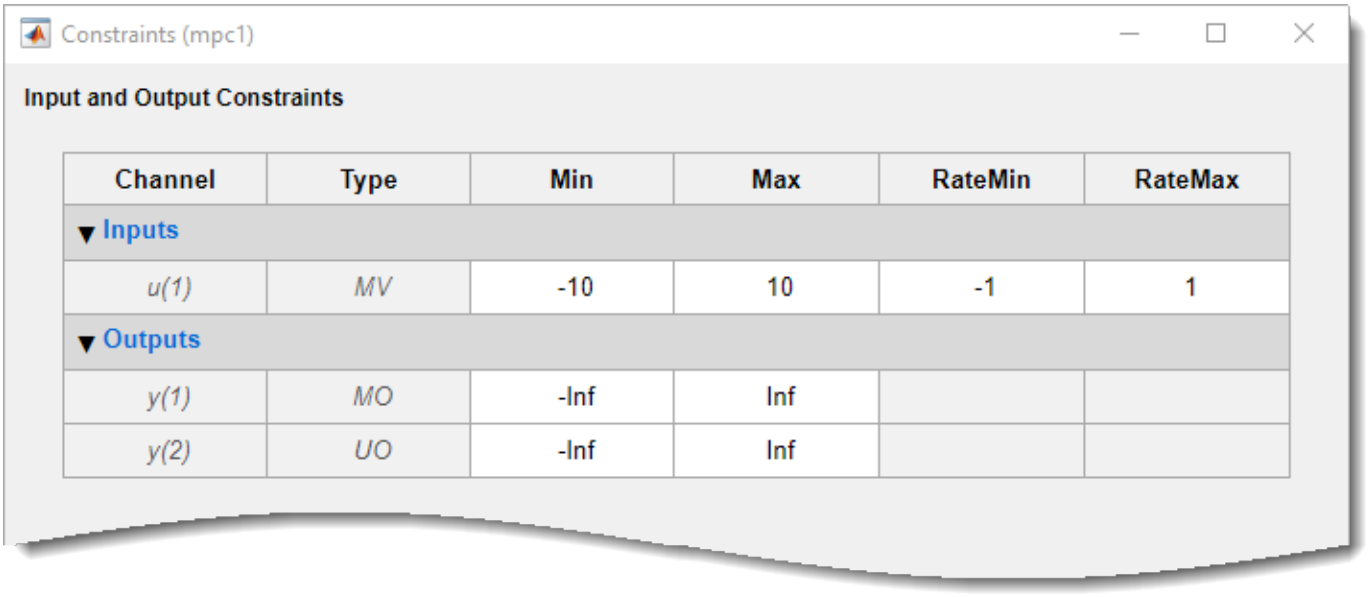
The response plots update to reflect the new horizons. The **Input Response** plot shows that the control actions violate the required constraint on the rate of change for the coolant temperature.

Define Input Constraints

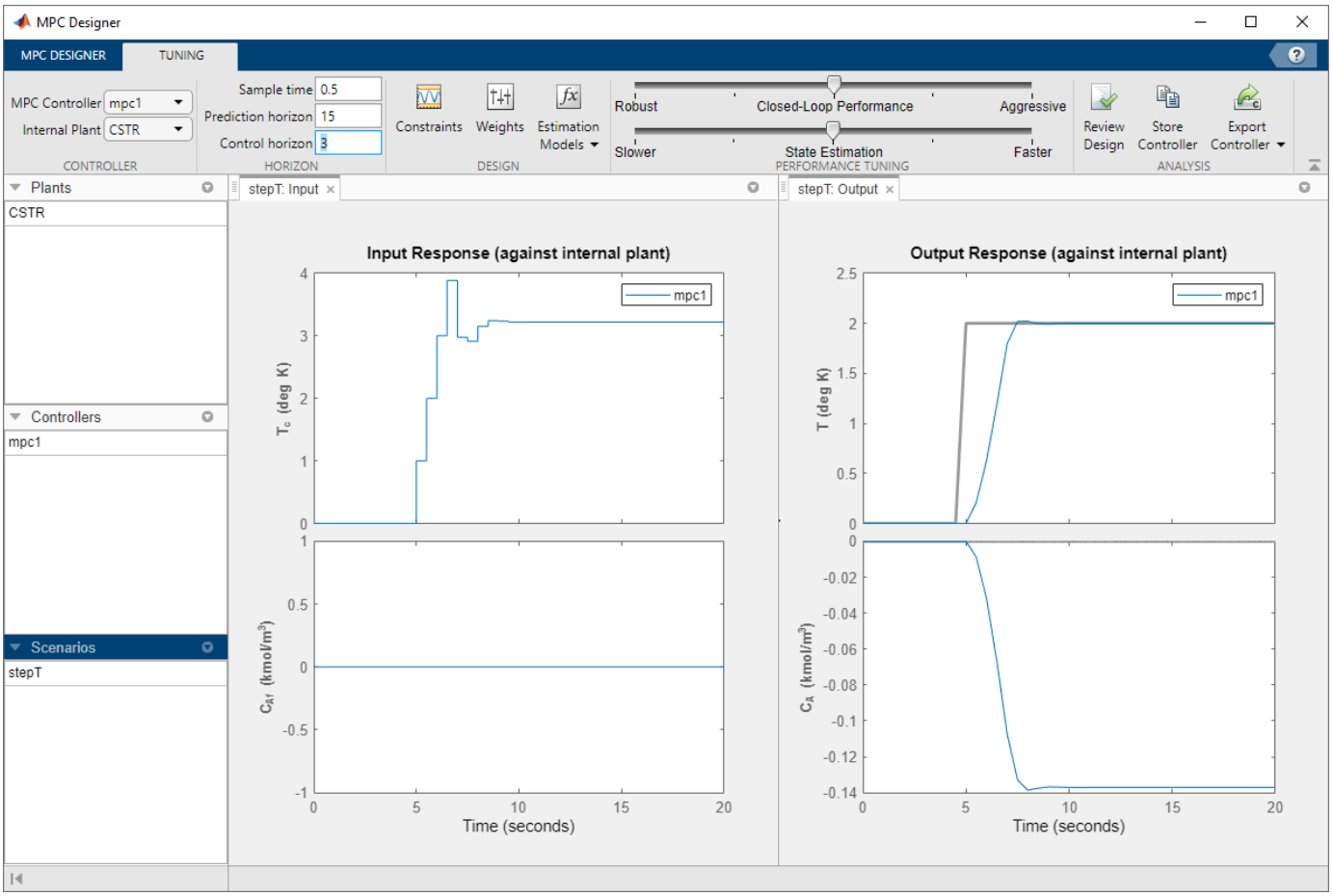
In the **Design** section, click **Constraints**.

In the Constraints dialog box, in the **Input and Output Constraints** section, in the **Inputs** row, enter the coolant temperature upper and lower bounds in the **Min** and **Max** columns respectively.

Specify the rate of change limits in the **RateMin** and **RateMax** columns.


Constraints (mpc1)
Input and Output Constraints
Channel
Type
Min
Max
RateMin
RateMax
Inputs
u(1)
MV
-10
10
-1
1
Outputs
y(1)
MO
-Inf
Inf
y(2)
UO
-Inf
Inf

Click **OK**.


MPC Designer
MPC DESIGNER
TUNING
MPC Controller mpc1
Internal Plant CSTR
CONTROLLER
Sample time 0.5
Prediction horizon 15
Control horizon 3
HORIZON
Constraints
Weights
Estimation Models
DESIGN
Robust
Closed-Loop Performance
Aggressive
Slower
State Estimation
Faster
PERFORMANCE TUNING
Review Design
Store Controller
Export Controller
ANALYSIS
Plants
CSTR
Controllers
mpc1
Scenarios
stepT
stepT: Input
stepT: Output
Input Response (against internal plant)
Output Response (against internal plant)
mpc1
Tc (deg K)
CAf (kmol/m3)
T (deg K)
CA (kmol/m3)
Time (seconds)

The **Input Response** plot shows the constrained manipulated variable control actions.

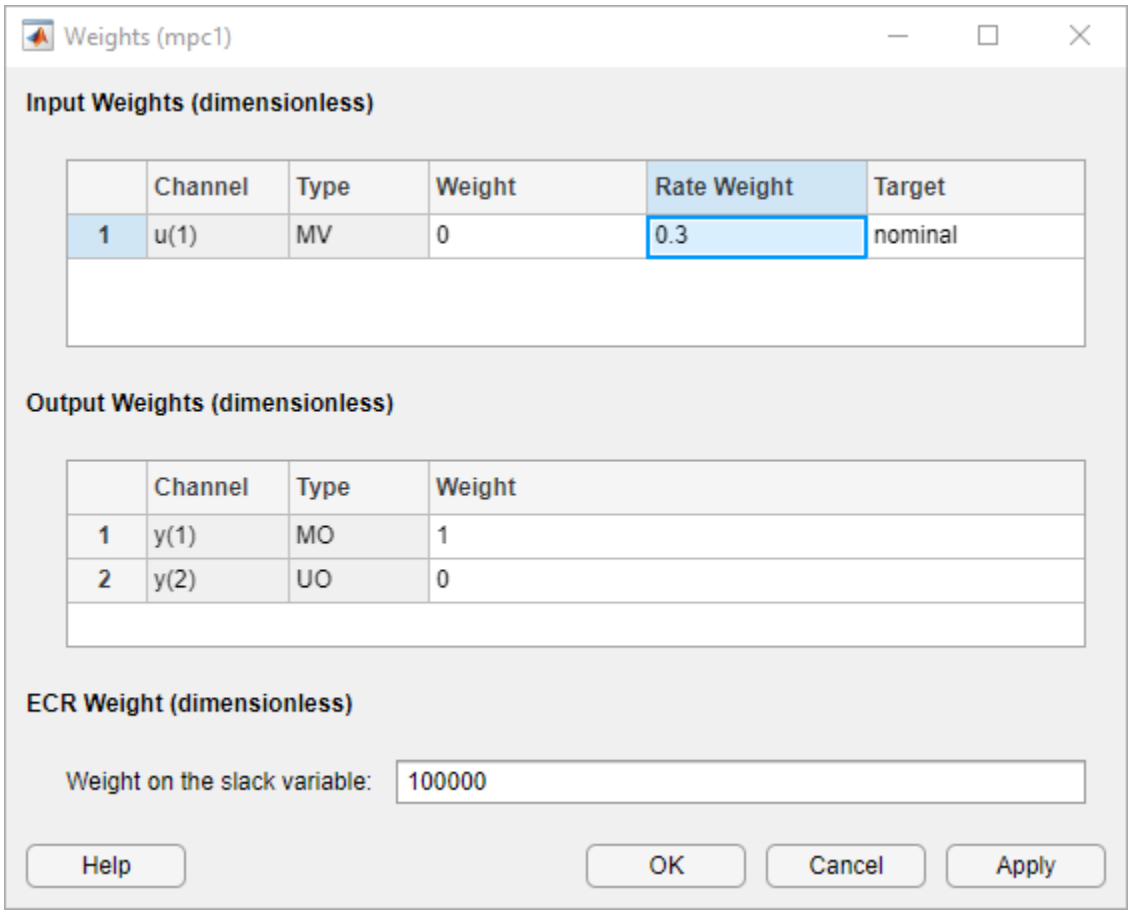
Specify Controller Tuning Weights

On the **Tuning** tab, in the **Design** section, click **Weights**.

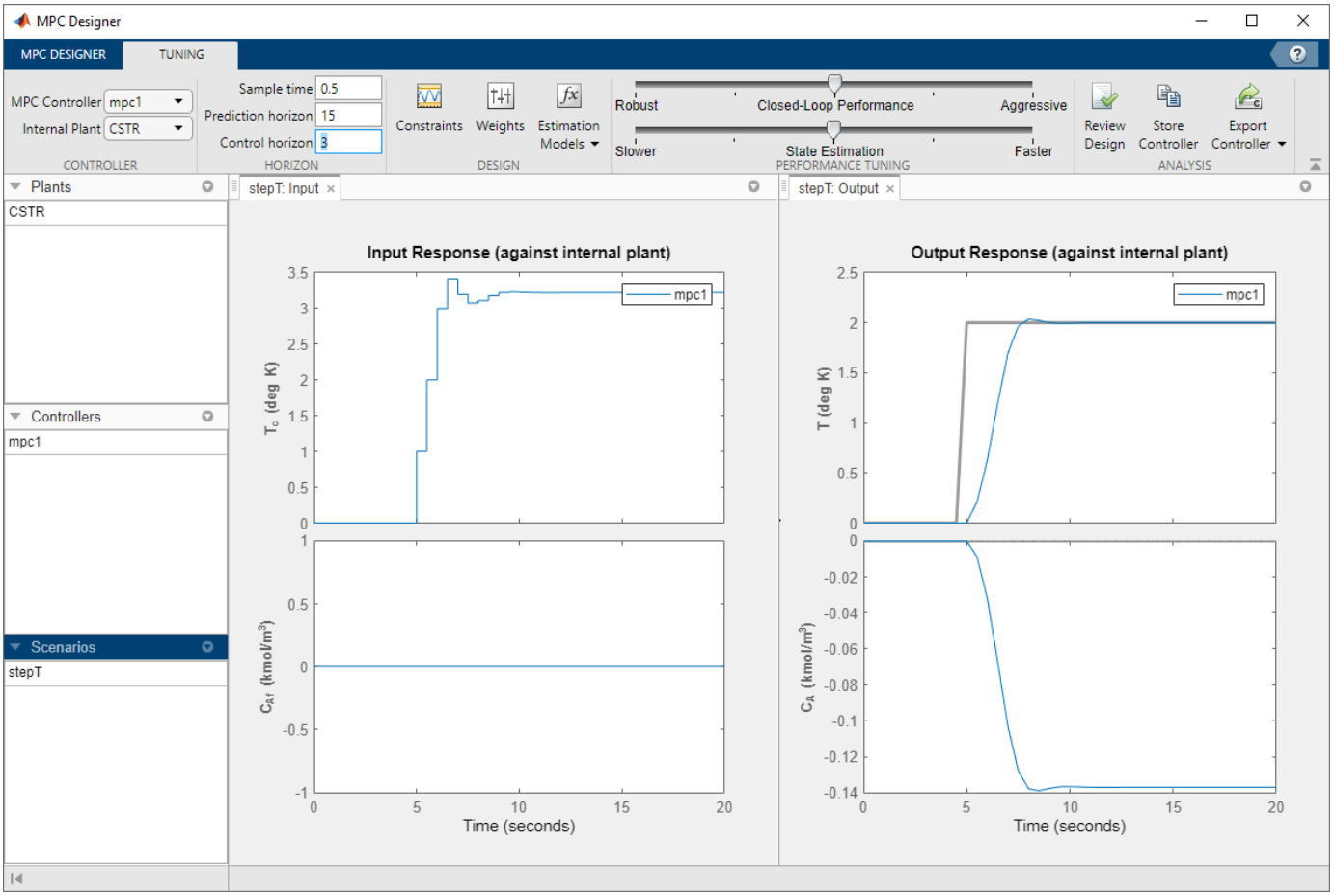
In the **Input Weights** table, increase the manipulated variable (MV) **Rate Weight** to `0.3`. Increasing the MV rate weight penalizes large MV changes in the controller optimization cost function.

In the **Output Weights** table, keep the default **Weight** values. By default, all unmeasured outputs have zero weights.

Since there is only one manipulated variable, if the controller tries to hold both outputs at specific setpoints, one or both outputs will exhibit steady-state error in their responses. Since the controller ignores setpoints for outputs with zero weight, setting the concentration output weight to zero allows reactor temperature setpoint tracking with zero steady-state error.



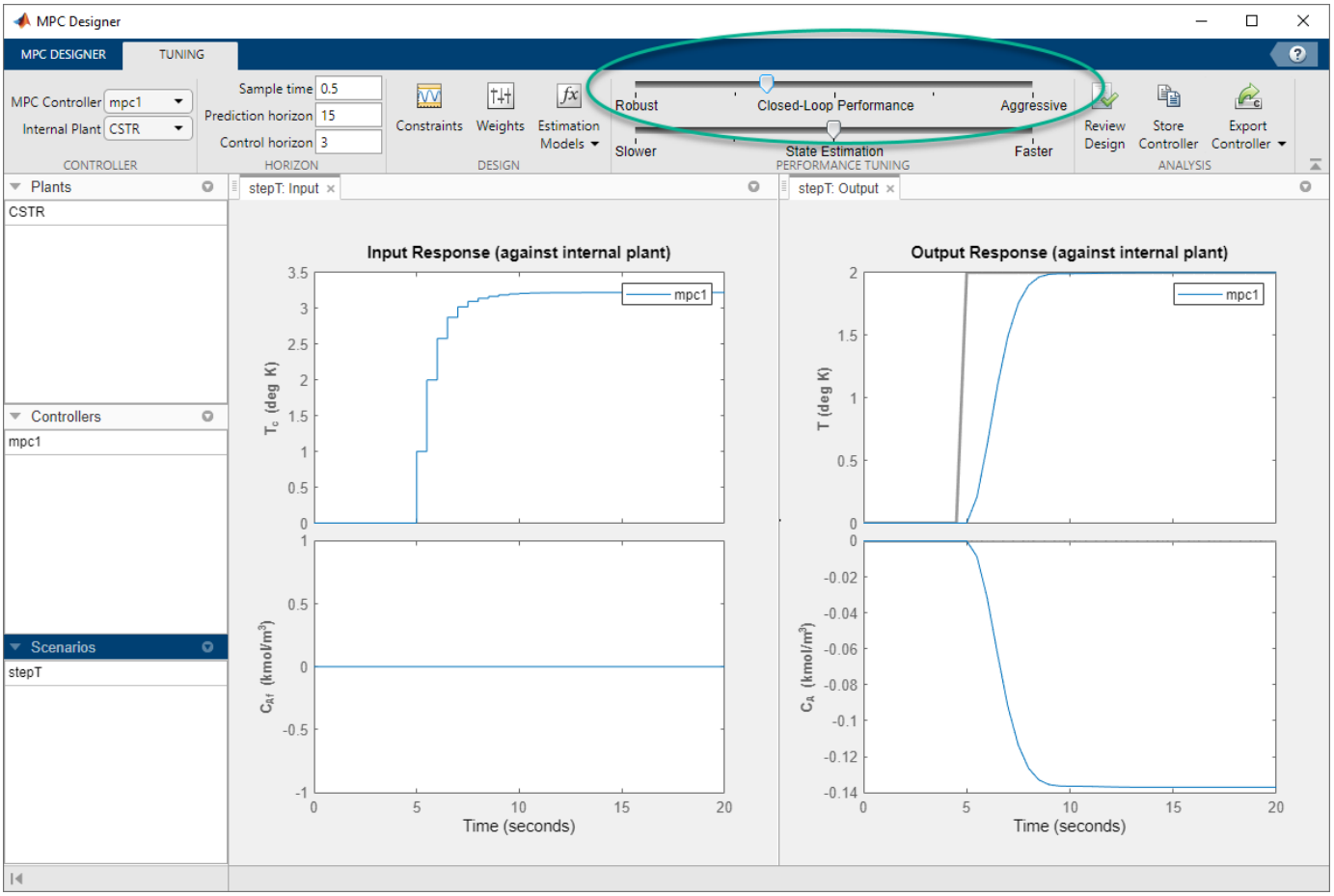
Click **OK**.



The **Input Response** plot shows the more conservative control actions, which result in a slower **Output Response**.

Eliminate Output Overshoot

Suppose the application demands zero overshoot in the output response. On the **Performance Tuning** tab, drag the **Closed-Loop Performance** slider to the left until the **Output Response** has no overshoot. Moving this slider to the left simultaneously increases the manipulated variable rate weight of the controller and decreases the output variable weight, producing a more robust controller.



When you adjust the controller tuning weights using the **Closed-Loop Performance** slider, **MPC Designer** does not change the weights you specified in the Weights dialog box. Instead, the slider controls an adjustment factor, which is used with the user-specified weights to define the actual controller weights.

This factor is `1` when the slider is centered; its value decreases as the slider moves left and increases as the slider moves right. The weighting factor multiplies the manipulated variable and output variable weights and divides the manipulated variable rate weights from the Weights dialog box. Therefore, moving the slider to increase robustness decreases both OV and MV weights and increases MV Rate weights, which leads to relaxed control of outputs and more conservative control moves.

To view the actual controller weights, export the controller to the MATLAB workspace, and view the `Weights` property of the exported controller object.

Test Controller Disturbance Rejection

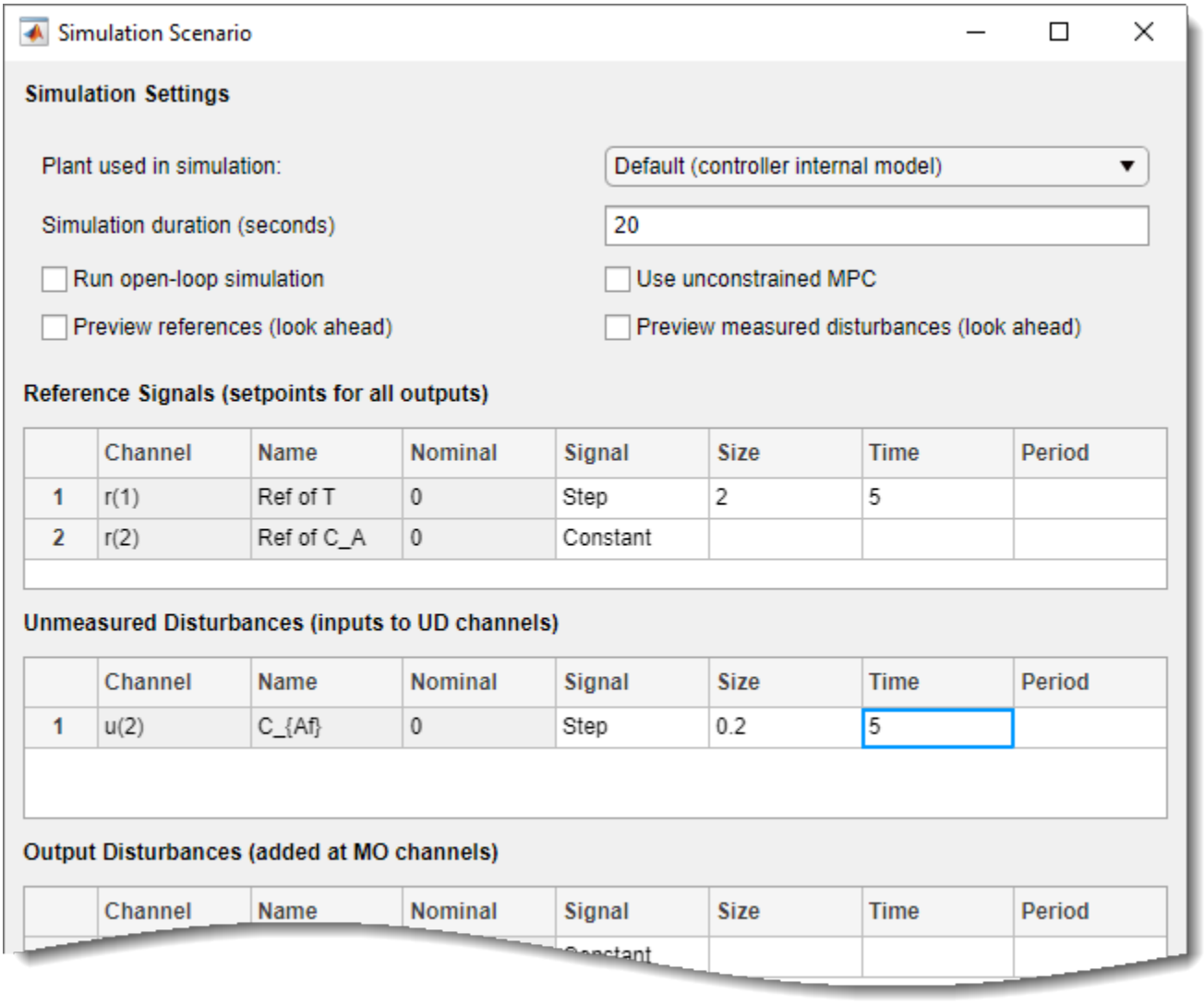
In a process control application, disturbance rejection is often more important than setpoint tracking. Simulate the controller response to a step change in the feed concentration unmeasured disturbance.

On the **MPC Designer** tab, in the **Scenario** section, click **Plot Scenario > New Scenario**.

In the Simulation Scenario dialog box, set the **Simulation duration** to 20 seconds.

In the **Reference Signals** table, in the first row, in the **Signal** drop-down list, select `Step`, then specify a step **Size** of 2, and a **Time** of 5. In the **Signal** column, in the second row, keep a `Constant` reference to hold the concentration setpoint at its nominal value.

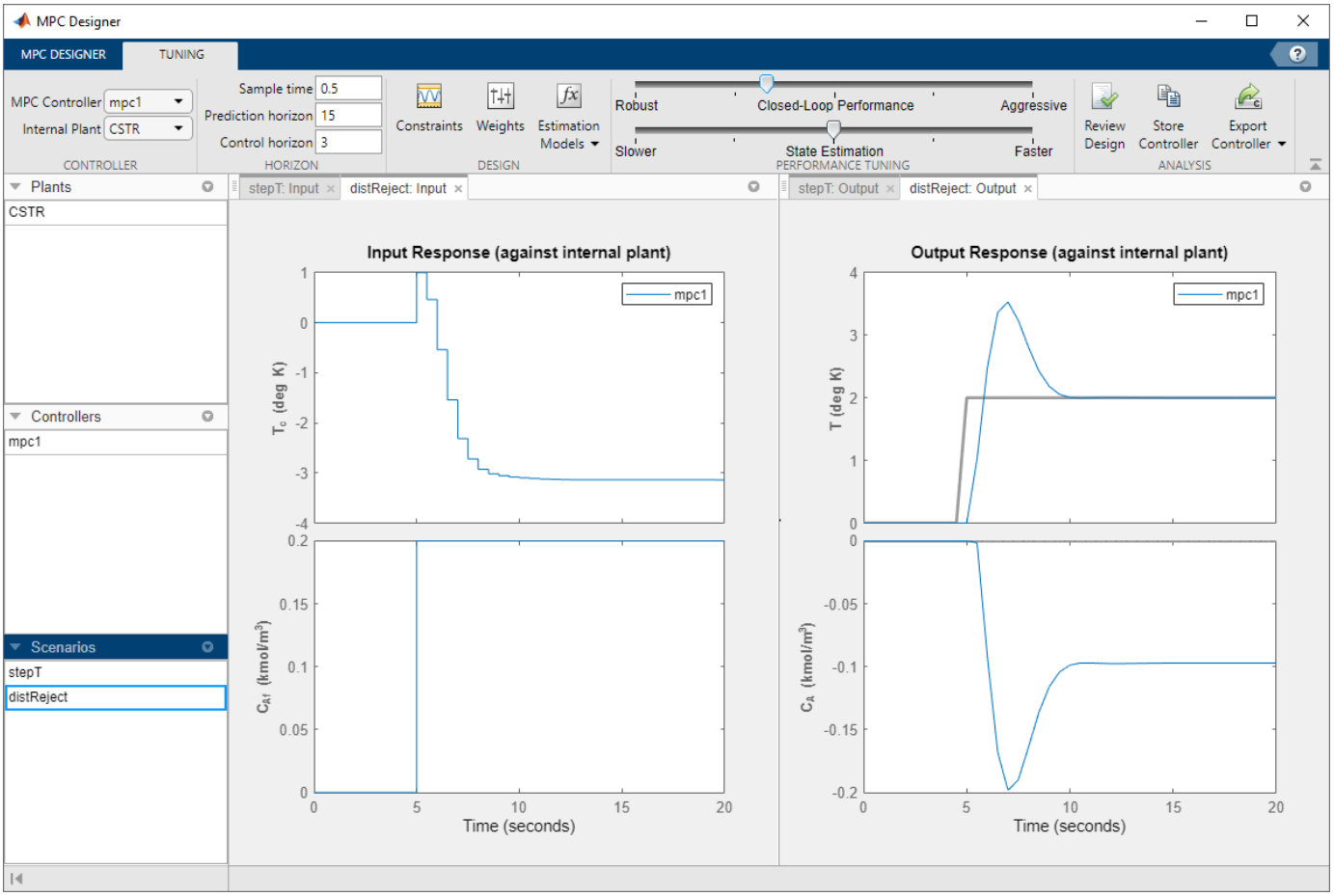
In the **Unmeasured Disturbances** row, in the **Signal** drop-down list, select `Step`. then specify a step **Size** of `0.2` and a **Time** of 5.



Click **OK**.

The app adds new scenario to the **Data Browser** and creates new corresponding **Input Response** and **Output Response** plots.

In the **Data Browser**, in the **Scenarios** section, rename `NewScenario` to `distReject`.



As you can see from the **Output Response** plots, the closed-loop system is still able to reach the desired reactor temperature. In this case, the required control actions, combined with the input disturbance, cause a steady-state decrease in the output concentration, $\mathbf{C_A}$ of 0.1 kmol/m^3.

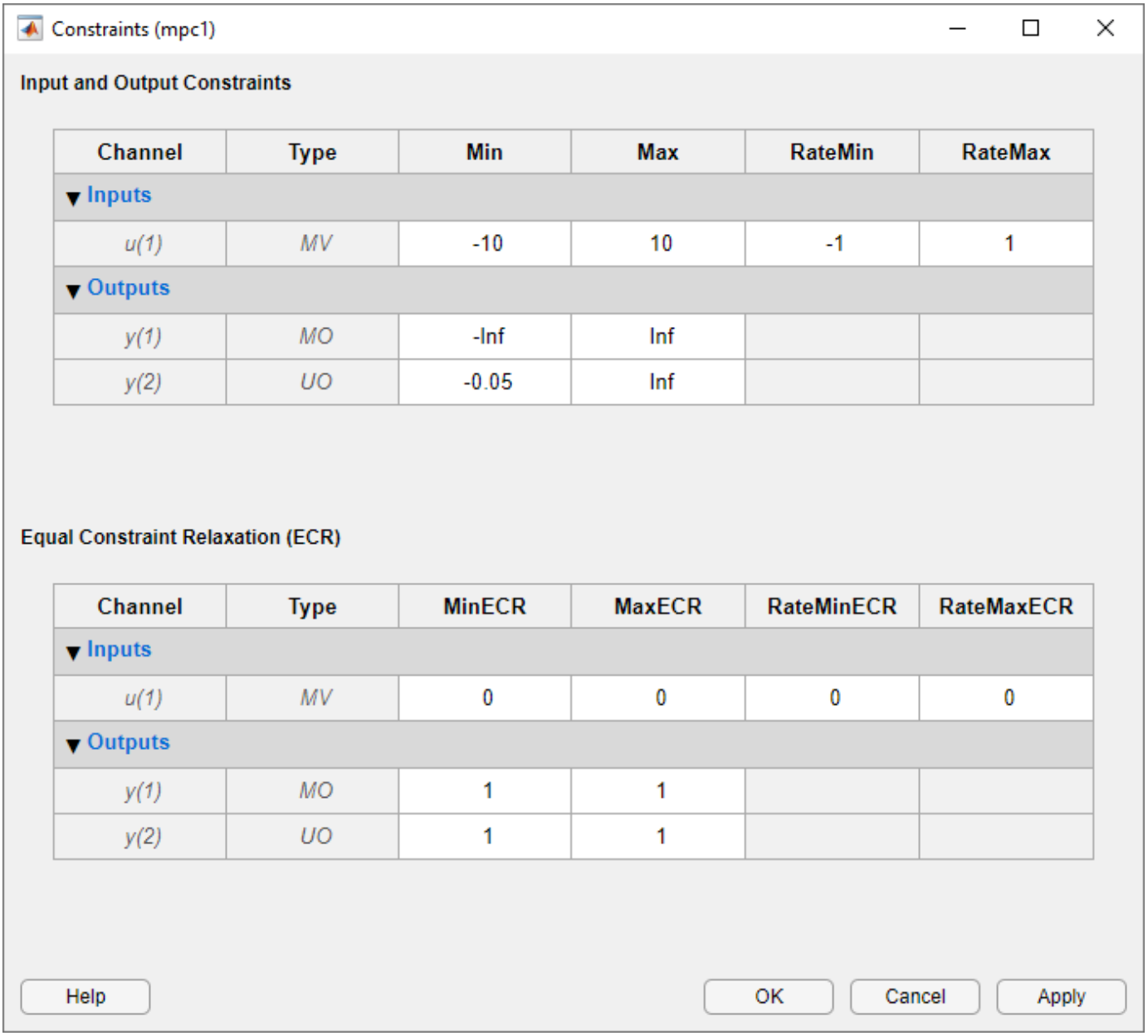
Specify Concentration Output Constraint

Previously, you defined the controller tuning weights to achieve the primary control objective of tracking the reactor temperature setpoint with zero steady-state error. Doing so enables the unmeasured reactor concentration to vary freely. Suppose that unwanted reactions occur once the reactor concentration drops below 0.05 kmol/m^3 with respect to its nominal value. To constrain the reactor concentration, specify an output constraint.

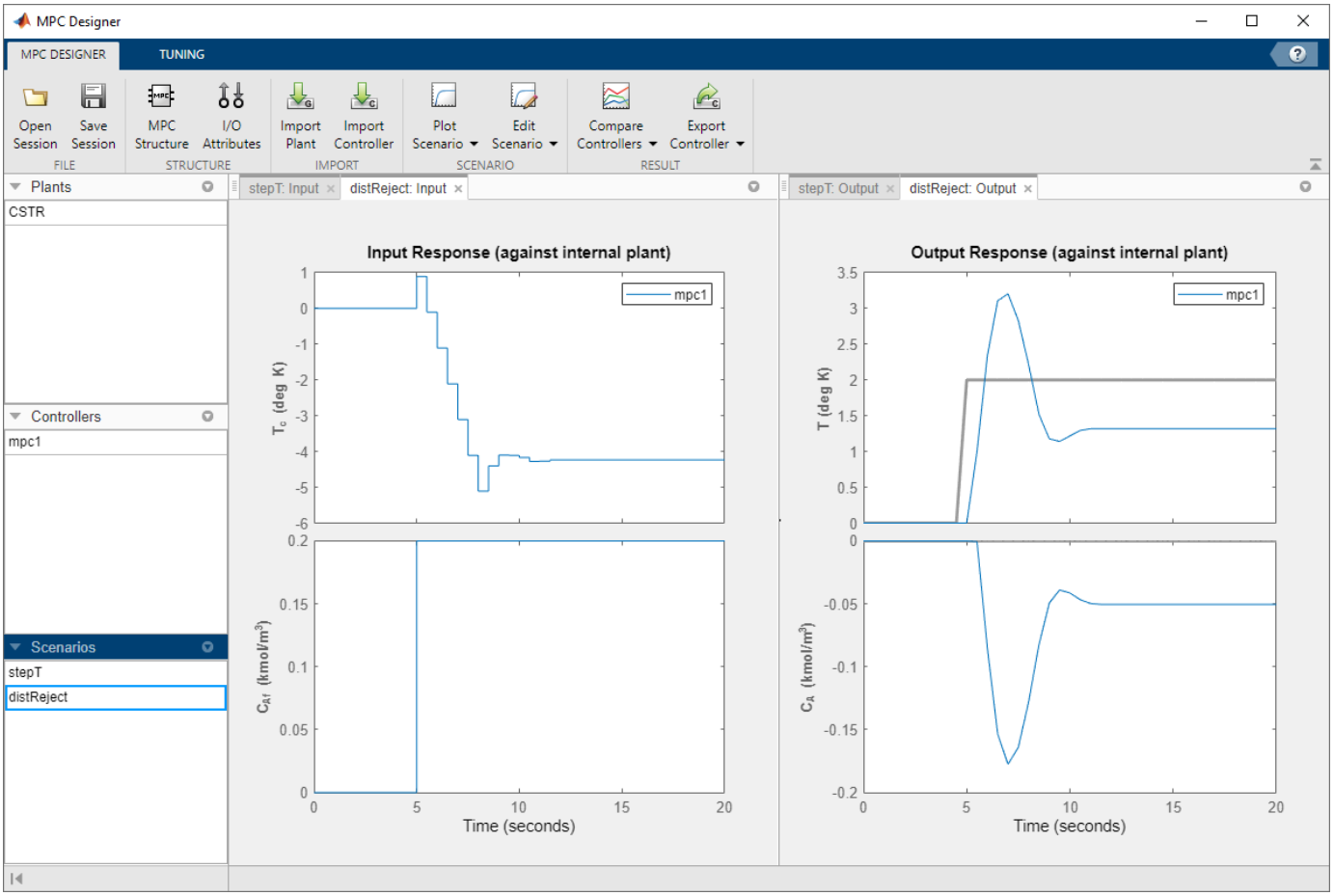
On the **Tuning** tab, in the **Design** section, click **Constraints**.

In the Constraints dialog box, in the **Input and Output Constraints** sections, in second row of the **Outputs** table, specify a **Min** unmeasured output (UO) value of `-0.05`.

By default, all output constraints are soft, meaning that their **MinECR** and **MaxECR** values are greater than zero. To soften the unmeasured output (UO) constraint further, increase its **MaxECR** value.


Constraints (mpc1)
Input and Output Constraints
Channel | Type | Min | Max | RateMin | RateMax
Inputs
u(1) | MV | -10 | 10 | -1 | 1
Outputs
y(1) | MO | -Inf | Inf
y(2) | UO | -0.05 | Inf
Equal Constraint Relaxation (ECR)
Channel | Type | MinECR | MaxECR | RateMinECR | RateMaxECR
Inputs
u(1) | MV | 0 | 0 | 0 | 0
Outputs
y(1) | MO | 1 | 1
y(2) | UO | 1 | 1
Help
OK
Cancel
Apply

Click **OK**.



In the **Output Response** plots, the reactor concentration, $\mathbf{C_A}$, stabilizes at -0.05 kmol/m^3 after 10 seconds. Since there is only one manipulated variable, the controller makes a compromise between the two competing control objectives: Temperature tracking and constraint satisfaction. A softer output constraint enables the controller to sacrifice the constraint requirement more to improve the temperature tracking.

Since the output constraint is soft, the controller maintains some level of temperature control by allowing small concentration constraint violations. In general, depending on your application requirements, you can experiment with different constraint settings to achieve an acceptable control objective compromise.

Export Controller

In the **Tuning** tab, in the **Analysis** section, click **Export Controller** to save the tuned controller, `mpc1`, to the MATLAB workspace.

Delete Plants, Controllers, and Scenarios

To delete a plant, controller, or scenario, in the **Data Browser**, right-click the item you want to delete, and select **Delete**.

You cannot delete the current controller. Also, you cannot delete a plant or scenario if it is the only listed plant or scenario.

If a plant is used by any controller or scenario, you cannot delete the plant.

To delete multiple plants, controllers, or scenarios, hold **Shift** and click each item that you want to delete.

References

[1] Seborg, D. E., T. F. Edgar, and D. A. Mellichamp, *Process Dynamics and Control*, 2nd Edition, Wiley, 2004, pp. 34–36 and 94–95.

See Also

Apps
MPC Designer

Objects
`mpc`

Related Examples

- “Design MPC Controller in Simulink” on page 3-32

More About

- “Tune Weights”
- “Specify Constraints”
- “MPC Signal Types” on page 2-2
- “MPC Prediction Models” on page 2-3
- “What is Model Predictive Control?” on page 1-3

Design MPC Controller at the Command Line

This example shows how to create and test a model predictive controller from the command line.

Define Plant Model

This example uses the plant model described in “Design Controller Using MPC Designer” on page 3-2. Create a state-space model of the plant and set some optional model properties such as names and units of input, state, and output variables.

```
% continuous-time state-space matrices, with temperature as first output
A = [   -5  -0.3427;
     47.68    2.785];
B = [    0   1
       0.3   0];
C = [0 1;
     1 0];
D = zeros(2,2);

% create state space plant model
CSTR = ss(A,B,C,D);

% set names
CSTR.InputName = {'T_c', 'C_A_f'};  % set names of input variables
CSTR.OutputName = {'T', 'C_A'};     % set names of output variables
CSTR.StateName = {'C_A', 'T'};      % set names of state variables

% set units
CSTR.InputUnit = {'deg K', 'kmol/m^3'};     % set units of input variables
CSTR.OutputUnit = {'deg K', 'kmol/m^3'};    % set units of output variables
CSTR.StateUnit = {'kmol/m^3', 'deg K'};     % set units of state variables
```

Note that this model is derived from the linearization of a nonlinear model around an operating point. Therefore, the values of the linear model input and output signals represent deviations with respect to their operating-point values in the nonlinear model. For more information, see “Linearization Using MATLAB Code” on page 2-22.

Assign Input and Output Signals to Different MPC Categories

The coolant temperature is the manipulated variable (MV), the inflow reagent concentration is an unmeasured disturbance input (UD), the reactor temperature is the measured output (MO), and the reagent concentration is an unmeasured output (UO).

```
CSTR=setmpcsignals(CSTR,'MV',1,'UD',2,'MO',1,'UO',2);
```

Display Basic Plant Properties and Plot Step Response

Use `damp` to display damping ratio, natural frequency, and time constant of the poles of the linear plant model.

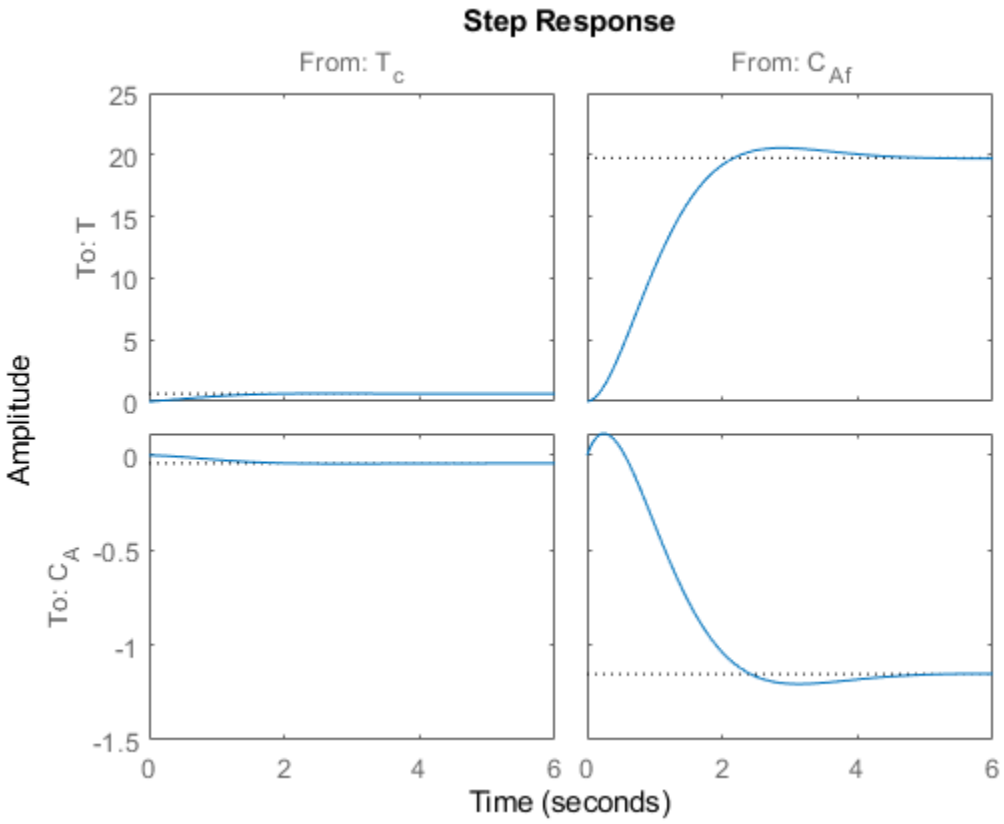
```
damp(CSTR)
```

```
          Pole              Damping       Frequency      Time Constant
                                       (rad/seconds)      (seconds)
```

```
 -1.11e+00 + 1.09e+00i     7.13e-01       1.55e+00         9.03e-01
 -1.11e+00 - 1.09e+00i     7.13e-01       1.55e+00         9.03e-01
```

Plot the open-loop step response.

```
step(CSTR)
```



Given the plant nominal stability, the time constant of about 1 second suggests a sample time not larger than of 0.5 seconds. With a sampling time of 0.5 seconds, a prediction horizon of 10 steps can cover the whole settling time of the open-loop plant, so you can use both parameters an initial guess. A shorter sample time implies less available time for the control computation. A longer horizon (more steps) implies a larger number of optimization variables, and therefore a more computationally demanding problem to be solved in the available time step.

Create Controller

To improve the clarity of the example, suppress Command Window messages from the MPC controller.

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

Create a model predictive controller with a control interval, or sample time, of `0.5` seconds, and with all other properties at their default values, including a prediction horizon of `10` steps and a control horizon of 2 steps.

```
mpcobj = mpc(CSTR,0.5) %#ok<*NOPTS>
```

```
MPC object (created on 03-Mar-2023 20:49:40):
---------------------------------------------
Sampling time:      0.5 (seconds)
Prediction Horizon: 10
Control Horizon:    2

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

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

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

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

Unconstrained

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

View and Modify Controller Properties

Display a list of the controller properties and their current values.

```
get(mpcobj)
```

```
                          Ts: 0.5
       PredictionHorizon (P): 10
          ControlHorizon (C): 2
                       Model: [1x1 struct]
   ManipulatedVariables (MV): [1x1 struct]
        OutputVariables (OV): [1x2 struct]
   DisturbanceVariables (DV): [1x1 struct]
                 Weights (W): [1x1 struct]
                   Optimizer: [1x1 struct]
                       Notes: {}
                    UserData: []
                     History: 03-Mar-2023 20:49:40
```

The displayed `History` value will be different for your controller, since it depends on when the controller was created. For a description of the editable properties of an MPC controller, enter `mpcprops` at the command line.

Use dot notation to modify these properties. For example, change the prediction horizon to `15`.

```
mpcobj.PredictionHorizon = 15;
```

Some property names have aliases. For example you can use the alias `MV` instead of `ManipulatedVariables`. Also, many of the controller properties are structures containing additional fields. Use the dot notation to view and modify these field values. For example, since by default, the controller has no constraints on manipulated variables and output variables, you can view and modify these constraints using dot notation.

Set constraints for the controller manipulated variable.

```
mpcobj.MV.Min = -10;    % K
mpcobj.MV.Max = 10;     % K
mpcobj.MV.RateMin = -1; % K/step
mpcobj.MV.RateMax = 1;  % K/step
```

You can abbreviate property names provided that the abbreviation is unambiguous. You can also view and modify the controller tuning weights. For example, modify the weights for the manipulated variable rate and the output variables.

```
mpcobj.W.ManipulatedVariablesRate = 0.3;
mpcobj.W.OutputVariables = [1 0];
```

You can also define time-varying constraints and weights over the prediction horizon, which shifts at each time step. For example, to force the manipulated variable to change more slowly towards the end of the prediction horizon, enter:

```
mpcobj.MV.RateMin = [-2; -1.5; -1; -1; -1; -0.5];

mpcobj.MV.RateMax = [2; 1.5; 1; 1; 1; 0.5];
```

The `-0.5` and `0.5` values are used for the fourth step and beyond.

Similarly, you can specify different output variable weights for each step of the prediction horizon. For example, enter:

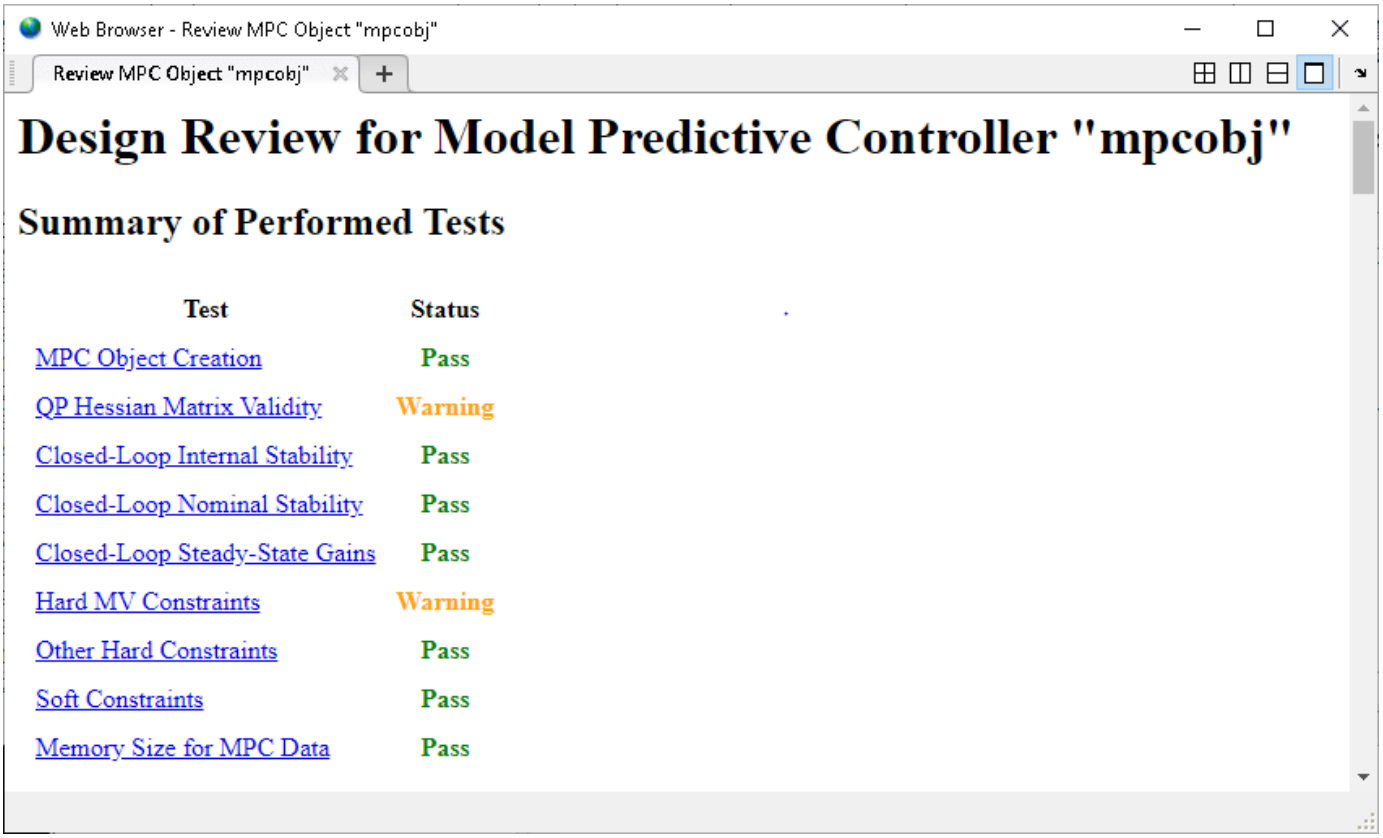
```
mpcobj.W.OutputVariables = [0.1 0; 0.2 0; 0.5 0; 1 0];
```

You can also modify the disturbance rejection characteristics of the controller. See `setEstimator`, `setindist`, and `setoutdist` for more information.

Review Controller Design

Generate a report on potential run-time stability and performance issues.

```
review(mpcobj)
```



In this example, the `review` command found two potential issues with the design. The first warning is caused by the fact that the weight on the `C_A` output error is zero. The second warning is caused by the fact that there are hard constraints on both `MV` and `MVRate`.

You can scroll down to see more information about each individual test result.

Steady-state closed loop output sensitivity gain

Compute the closed-loop, steady-state gain matrix for the closed loop system.

```
SoDC = cloffset(mpcobj)
```

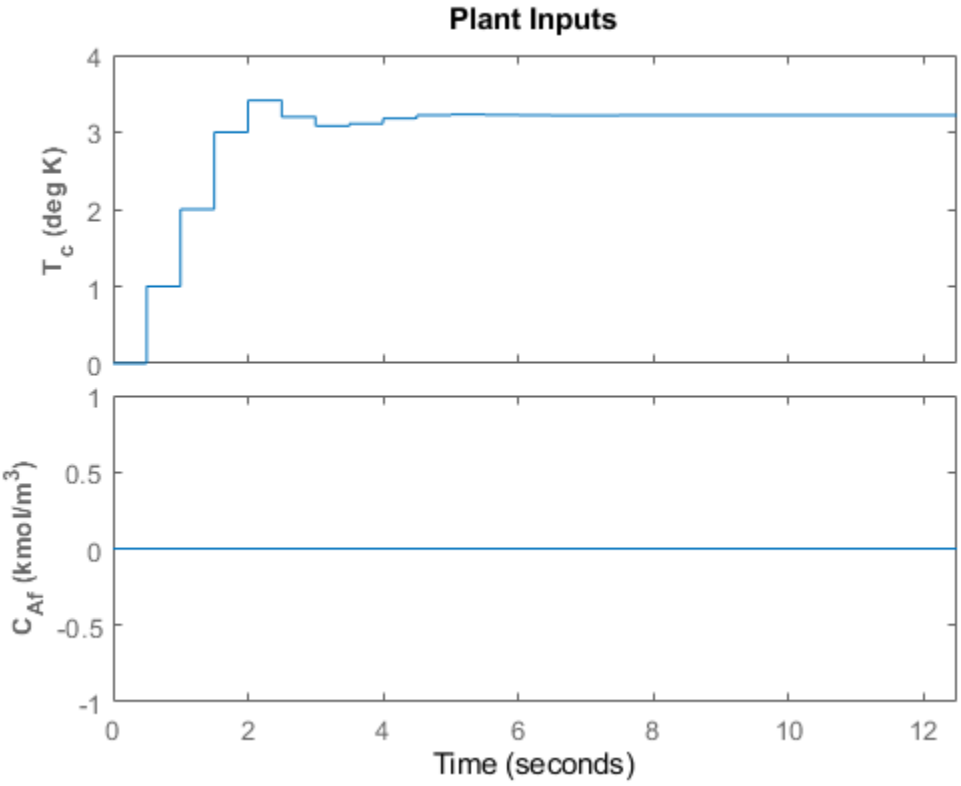
```
SoDC =

  -1.1990e-14
```

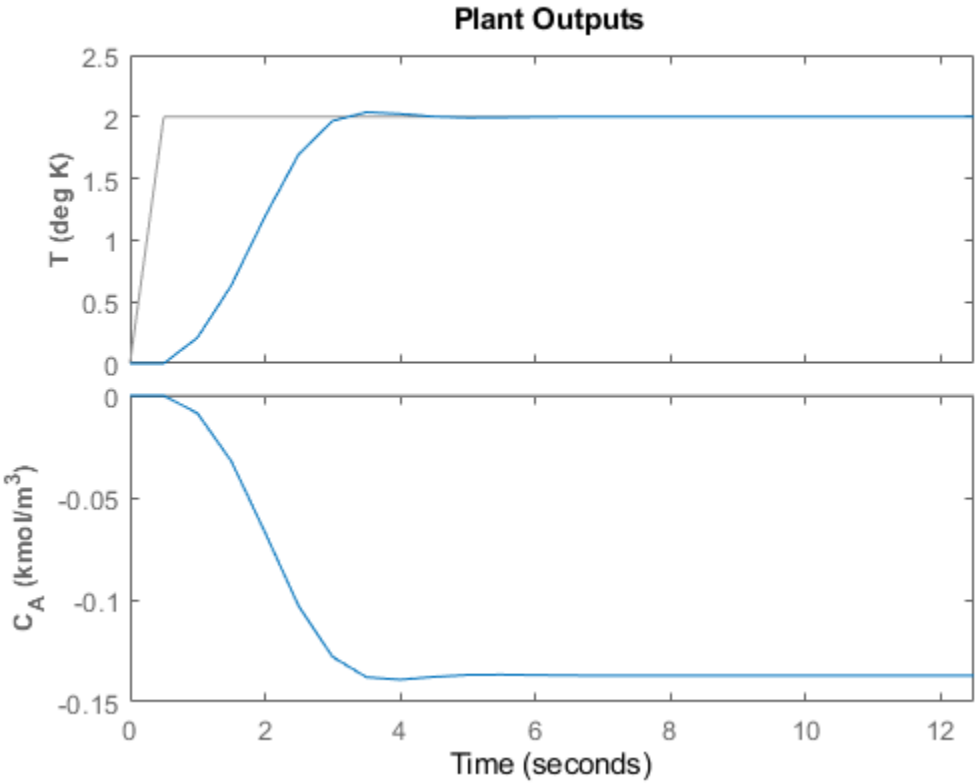
Perform Linear Simulations

Use the `sim` function to run a linear simulation of the system. For example, simulate the closed-loop response of `mpcobj` for `26` control intervals. Starting from the second step, specify setpoints of `2` and `0` for the reactor temperature (first output) and the reagent concentration (second output) respectively. Note that the setpoint for the concentration is ignored because the tuning weight for the second output is zero.

```
T = 26;
r = [0 0;
```

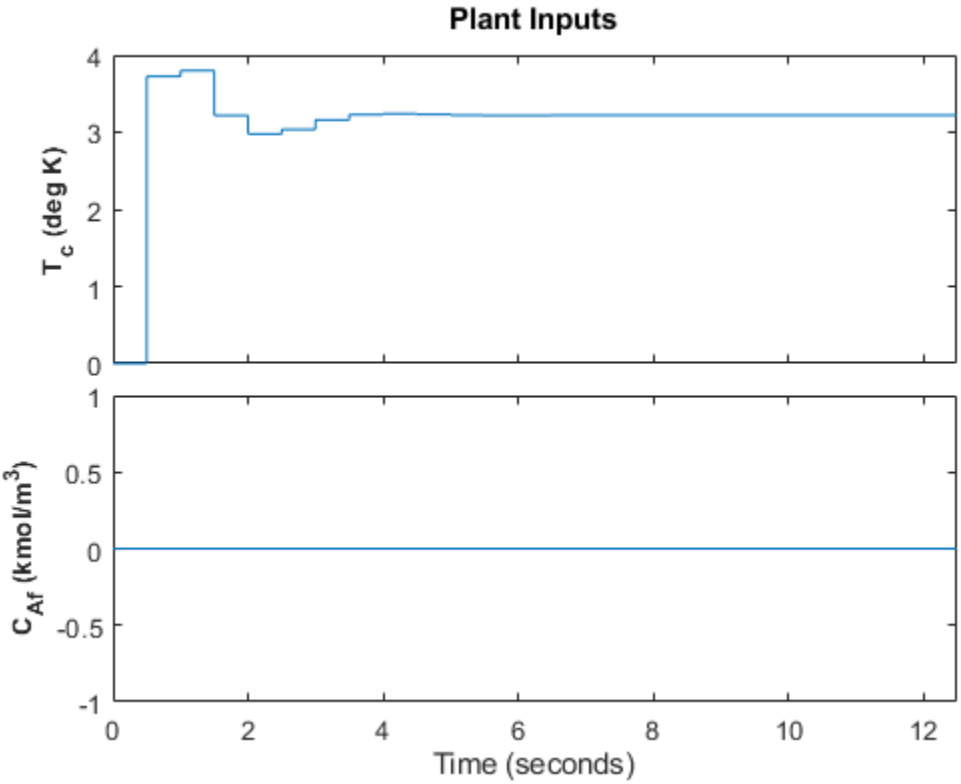
```
     2 0];
sim(mpcobj,T,r)
```

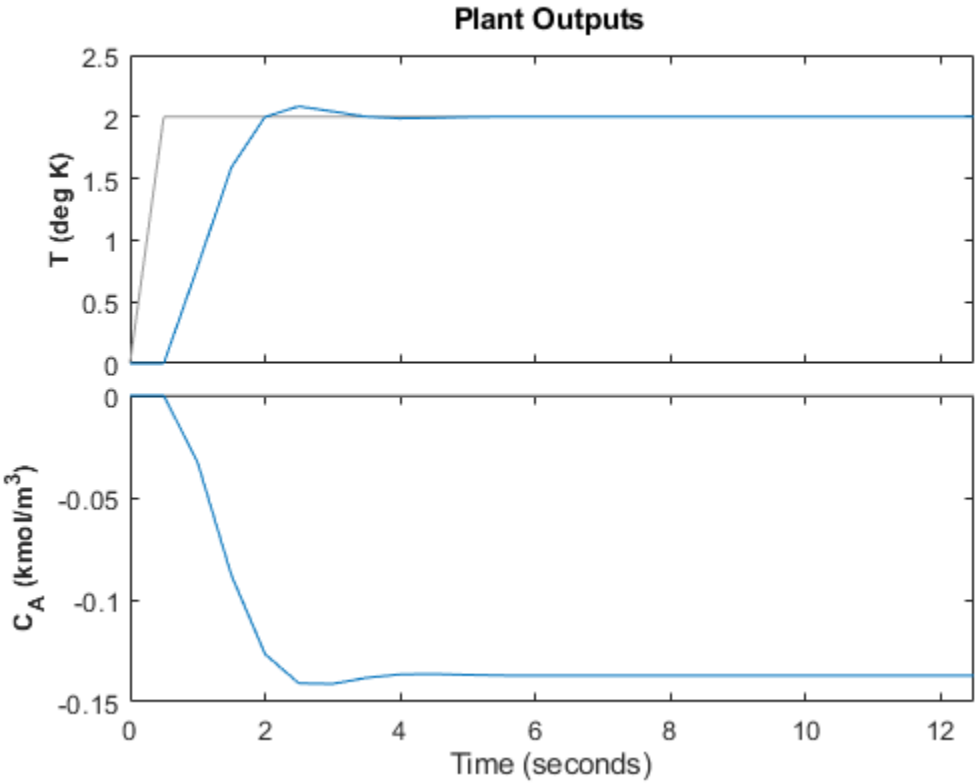




You can modify the simulation options using `mpcsimopt`. For example, run a simulation with the manipulated variable constraints turned off.

```
mpcopts = mpcsimopt;
mpcopts.Constraints = 'off';
sim(mpcobj,T,r,mpcopts)
```

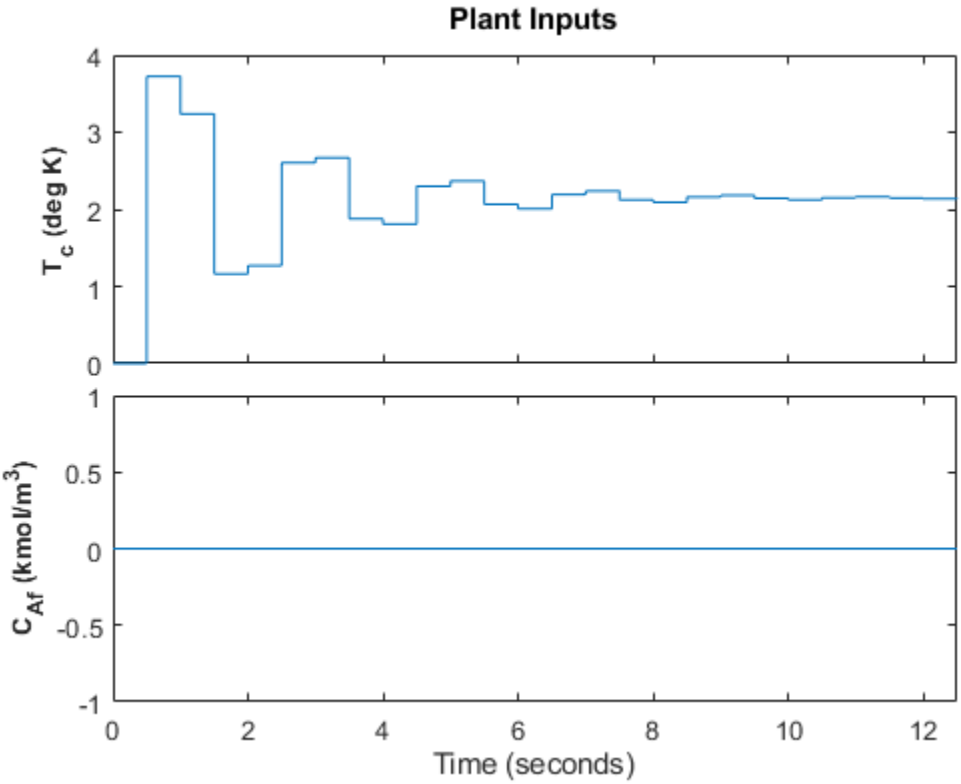

Plant Inputs
T_c (deg K)
C_{Af} (kmol/m^3)
Time (seconds)
4
3
2
1
0
1
0.5
0
-0.5
-1
0
2
4
6
8
10
12

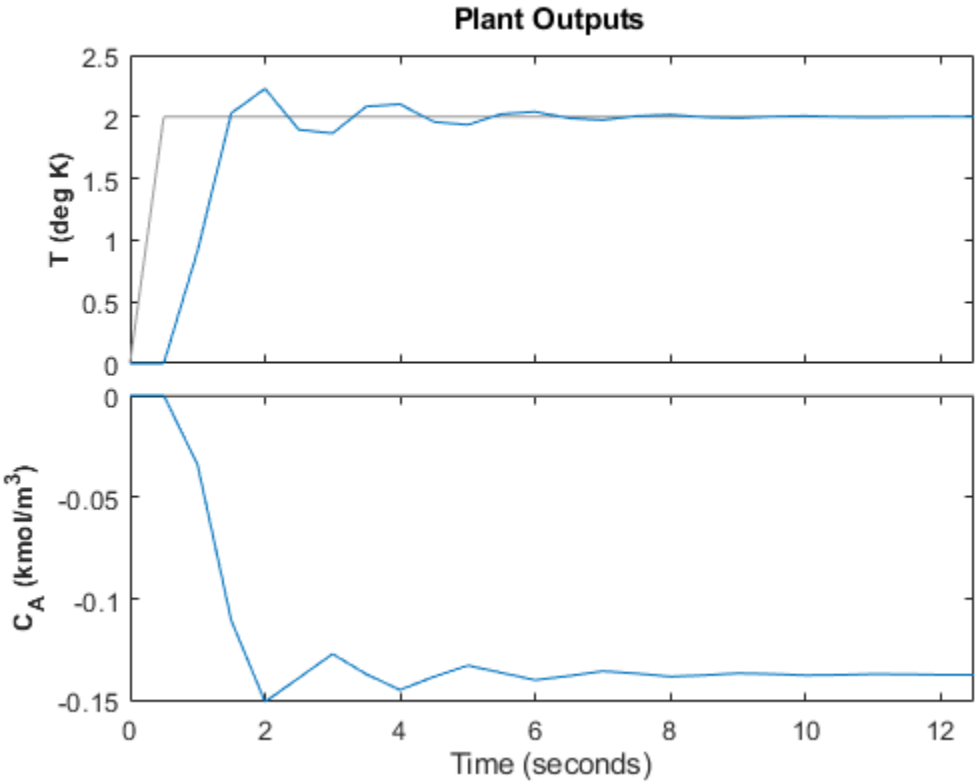


The first move of the manipulated variable now exceeds the specified 1-unit rate constraint.

You can also perform a simulation with a plant/model mismatch. For example, define a plant with 50% larger gains than those in the model used by the controller, and a time delay of 0.1 seconds.

```
mpcopts.Model = tf(1.5,1,'InputDelay',0.1)*CSTR;
sim(mpcobj,T,r,mpcopts)
```


Plant Inputs
T_c (deg K)
C_{Af} (kmol/m^3)
Time (seconds)
4
3
2
1
0
1
0.5
0
-0.5
-1
0
2
4
6
8
10
12



The plant/model mismatch degrades the controller performance, as you can tell from the oscillatory behavior of the closed loop responses. Degradation can be severe and must be tested on a case-by-case basis.

Other simulation options include the addition of a specified noise sequence to the manipulated variables or measured outputs, open-loop simulations, and a look-ahead option for better setpoint tracking or measured disturbance rejection.

Store and Plot Simulation Results

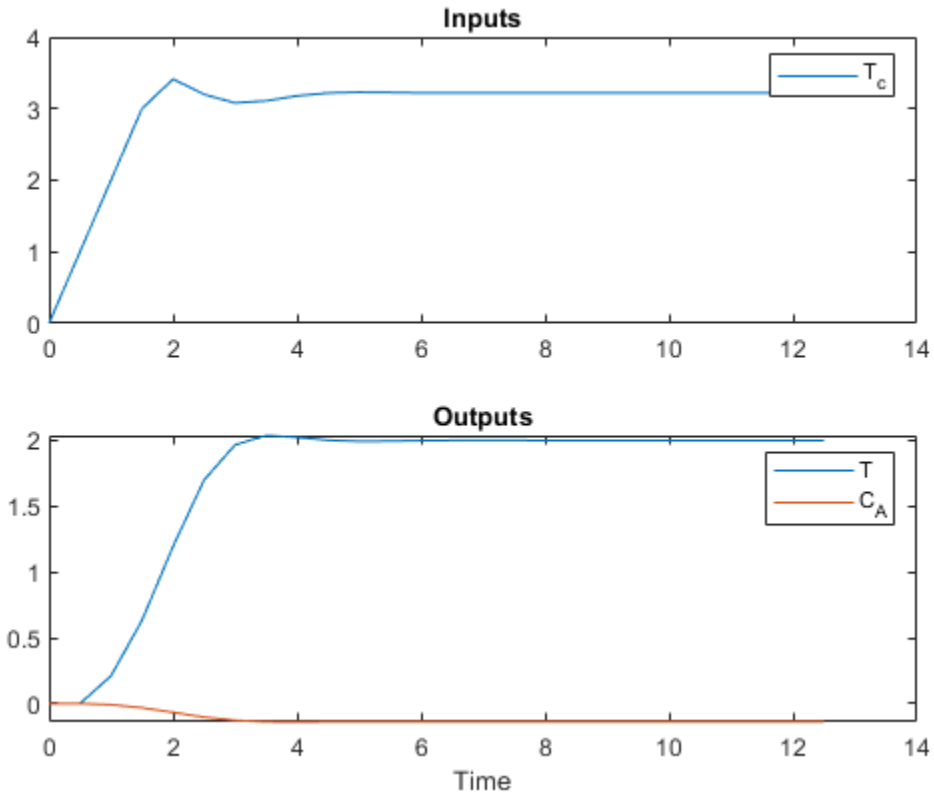
Simulate the system storing the simulation results in the MATLAB® Workspace.

```
[y,t,u] = sim(mpcobj,T,r);
```

This syntax suppresses automatic plotting and returns the simulation results in the `y`, `t` and `u` variables. You can use the results for other purposes, including custom plotting. For example, plot the manipulated variable and both output variables in the same figure.

```
figure

subplot(2,1,1)
plot(t,u)
title('Inputs')
legend('T_c')

subplot(2,1,2)
plot(t,y)
```

```
title('Outputs')
legend('T','C_A')
xlabel('Time')
```



Restore the `mpcverbosity` setting.

```
mpcverbosity(old_status);
```

See Also

Apps
MPC Designer

Functions
`review` | `sim`

Objects
`mpc`

Blocks
MPC Controller

Related Examples

- “Design Controller Using MPC Designer” on page 3-2

- “Design MPC Controller in Simulink” on page 3-32

More About

- “Tune Weights”
- “MPC Signal Types” on page 2-2
- “MPC Prediction Models” on page 2-3
- “What is Model Predictive Control?” on page 1-3

Design MPC Controller in Simulink

This example shows how to design a model predictive controller for a continuous stirred-tank reactor (CSTR) in Simulink using **MPC Designer**.

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

If you do not have Simulink Control Design software, you must first create an `mpc` object in the MATLAB workspace. For more information, see “Design Controller Using MPC Designer” on page 3-2 and “Design MPC Controller at the Command Line” on page 3-19.

CSTR Model

The nonlinear model of a Continuously Stirred Tank Reactor (CSTR) is described in “CSTR Model” on page 2-8. In the model, the inputs are arranged in the vector $u(t)$ and are as follows.

- u_1 — C_{Af}, the concentration of reagent A in the inlet feed stream, measured in kmol/m^3
- u_2 — T_f, the temperature of the inlet feed stream, measured in K
- u_3 — T_c, the temperature of the jacket coolant, measured in K

while the state variables are arranged in the vector $x(t)$.

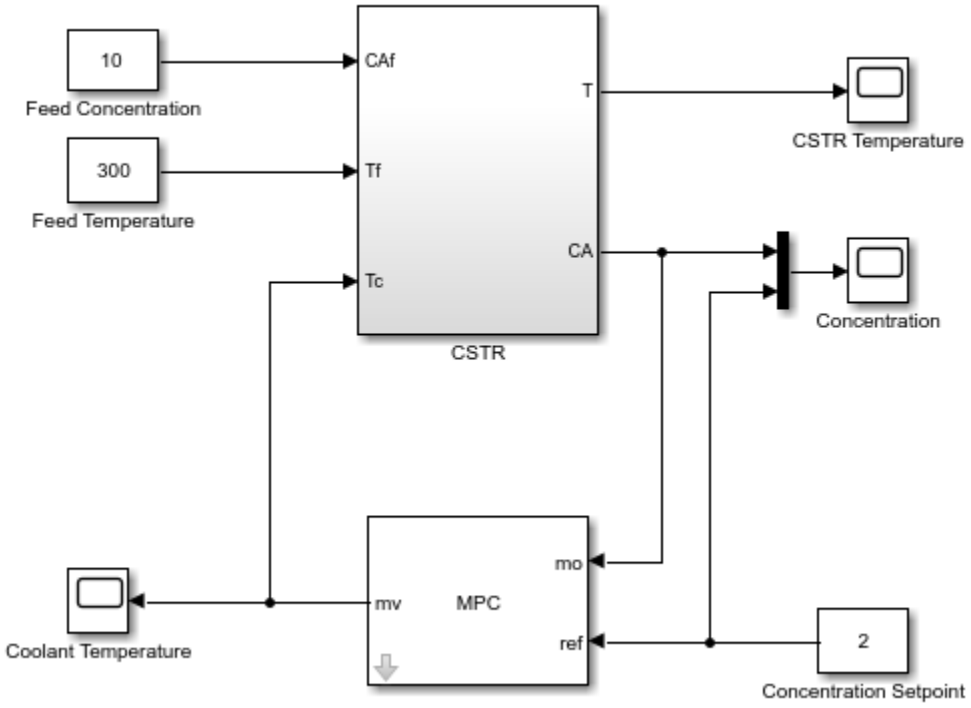
- x_1 — C_A, the concentration of reagent A in the reactor, measured in kmol/m^3
- x_2 — T, the temperature in the reactor, measured in K

The control objective is to maintain the residual concentration, C_A, at its nominal setpoint by adjusting the coolant temperature, T_c. Changes in the feed concentration, C_{Af}, and feed temperature, T_f, cause disturbances in the CSTR reaction.

The reactor temperature, T, is usually measured. However, for this example, ignore the reactor temperature, and assume that the residual concentration is measured directly.

Open the Simulink model.

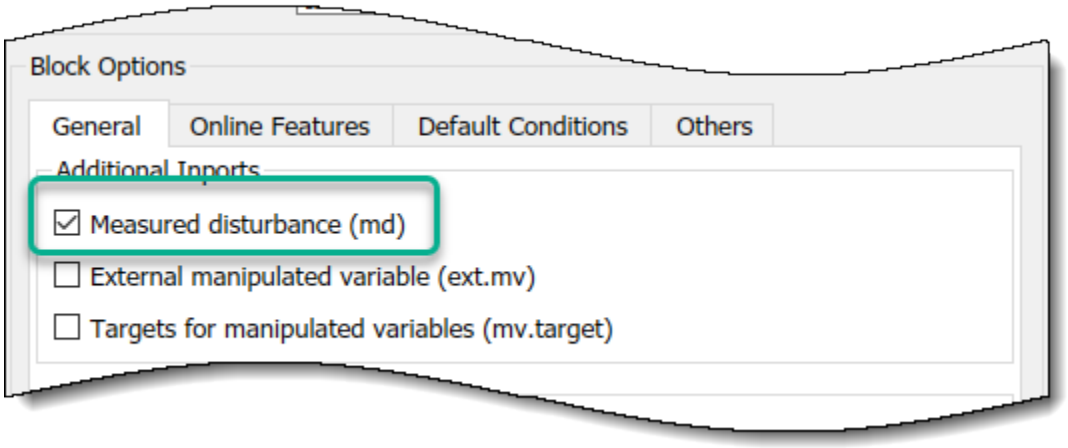
```
open_system('CSTR_ClosedLoop')
```



Connect Measured Disturbance Signal

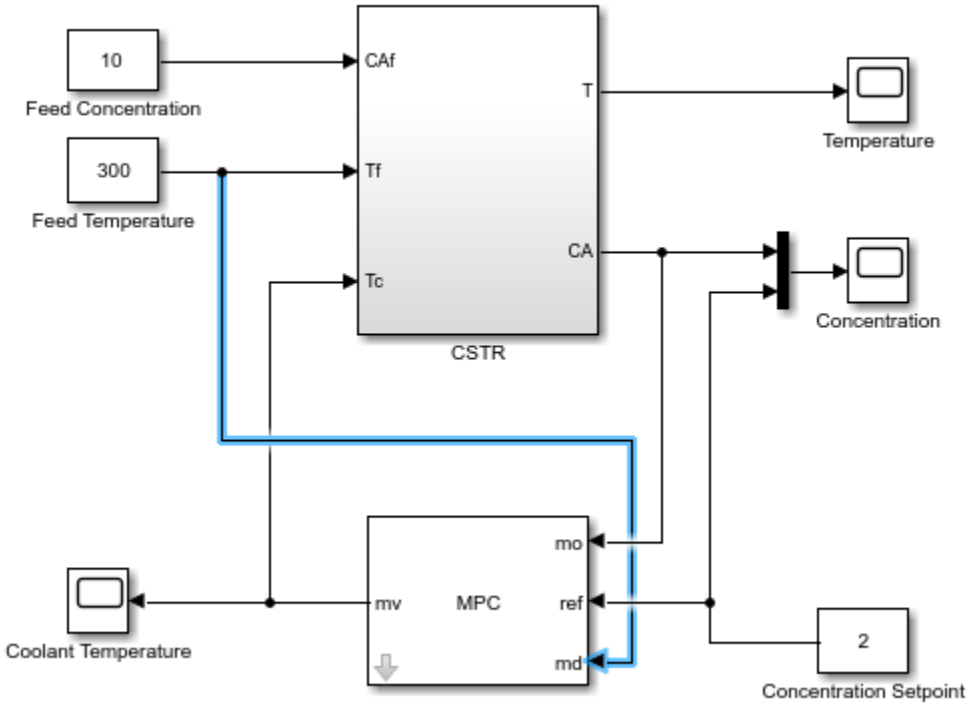
In the Simulink model window, double-click the MPC Controller block.

In the Block Parameters dialog box, on the **General** tab, select the **Measured disturbance (md)** check box.



Click **Apply** to add the `md` input port to the controller block.

In the Simulink model window, connect the Feed Temperature block output to the `md` input port.



Linearize Simulink Model

In this example, you linearize the Simulink model from within **MPC Designer**, which requires Simulink Control Design software. For more information, see “Linearize Simulink Models Using MPC Designer” on page 2-31.

If you do not have Simulink Control Design software, you must first create an `mpc` object in the MATLAB workspace and specify that controller object in the MPC Controller block.

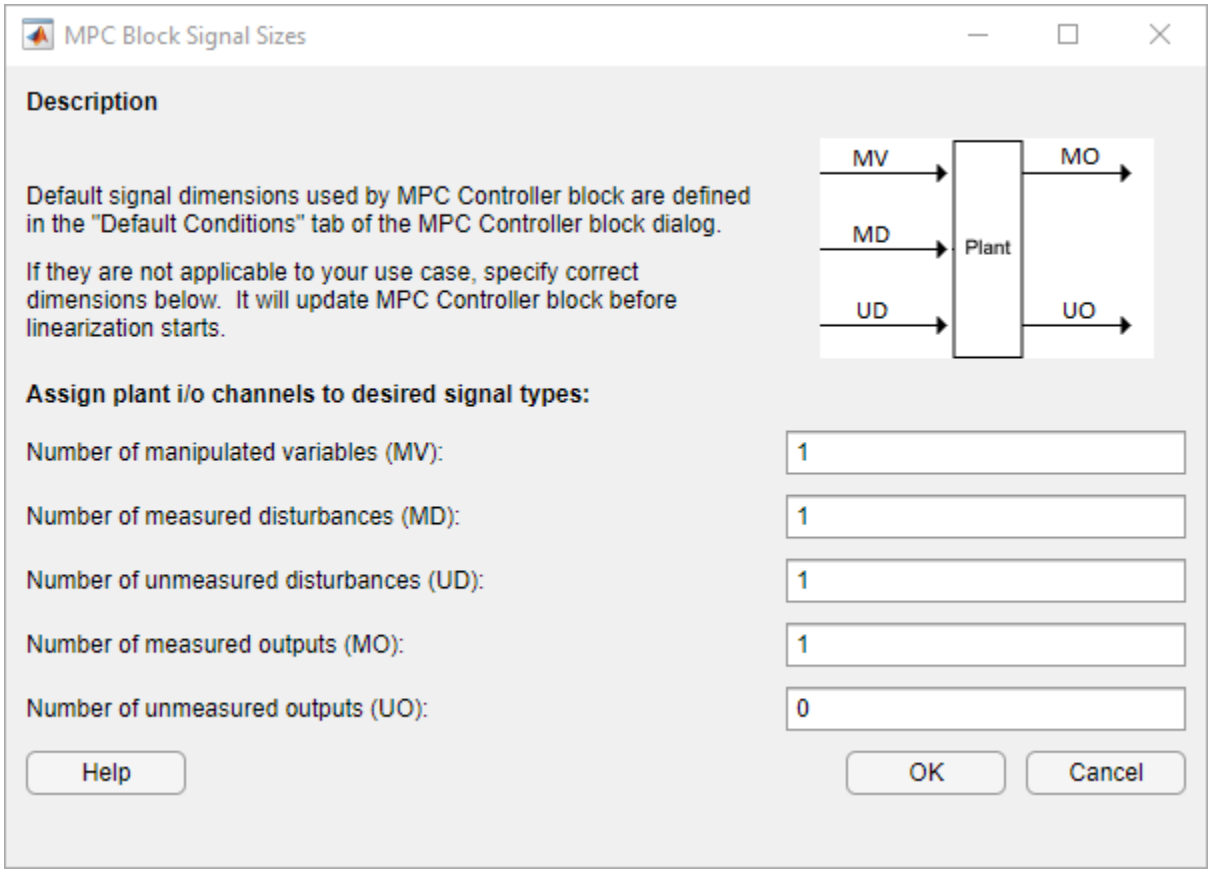
To open **MPC Designer**, open the MPC Controller block and click **Design**.

In **MPC Designer**, on the **MPC Designer** tab, in the **Structure** section, click **MPC Structure**.

In the Define MPC Structure By Linearization dialog box, in the **Controller Sample Time** section, specify a sample time of `0.1`.

In the **MPC Structure** section, click **Change I/O Sizes** to add the unmeasured disturbance and measured disturbance signal dimensions.

In the MPC Block Signal Sizes dialog box, specify the number of input/output channels of each type.

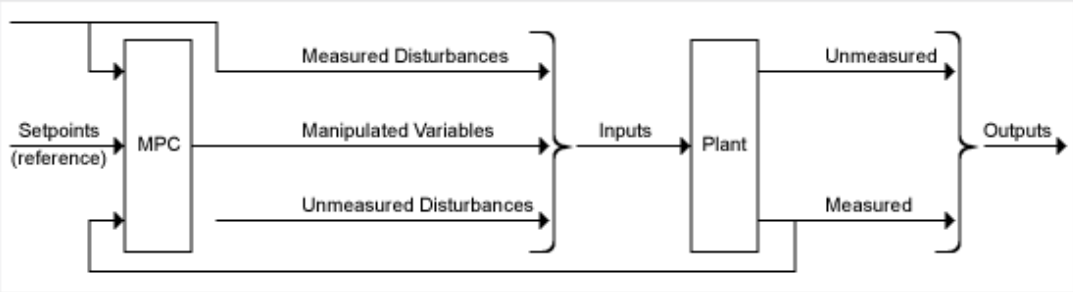


Click **OK**.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Signals for Plant Inputs** section, the app adds a row for **Unmeasured Disturbances (UD)**.

Define MPC Structure By Linearization

MPC Structure



Number of MVs: 1
Number of MDs: 1
Number of UDs: 1
Number of MOs: 1
Number of UOs: 0

Change I/O Sizes

Controller Sample Time

Specify MPC controller sample time (default sample time in the MPC block): 0.1

Simulink Operating Point

Select: Model Initial Condition | Edit | Create

Simulink Signals for Plant Inputs

	Selected	Type	Block Path
1	☐	Manipulated Variables (MV)	CSTR_ClosedLoop/MPC Controller:1
2	☐	Measured Disturbances (MD)	CSTR_ClosedLoop/Feed Temperature:1
3	☐	Unmeasured Disturbances (UD)	Select a UD signal in the Simulink model

Select Signals

Simulink Signals for Plant Outputs

	Selected	Type	Block Path
1	☐	Measured Outputs (MO)	CSTR_ClosedLoop/CSTR:2

Select Signals

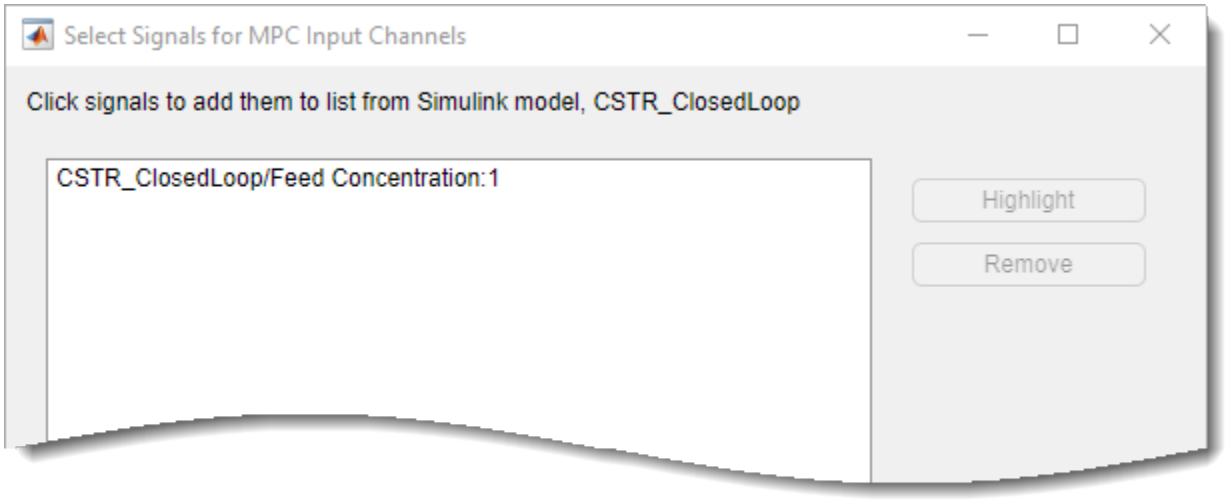
Help | Import | Cancel

The manipulated variable, measured disturbance, and measured output are already assigned to their respective Simulink signal lines, which are connected to the MPC Controller block.

In the **Simulink Signals for Plant Inputs** section, select the **Unmeasured Disturbances (UD)** row, and click **Select Signals**.

In the Simulink model window, click the output signal from the Feed Concentration block.

The signal is highlighted and its block path is added to the Select Signal for MPC Input Channels dialog box.

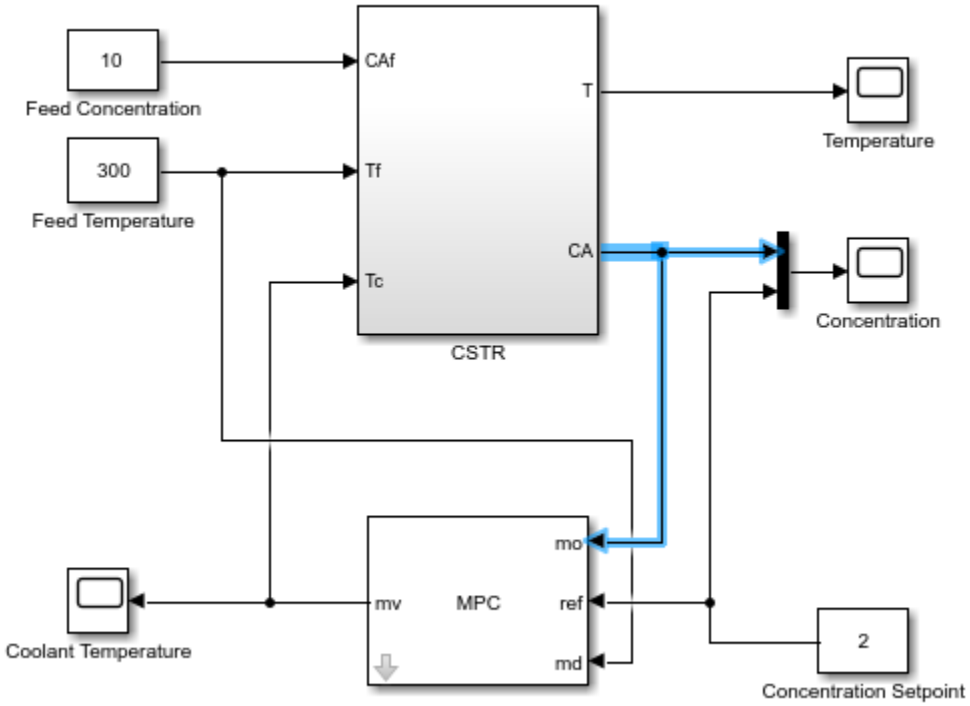


In the Select Signals for MPC Input Channels dialog box, click **OK**.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Signals for Plant Inputs** table, the **Block Path** for the unmeasured disturbance signal is updated.

In this example, you linearize the Simulink model at a steady-state equilibrium operating point where the residual concentration is 2 kmol/m^3. To compute such an operating point, add the `CA` signal as a trim output constraint, and specify its target constraint value.

In the Simulink model window, select the signal line connected to `CA` output port of the CSTR block.

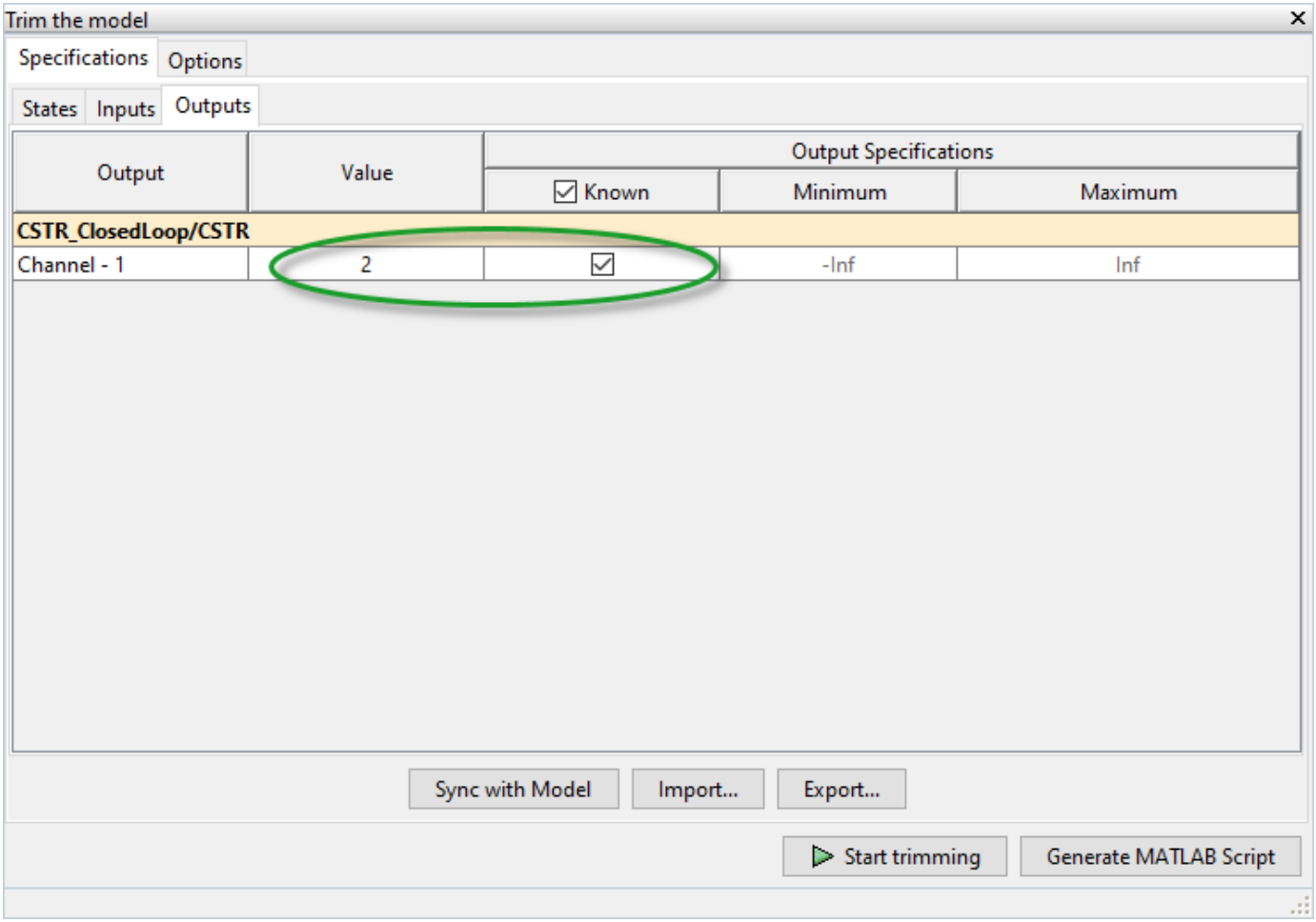


On the **Apps** tab, click **Linearization Manager**. Then, on the **Linearization** tab, in the **Insert Analysis Points** gallery, under the **Trim** section, select **Trim Output Constraint**.

The `CA` signal can now be used to define output specifications for calculating a model steady-state operating point.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Operating Point** section, in the **Create** drop-down list, select **Trim Model**.

In the Trim the model dialog box, on the **Outputs** tab, check the box in the **Known** column for `Channel - 1` and specify a **Value** of `2`.



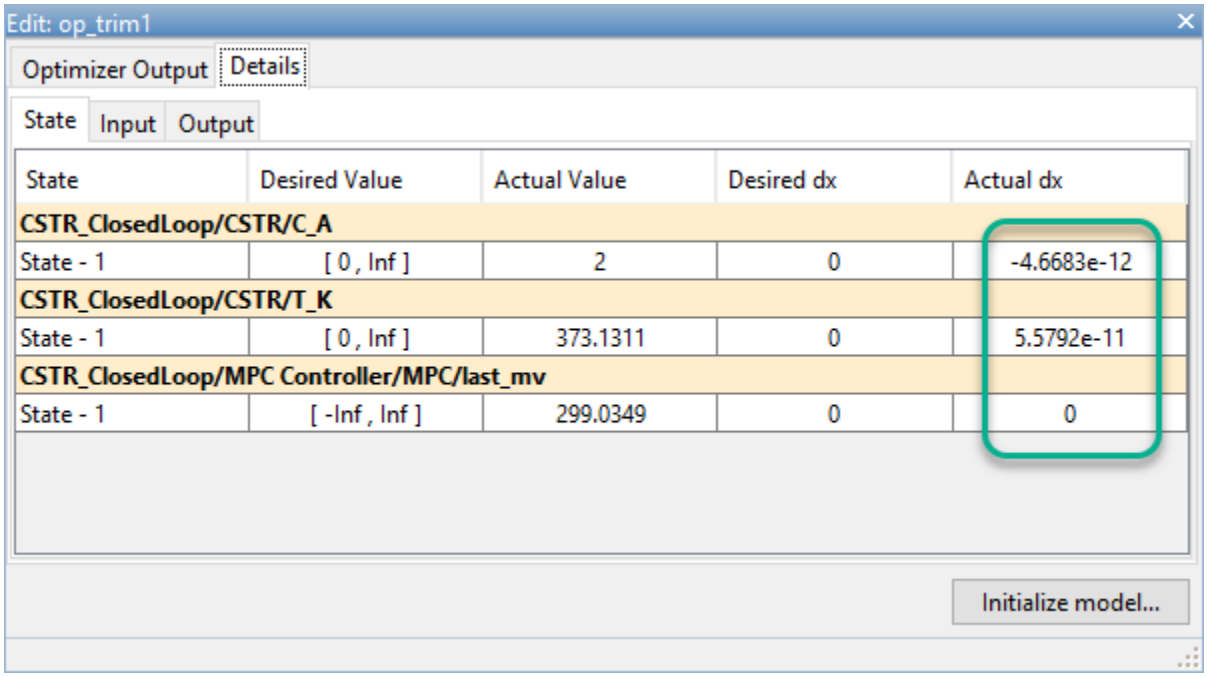
This setting constrains the value of the output signal during the operating point search to a known value.

Click **Start Trimming**.

The Trim progress viewer window opens up showing the optimization progress towards finding a point in the state-input space of the model with the characteristics specified in the **States**, **Inputs**, and **Outputs** tabs. After the optimization process terminates, close the trim progress window as well as the Trim the model dialog box.

In the Define MPC Structure By Linearization dialog box, in the **Simulink Operating Point** section, the computed operating point, `op_trim1 (MPC_OP_Workspace)`, is added to the drop-down list and selected.

In the **Simulink Operating Point** section, click **Edit**.



In the Edit dialog box, on the **State** tab, in the **Actual dx** column, the near-zero derivative values indicate that the computed operating point is at steady-state.

To set the initial states of the Simulink model to the operating point values in the **Actual Values** column, click **Initialize model**. Doing so enables you to later simulate the Simulink model at the computed operating point rather than at the default model initial conditions.

In the Initialize Model dialog box, click **OK**.

When setting the model initial conditions, **MPC Designer** exports the operating point to the MATLAB workspace. Also, in the Simulink Configuration Parameters dialog box, in the **Data Import/Export** section, it selects the **Input** and **Initial state** parameters and configures them to use the states and inputs in the exported operating point.

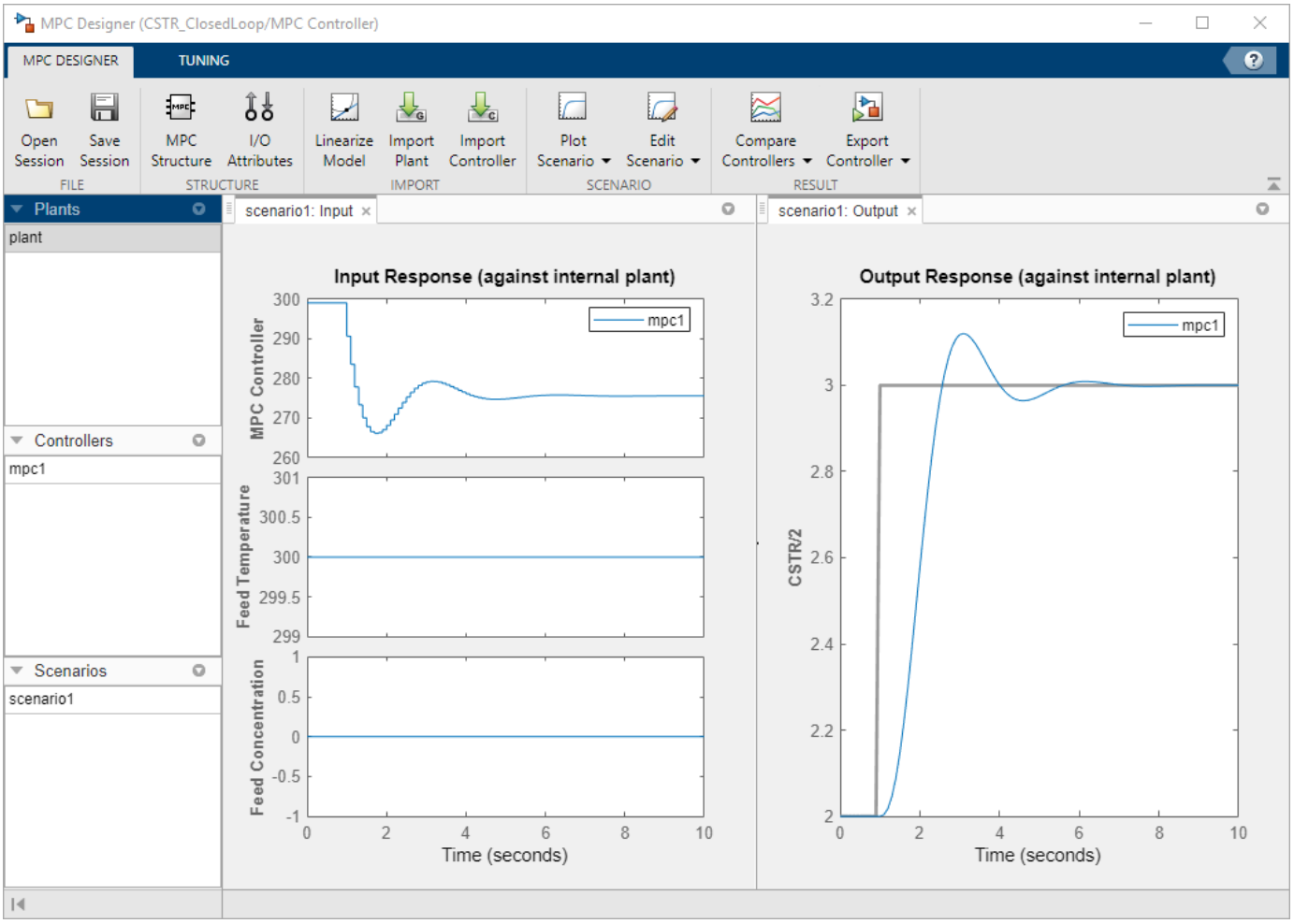
To reset the model initial conditions, for example if you delete the exported operating point, clear the **Input** and **Initial state** parameters.

Close the Edit dialog box.

In the Define MPC Structure By Linearization dialog box, linearize the Simulink model and import the linearized model in **MPC Designer** by clicking **Import**.

In the data browser section on the left hand side, the app adds the following items.

- Linearized and discretized plant model `plant`
- Default MPC controller `mpc1` created using the linearized plant as an internal prediction model
- Default simulation scenario `scenario1`

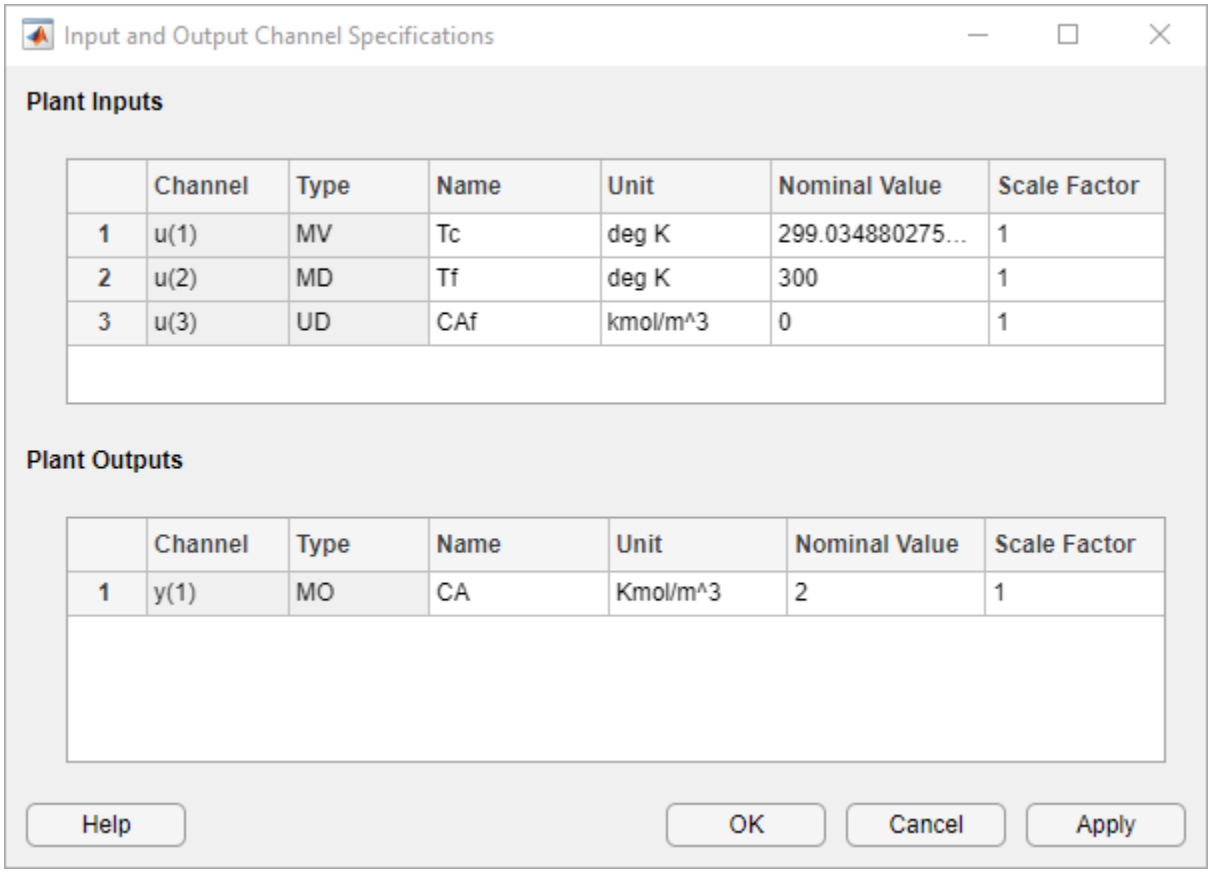


Define Input/Output Channel Attributes

On the **MPC Designer** tab, in the **Structure** section, click **I/O Attributes**.

In the Input and Output Channel Specifications dialog box, in the **Name** column, specify meaningful names for each input and output channel.

In the **Unit** column, specify appropriate units for each signal.



The **Nominal Value** for each signal is the corresponding steady-state value at the computed operating point.

Click **OK**.

Define Disturbance Rejection Simulation Scenarios

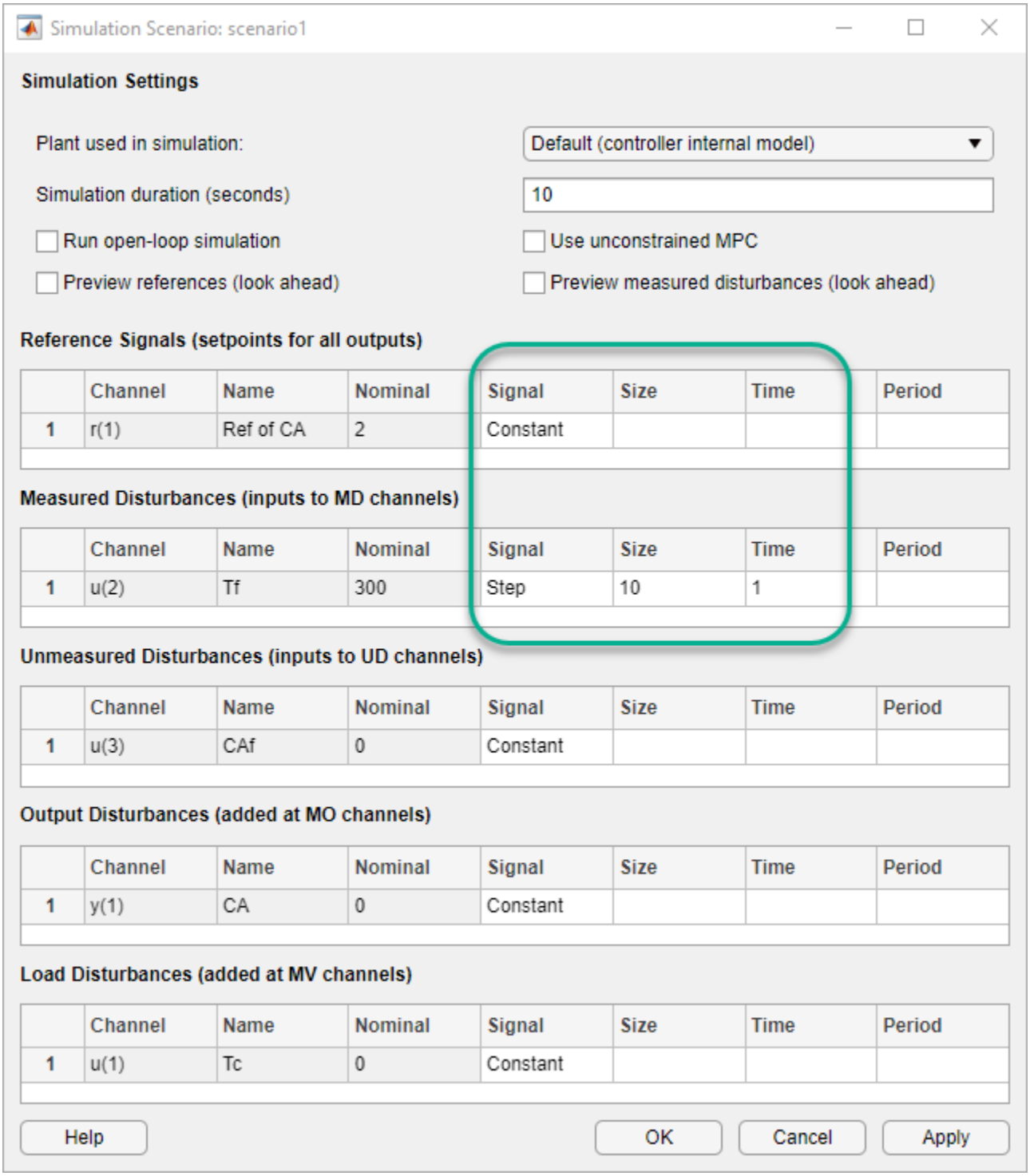
The primary objective of the controller is to hold the residual concentration C_A at the nominal value of 2 kmol/m^3. To do so, the controller must reject both measured and unmeasured disturbances.

On the **MPC Designer** tab, in the **Scenario** section, select **Edit Scenario > scenario1**.

In the Simulation Scenario dialog box, in the **Reference Signals (setpoints for all outputs)** table, in the **Signal** drop-down list select **Constant** to hold the output setpoint at its nominal value.

In the **Measured Disturbances** table, in the **Signal** drop-down list, select **Step**.

Specify a step **Size** of `10` and a step **Time** of `0`.



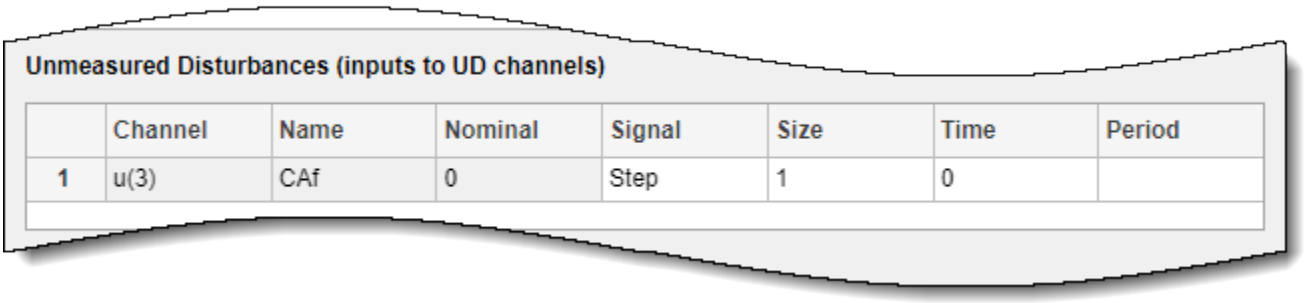
Click **OK**.

In the **Data Browser**, under **Scenarios**, click `scenario1`. Click `scenario1` a second time, and rename it `MD_reject`.

In the **Scenario** section, click **Plot Scenario** > **New Scenario**.

In the Simulation Scenario dialog box, in the **Unmeasured Disturbances** table, in the **Signal** drop-down list, select **Step**.

Specify a step **Size** of `1` and a step **Time** of `0`.



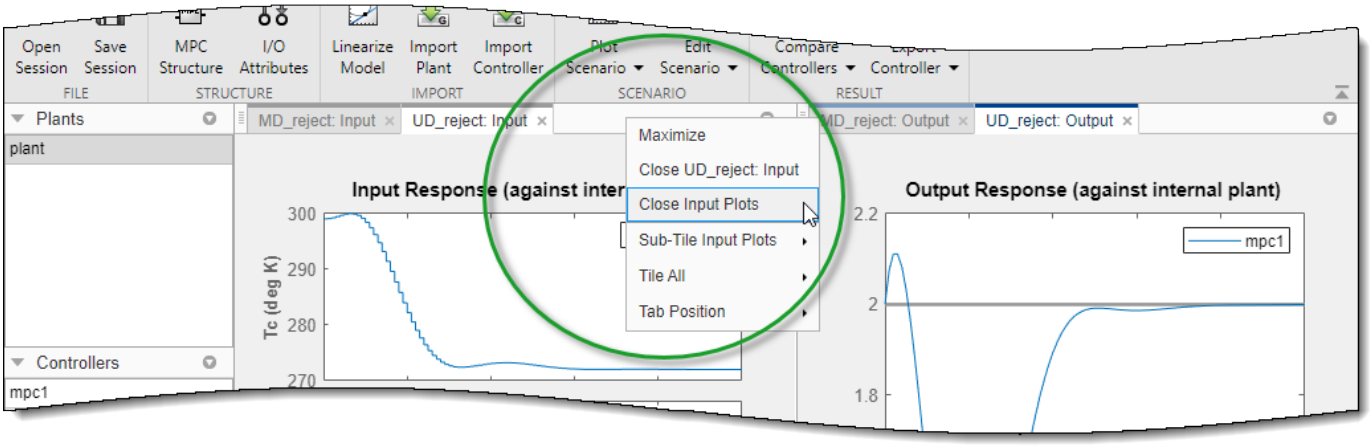
Click **OK**.

In the **Data Browser**, under **Scenarios**, rename `NewScenario` to `UD_reject`.

Arrange Output Response Plots

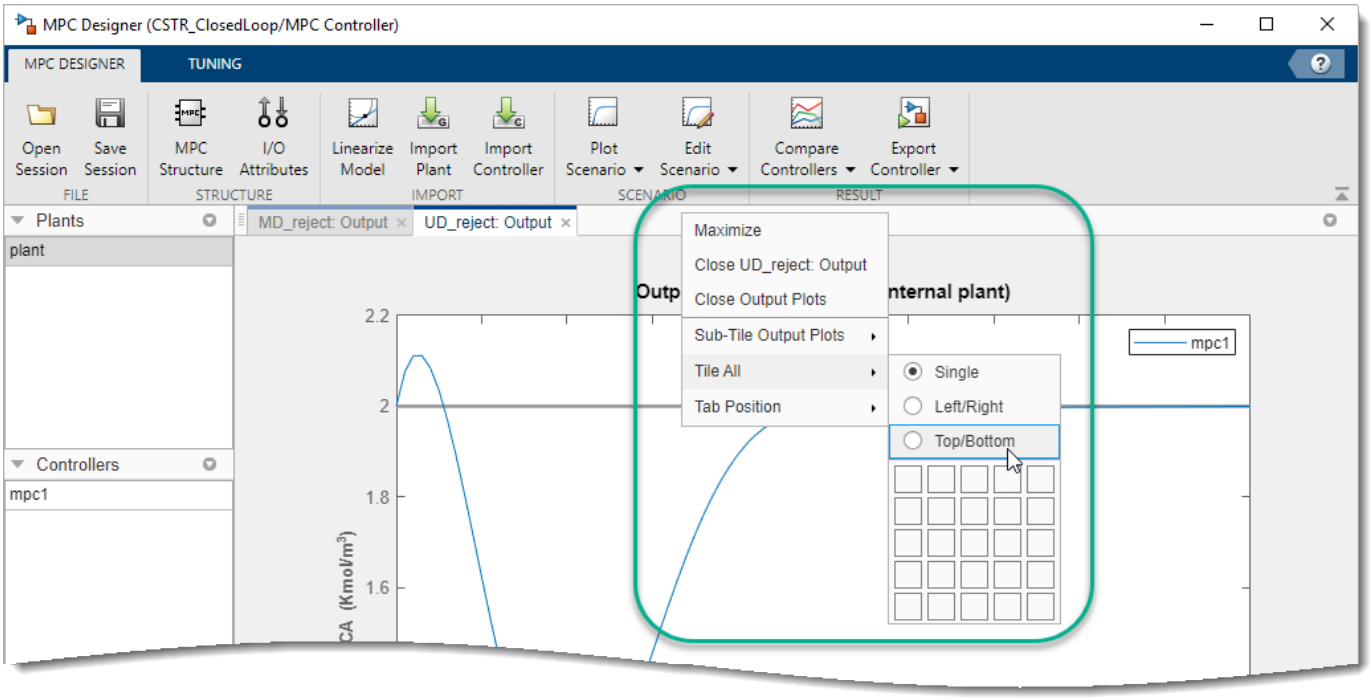
To make viewing the tuning results easier, arrange the plot area to display the Output Response plots for both scenarios at the same time.

Right click on the input plots tab bar and select **Close Input Plots**.

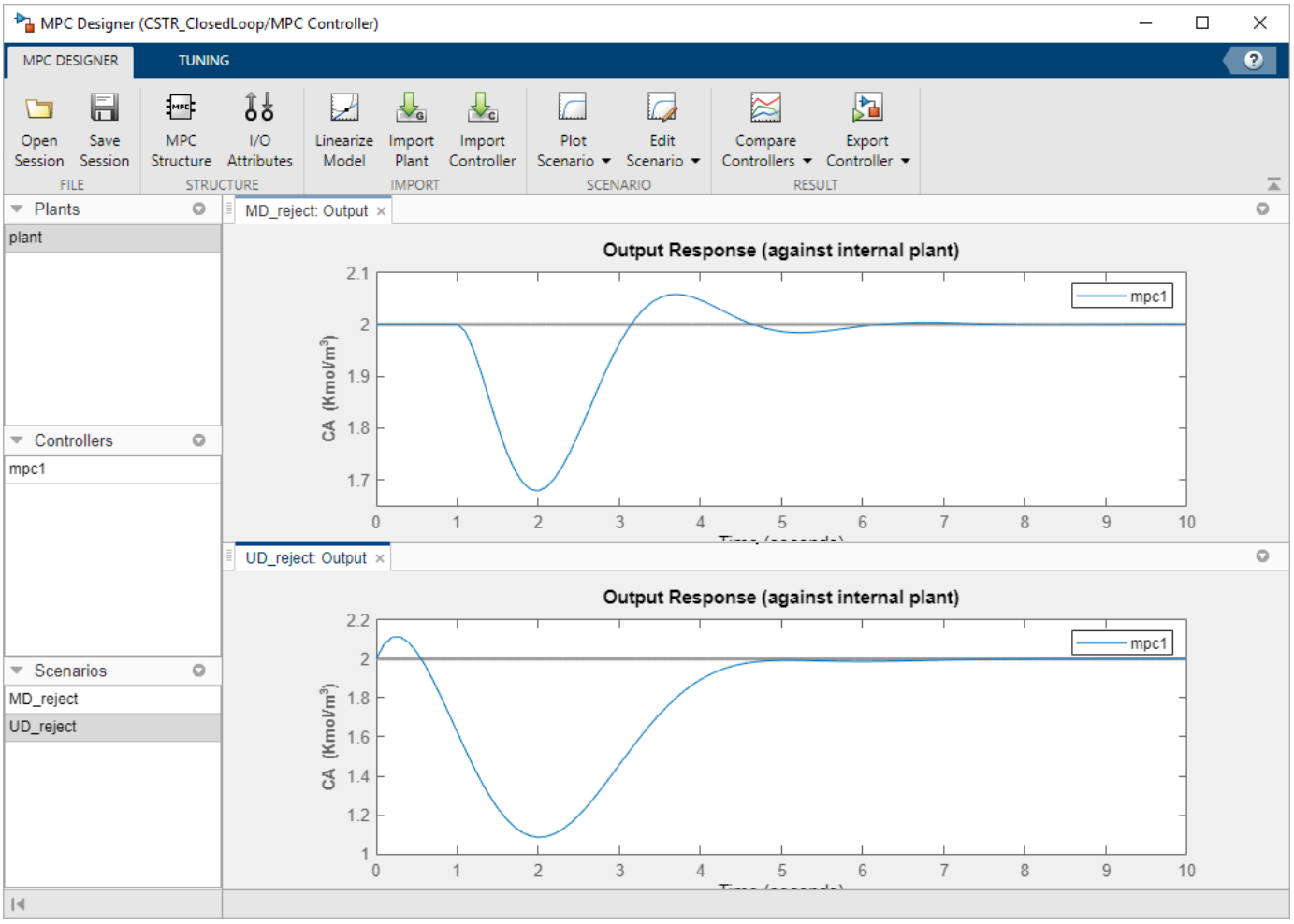


The plot display area changes to display only the output plots.

Right click on the output plots tab bar and select **Tile All** > **Top/Bottom**.



The plot display area changes to display **MD_reject: Output** tab is in the upper plot area and the **UD_reject: Output** plot is in the lower plot area.



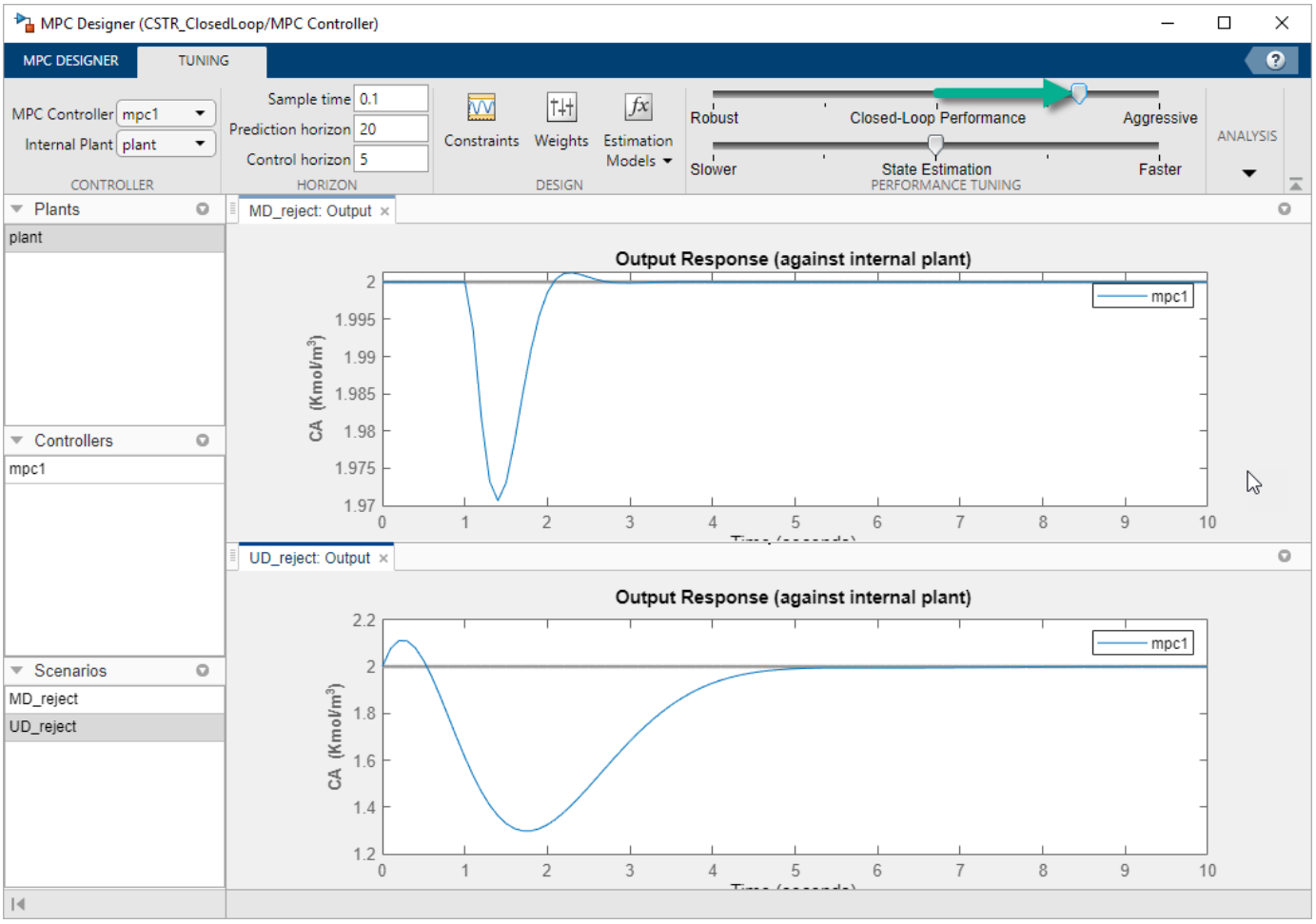
Tune Controller Performance

On the **Tuning** tab, in the **Horizon** section, specify a **Prediction horizon** of `20` and a **Control horizon** of 5.

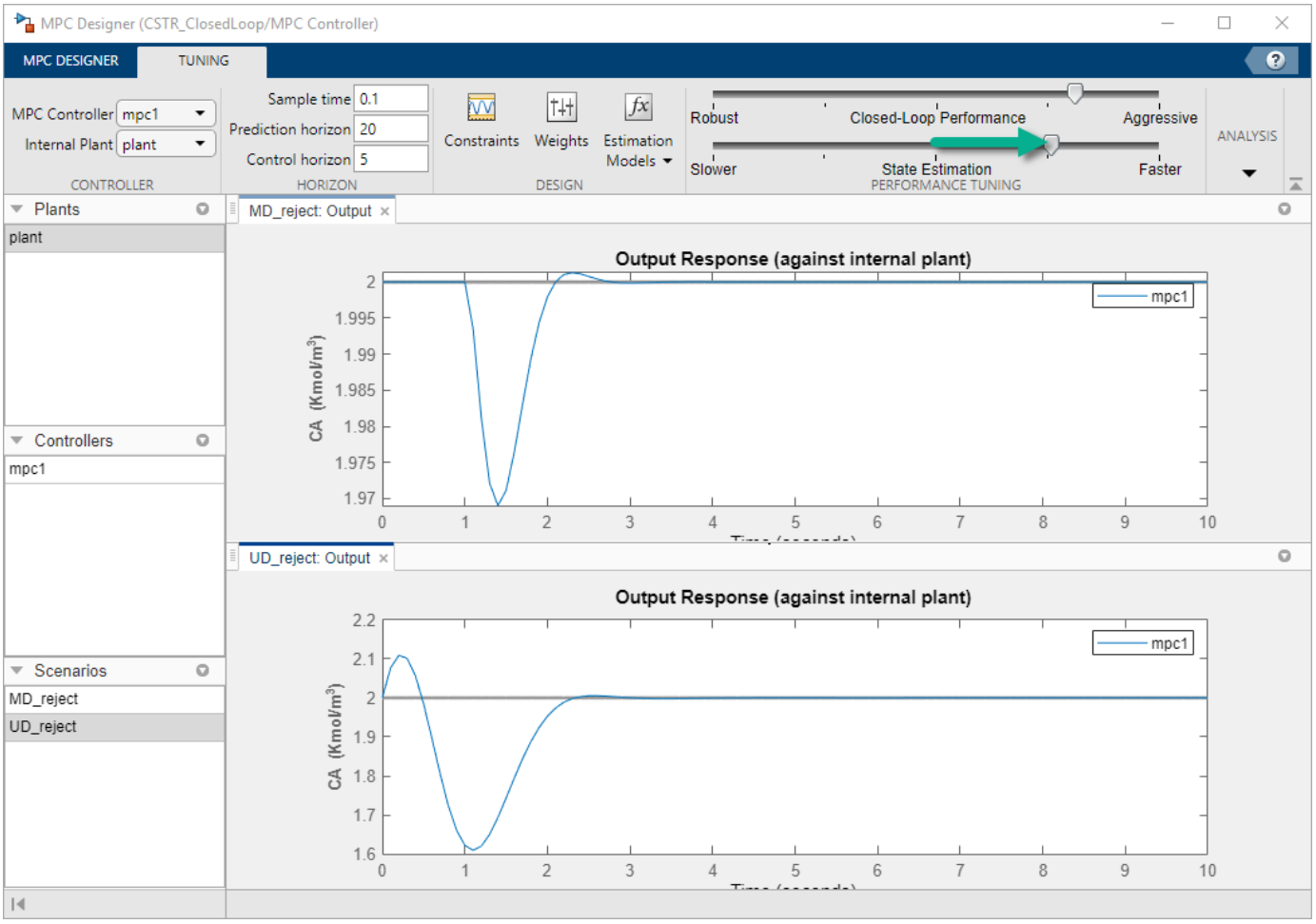
The **Output Response** plots update based on the new horizon values.

Use the default controller constraint and weight configurations.

In the **Performance Tuning** section, drag the **Closed-Loop Performance** slider to the right, which leads to tighter control of outputs and more aggressive control moves. Drag the slider until the **MD_reject: Output** response reaches steady state in less than 3 seconds.



Drag the **State Estimation** slider to the right, which leads to more aggressive unmeasured disturbance rejection. Drag the slider until the **UD_reject: Output** response reaches steady state in less than 3 seconds.



Update Simulink Model with Tuned Controller

In the **Analysis** section, select **Export Controller** > **Update Block Only**. The app exports tuned controller `mpc1` to the MATLAB workspace. In the Simulink model, the MPC Controller block is updated to use the exported controller.

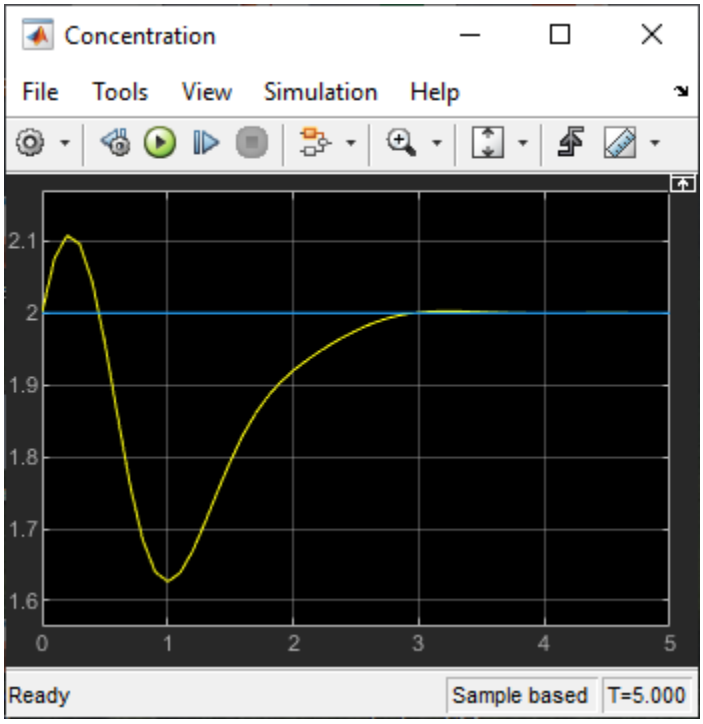
Simulate Unmeasured Disturbance Rejection

In the Simulink model window, on the **Simulation** tab, change **Stop Time** to `5` seconds.

The model initial conditions are set to the nominal operating point used for linearization.

To simulate a unit step in the feed concentration at time zero, open the Feed Concentration block and increase its **Constant value** parameter from `10` to `11`.

In the Simulink model window, open the Concentration scope and run the simulation. To scale the plot vertical axis, click on the vertical scaling button in the scope plot toolbar.



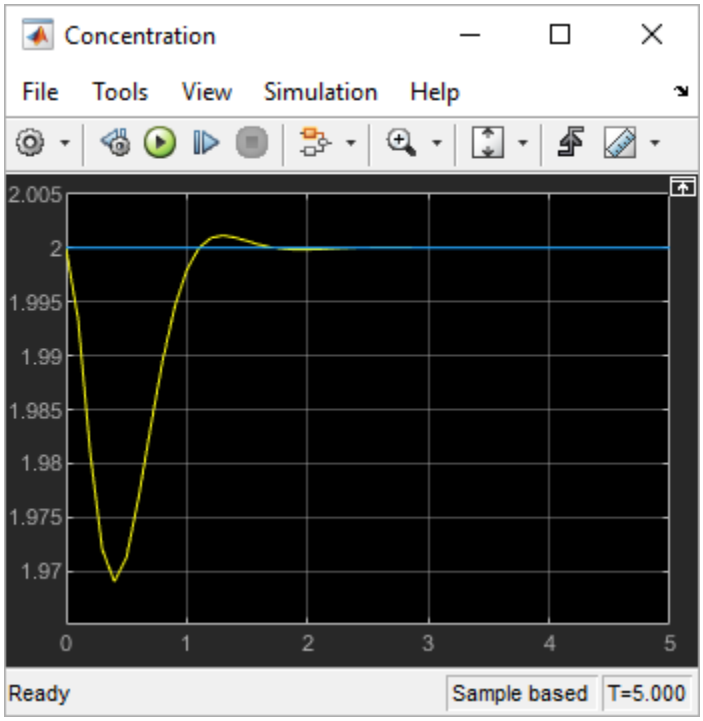
The output response is similar to the **UD_reject** response, however the settling time is around 1 second longer. The different result is due to the mismatch between the linear plant used in the **MPC Designer** simulation and the nonlinear plant in the Simulink model.

Simulate Measured Disturbance Rejection

To simulate the measured disturbance rejection, first return the Feed Concentration block to its nominal value of `10`.

To simulate a step change in the feed temperature at time zero, open the Feed Temperature block and increase its **Constant value** parameter from `300` to `310`.

Run the simulation, and then scale the plot vertical axis.



The output response is similar to the **MD_reject** response from the **MPC Designer** simulation.

See Also

Apps
MPC Designer

Functions
`linearize`

Objects
`mpc`

Blocks
MPC Controller

Related Examples

- “CSTR Model” on page 2-8
- “Linearize Simulink Models” on page 2-22
- “Design Controller Using MPC Designer” on page 3-2

More About

- “Tune Weights”
- “MPC Signal Types” on page 2-2
- “MPC Prediction Models” on page 2-3

- “What is Model Predictive Control?” on page 1-3

Model Predictive Control of a Single-Input-Single-Output Plant

This example shows how to control a double integrator plant with input saturation in Simulink®.

Define the Plant Model

Define the plant model as a double integrator (the input is the manipulated variable and the output the measured output).

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

Design the MPC Controller

Create the controller object with a sampling period of 0.1 seconds, a prediction horizon of 10 steps and a control horizon of and 3 moves.

```
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.
```

Because you have not specified the weights of the quadratic cost function to be minimized by the controller, their value is assumed to be the default one (0 for the manipulated variables, 0.1 for the manipulated variable rates, 1 for the output variables). Also, at this point the MPC problem is still unconstrained as you have not specified any constraint yet.

Specify actuator saturation limits as constraints on the manipulated variable.

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

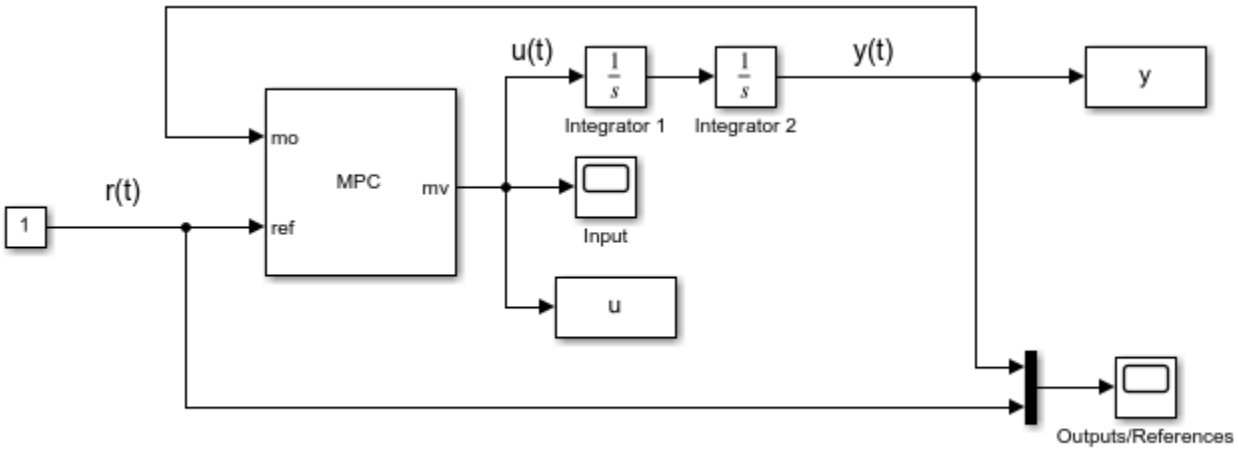
Simulate Using Simulink

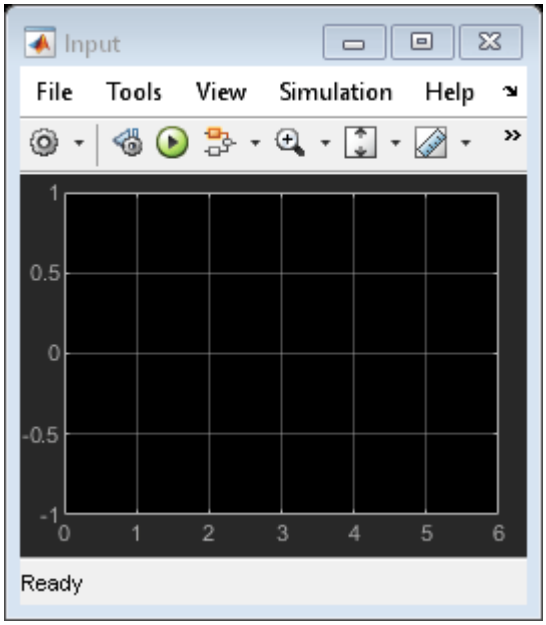
Simulink is a graphical block diagram environment for multidomain system simulation. You can connect blocks representing dynamical systems (in this case the plant and the MPC controller) and simulate the closed loop.

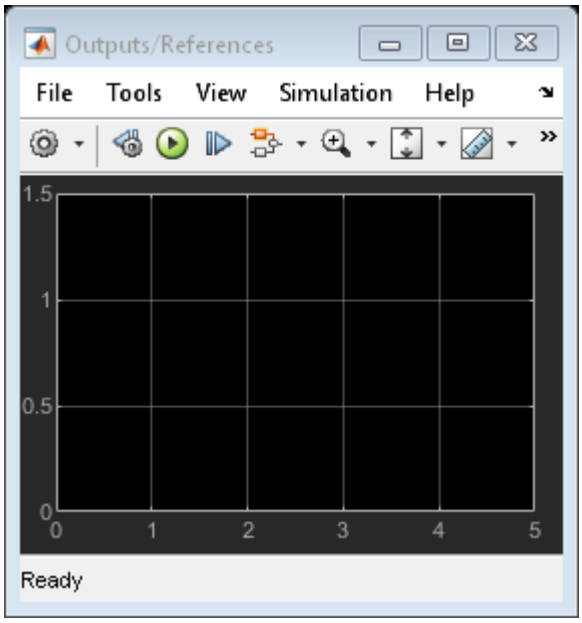
```
% Check that Simulink is installed, otherwise display a message and return
if ~mpcchecktoolboxinstalled('simulink')
    disp('Simulink is required to run this example.')
    return
end
```

Open the pre-existing Simulink model for the closed-loop simulation. The plant model is implemented with two integrator blocks in series. The variable-step ode45 integration algorithm is used to calculate the continuous time loop behavior. The MPC Controller block is configured to use the workspace `mpcobj` object as controller. The manipulated variables and the output and reference signal. The output signal is also saved by the To-Workspace block.

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



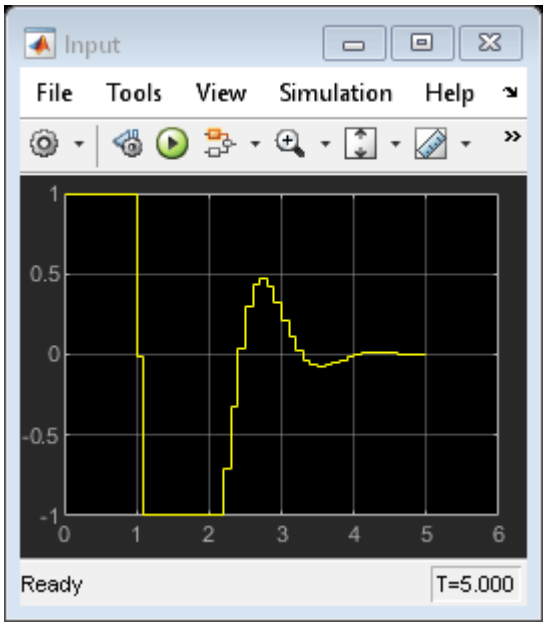


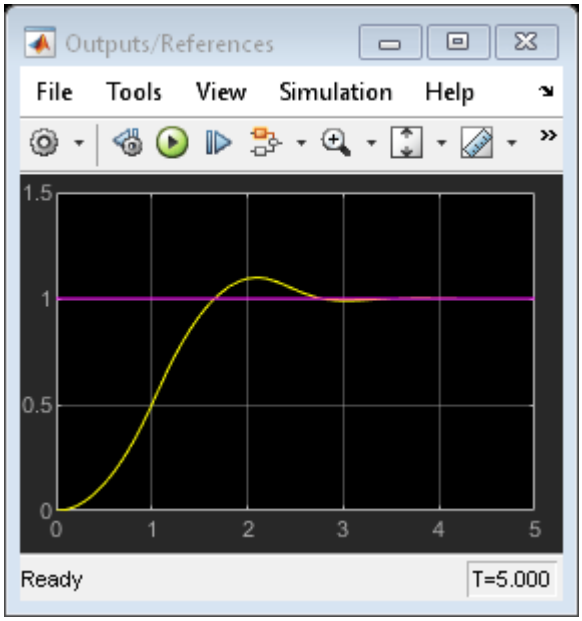


Simulate closed-loop control of the linear plant model in Simulink. Note that before the simulation starts the plant model in `mpcobj` is converted to a discrete state space model. By default, the controller uses as observer a Kalman filter designer assuming a white noise disturbance on each plant output.

```
sim(mdl)        % you can also simulate by pressing the "Run" button.
```

```
-->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.
```





The closed-loop response shows good setpoint tracking performance, as the plant output tracks its reference after about 2.5 seconds. As expected, the manipulated variable stays within the predefined constraints.

Close the open Simulink model without saving any change.

```
bdclose(mdl)
```

See Also

Apps
MPC Designer

Objects
`mpc`

Blocks
MPC Controller

Related Examples

- “Model Predictive Control of Multi-Input Single-Output Plant” on page 3-56
- “Model Predictive Control of a Multi-Input Multi-Output Nonlinear Plant” on page 3-93

More About

- “MPC Signal Types” on page 2-2
- “MPC Prediction Models” on page 2-3
- “What is Model Predictive Control?” on page 1-3

Model Predictive Control of Multi-Input Single-Output Plant

This example shows how to design, analyze, and simulate a model predictive controller with hard and soft constraints for a plant with one measured output (MO) and three inputs. The inputs consist of one manipulated variable (MV), one measured disturbance (MD), and one unmeasured disturbance (UD). After designing a controller and analyzing its closed-loop steady-state gains, you perform simulations with the `sim` command, in a `for` loop using `mpcmove`, and with Simulink®. Simulations with model mismatches, without constraints, and in open-loop are shown. Input and output disturbances and noise models are also treated, as well as how to change the Kalman gains of the built-in state estimator.

Define Plant Model

Define a plant model. For this example, use continuous-time transfer functions from each input to the output.

```
plantTF = tf({1,1,1},{[1 .5 1],[1 1],[.7 .5 1]}) % define and display tf object
```

```
plantTF =

  From input 1 to output:
         1
  ---------------
  s^2 + 0.5 s + 1

  From input 2 to output:
    1
  -----
  s + 1

  From input 3 to output:
           1
  -------------------
  0.7 s^2 + 0.5 s + 1

Continuous-time transfer function.
```

For this example, explicitly convert the plant to a discrete-time state-space form before passing it to the MPC controller creation function.

The controller creation function can accept either continuous-time or discrete-time plants. During initialization, a continuous-time plant (in any format) is automatically converted into a discrete-time state-space model using the zero-order hold (ZOH) method. Delays, if present, are incorporated in the state-space model.

You can convert the plant to discrete-time yourself when you need the discrete-time system matrices for analysis or simulation (as in this example) or when you want to use a discrete-time conversion method other than ZOH.

```
plantCSS = ss(plantTF);          % transfer function to continuous state space
Ts = 0.2;                        % specify a sample time of 0.2 seconds
plantDSS = c2d(plantCSS,Ts)      % convert to discrete-time state space, using ZOH
```

```
plantDSS =
```

```
A =
          x1       x2       x3       x4       x5
 x1   0.8862  -0.1891        0        0        0
 x2   0.1891   0.9807        0        0        0
 x3        0        0   0.8187        0        0
 x4        0        0        0    0.841  -0.2637
 x5        0        0        0   0.1846   0.9729

B =
          u1       u2       u3
 x1   0.1891        0        0
 x2  0.01929        0        0
 x3        0   0.1813        0
 x4        0        0   0.1846
 x5        0        0  0.01899

C =
        x1     x2     x3     x4     x5
 y1      0      1      1      0  1.429

D =
     u1  u2  u3
 y1   0   0   0

Sample time: 0.2 seconds
Discrete-time state-space model.
```

By default, the software assumes that all the plant input signals are manipulated variables. To specify the signal types, such as measured and unmeasured disturbances, use the `setmpcsignals` function. In this example, the first input signal is a manipulated variable, the second is a measured disturbance, and the third is an unmeasured disturbance. This information is stored in the plant model `plantDSS` and later used by the MPC controller.

```
plantDSS = setmpcsignals(plantDSS,'MV',1,'MD',2,'UD',3); % specify signal types
```

Design MPC Controller

Create the controller object, specifying the sample time, as well as the prediction and control horizons (10 and 3 steps, respectively).

```
mpcobj = mpc(plantDSS,Ts,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.
```

Since you have not specified the weights of the quadratic cost function to be minimized, the controller uses their default values (0 for manipulated variables, 0.1 for manipulated variable rates, and 1 for the output variables). Also, at this point the MPC problem is still unconstrained as you have not specified any constraint yet.

Define hard constraints on the manipulated variable.

```
mpcobj.MV = struct('Min',0,'Max',1,'RateMin',-10,'RateMax',10);
```

You can use input and output disturbance models to define the dynamic characteristics of additive input and output unmeasured disturbances. These models allow the controller to better reject such

disturbances, if they occur at run time. By default, to be able to better reject step-like disturbances, `mpc` uses an integrator as disturbance model for:

- Each unmeasured disturbance input and
- Each unmeasured disturbance acting on each measured outputs

unless doing so causes a violation of state observability.

The MPC object also has a noise model that specifies the characteristics of the additive noise that is expected on the measured output variables. By default, this, is assumed to be a unit static gain, which is equivalent to assuming that the controller expects the measured output variables to be affected, at run time, by white noise (with a covariance matrix that depends on the input matrices of the whole prediction model). For more information, see “MPC Prediction Models” on page 2-3.

Display the input disturbance model. As expected, it is a discrete-time integrator.

```
getindist(mpcobj)
```

```
-->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.

ans =

  A =
       x1
   x1   1

  B =
       UD1-wn
   x1     0.2

  C =
        x1
   UD1   1

  D =
        UD1-wn
   UD1       0

Sample time: 0.2 seconds
Discrete-time state-space model.
```

Display the output disturbance model.

```
getoutdist(mpcobj)
```

```
ans =

  Empty state-space model.
```

Specify the disturbance model for the unmeasured input as an integrator driven by white noise with variance `1000`.

```
mpcobj.Model.Disturbance = tf(sqrt(1000),[1 0]);
```

Display the input disturbance model again to verify that it changed.

```
getindist(mpcobj)
```

```
ans =

  A =
       x1
   x1   1

  B =
       Noise#1
   x1      0.8

  C =
          x1
   UD1  7.906

  D =
        Noise#1
   UD1        0

Sample time: 0.2 seconds
Discrete-time state-space model.
```

Display the MPC controller object `mpcobj` to review its properties.

```
mpcobj
```

```
MPC object (created on 03-Mar-2023 21:34:38):
---------------------------------------------
Sampling time:      0.2 (seconds)
Prediction Horizon: 10
Control Horizon:    3

Plant Model:
                                      --------------
      1  manipulated variable(s)   -->|  5 states  |
                                      |            |-->  1 measured output(s)
      1  measured disturbance(s)   -->|  3 inputs  |
                                      |            |-->  0 unmeasured output(s)
      1  unmeasured disturbance(s) -->|  1 outputs |
                                      --------------
Indices:
  (input vector)    Manipulated variables: [1 ]
                    Measured disturbances: [2 ]
                  Unmeasured disturbances: [3 ]
  (output vector)        Measured outputs: [1 ]

Disturbance and Noise Models:
        Output disturbance model: default (type "getoutdist(mpcobj)" for details)
         Input disturbance model: user specified (type "getindist(mpcobj)" for more 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(mpcobj)" for details)

Constraints:
 0 <= MV1 <= 1, -10 <= MV1/rate <= 10, MO1 is unconstrained

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

Examine Steady-State Offset

To examine whether the MPC controller can reject constant output disturbances and track a constant setpoint with zero offsets in steady state, calculate the closed-loop DC gain from output disturbances to controlled outputs using the `cloffset` command. This gain is also known as the steady state output sensitivity of the closed loop.

```
DC = cloffset(mpcobj);
fprintf('DC gain from output disturbance to output = %5.8f (=%g) \n',DC,DC);
```

```
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
DC gain from output disturbance to output = 0.00000000 (=1.66533e-15)
```

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 tracks the desired output reference setpoint perfectly in steady state.

Simulate Closed-Loop Response Using `sim`

The `sim` command provides a quick way to simulate an MPC controller in a closed loop with a linear time-invariant plant when constraints and weights stay constant and you can easily and completely specify the disturbance and reference signals ahead of time.

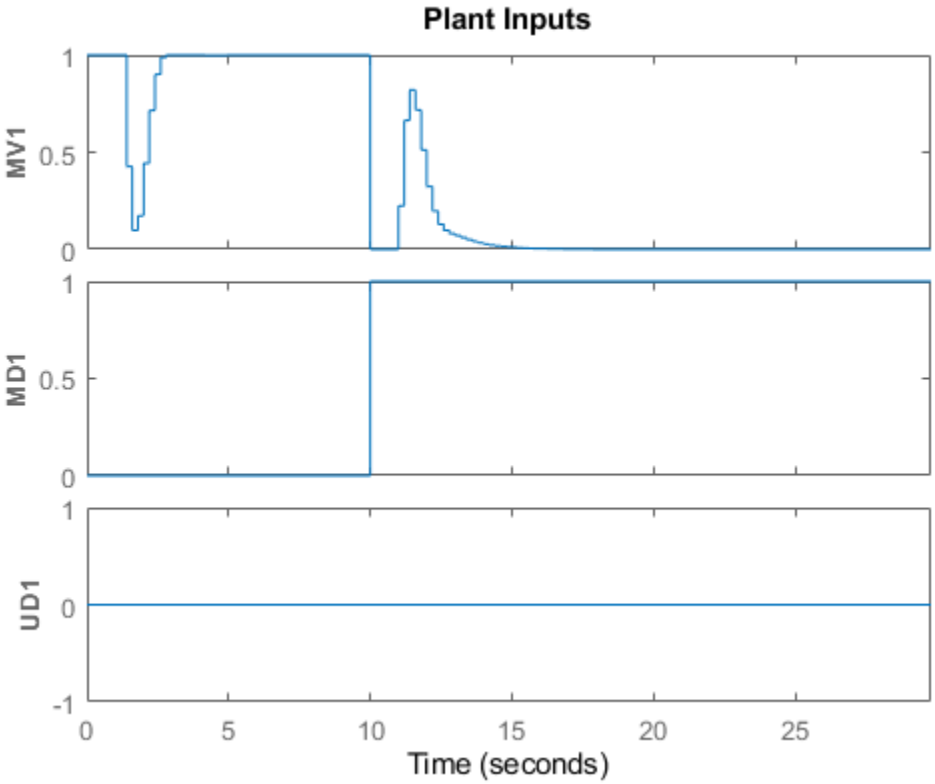
First, specify the simulation time and the reference and disturbance signals

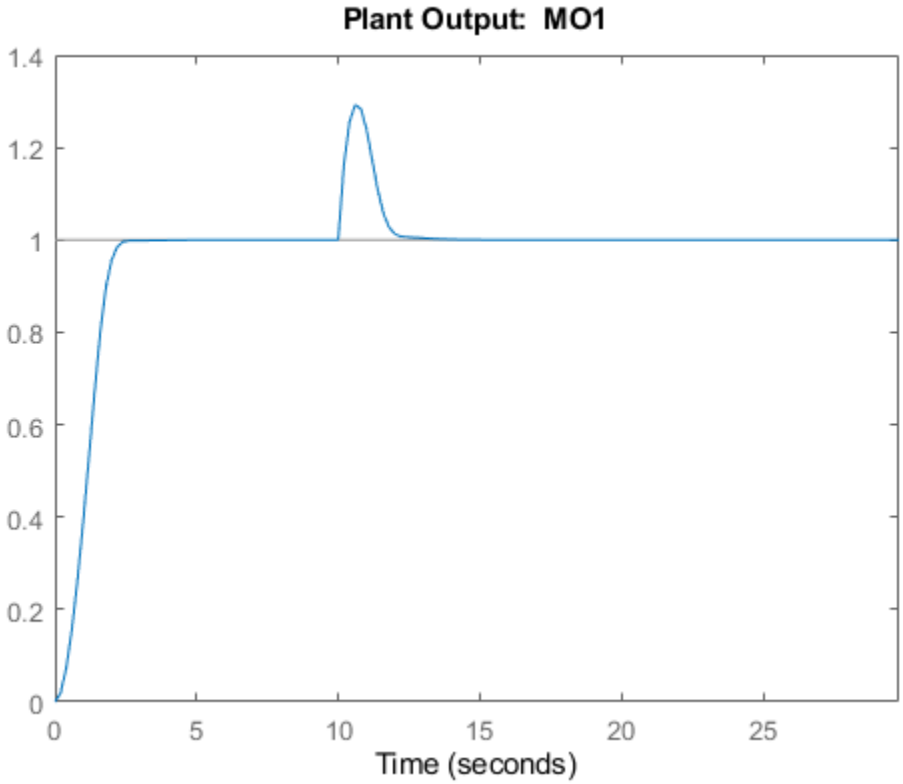
```
Tstop = 30;                                % simulation time
Nf = round(Tstop/Ts);                      % number of simulation steps
r = ones(Nf,1);                            % output reference signal
v = [zeros(Nf/3,1);ones(2*Nf/3,1)];        % measured input disturbance signal
```

Run the closed-loop simulation and plot the results. The plant specified in `mpcobj.Model.Plant` is used both as the plant in the closed-loop simulation and as the internal plant model used by the controller to predict the response over the prediction horizon. The plant model is discretized or resampled if needed, and the simulation runs in discrete time, with sample time `mpcobj.Ts`.

Use `sim` to simulate the closed-loop response to reference `r` and measured input disturbance `v` system for `Nf` steps.

```
sim(mpcobj,Nf,r,v)      % simulate plant and controller in closed loop
```


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



The manipulated variable hits the upper bound initially, and brings the plant output to the reference value within a few seconds. The manipulated variable then settles at its maximum allowed value, 1. After 10 seconds, the measured disturbance signal rises from 0 to 1, which causes the plant output to exceed its reference value by about 30%. The manipulated variable hits the lower bound in an effort to reject the disturbance. The controller is able to bring the plant output back to the reference value after a few seconds, and the manipulated variable settles at its minimum value. The unmeasured disturbance signal is always zero, because no unmeasured disturbance signal has been specified yet.

You can use a simulation options object to specify additional simulation options and additional signals, such as noise and unmeasured disturbances, that feed into the plant but are unknown to the controller. For this example, use a simulation option object to add an unmeasured input disturbance signal to the manipulated variable and to add noise on the measured output signal. Create a simulation options object with default options.

```
SimOptions = mpcsimopt;                          % create object
```

Create a disturbance signal and specify it in the simulation options object.

```
d = [zeros(2*Nf/3,1);-0.5*ones(Nf/3,1)];         % step disturbance
SimOptions.UnmeasuredDisturbance = d;            % unmeasured input disturbance
```

Specify noise signals in the simulation options object. At simulation time, the simulation function directly adds the specified output noise to the measured output before feeding it to the controller. It also directly adds the specified input noise to the manipulated variable (not to any disturbance signals) before feeding it to the plant.

```
SimOptions.OutputNoise=.001*(rand(Nf,1)-.5);  % output measurement noise
SimOptions.InputNoise=.05*(rand(Nf,1)-.5);    % noise on manipulated variables
```

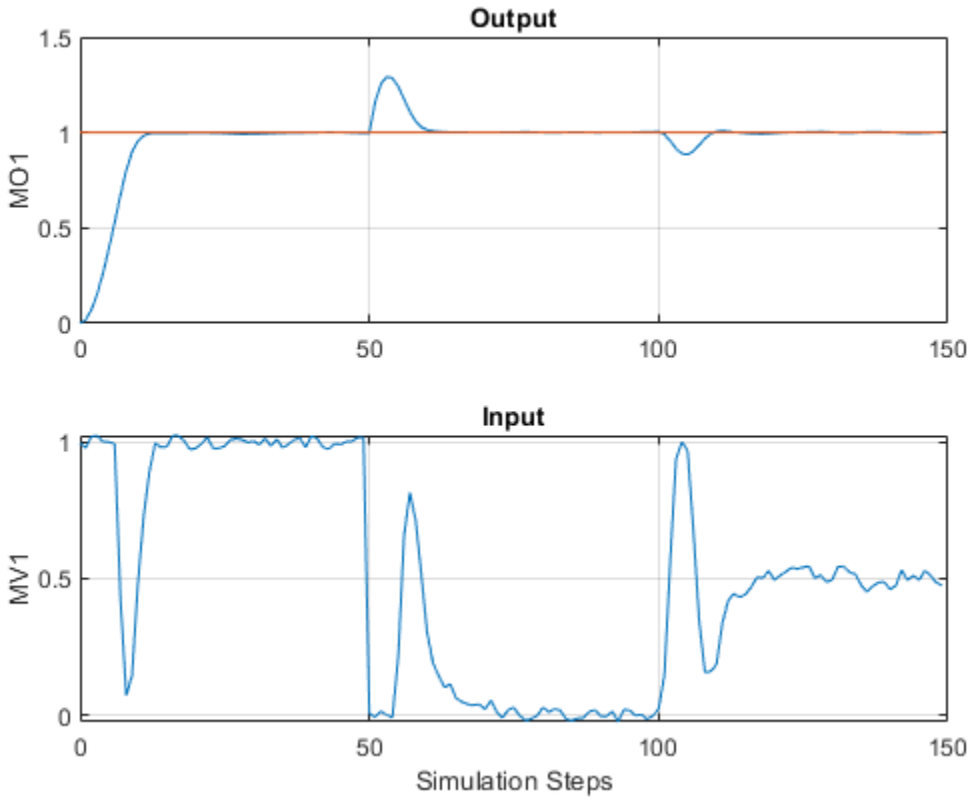
You can also use the `OutputNoise` field of the simulation option object to specify a more general additive output disturbance signal (such as a step) on the measured plant output.

Simulate the closed-loop system and save the results to the workspace variables `y`, `t`, `u`, and `xp`. Saving this variables allows you to selectively plot signals in a new figure window and in any given color and order.

```
[y,t,u,xp] = sim(mpcobj,Nf,r,v,SimOptions);
```

Plot the results.

```
figure                                  % create new figure

subplot(2,1,1)                          % create upper subplot
plot(0:Nf-1,y,0:Nf-1,r)                 % plot plant output and reference
title('Output')                         % add title so upper subplot
ylabel('MO1')                           % add a label to the upper y axis
grid                                    % add a grid to upper subplot

subplot(2,1,2)                          % create lower subplot
plot(0:Nf-1,u)                          % plot manipulated variable
title('Input');                         % add title so lower subplot
xlabel('Simulation Steps')              % add a label to the lower x axis
ylabel('MV1')                           % add a label to the lower y axis
grid                                    % add a grid to lower subplot
```



Despite the added noise, which is especially visible on the manipulated variable plot, and despite the measured and unmeasured disturbances starting after 50 and 100 steps, respectively, the controller is able to achieve good tracking. The manipulated variable settles at about 1 after the initial part of the simulation (steps from 20 to 50), at about 0 to reject the measured disturbance (steps from 70 to 100), and at about 0.5 to reject both disturbances (steps from 120 to 150).

Simulate Closed-Loop Response with Model Mismatch

Test the robustness of the MPC controller against a model mismatch. Specify the true plant that you want to use in simulation as `truePlantCSS`. For this example, the denominator of each of the three plant transfer functions has one or two coefficients that differ from the corresponding ones in the plant defined earlier in Define Plant model section, which the MPC controller uses for prediction.

```
truePlantTF = tf({1,1,1},{[1 .8 1],[1 2],[.6 .6 1]})    % specify and display transfer functions
truePlantCSS = ss(truePlantTF);                          % convert to continuous state space
truePlantCSS = setmpcsignals(truePlantCSS,'MV',1,'MD',2,'UD',3); % specify signal types
```

```
truePlantTF =

  From input 1 to output:
         1
  ---------------
  s^2 + 0.8 s + 1

  From input 2 to output:
    1
  -----
  s + 2

  From input 3 to output:
           1
  -------------------
  0.6 s^2 + 0.6 s + 1

Continuous-time transfer function.
```

Update the simulation option object by specifying `SimOptions.Model` as a structure with two fields, `Plant` (containing the true plant model) and `Nominal` (containing the operating point values for the true plant). For this example, the nominal values for the state derivatives and the inputs are not specified, so they are assumed to be zero, resulting in `y = SimOptions.Model.Nominal.Y + C*(x-SimOptions.Model.Nominal.X)`, where `x` and `y` are the state and measured output of the plant, respectively.

```
% create the structure and assign the 'Plant' field
SimOptions.Model = struct('Plant',truePlantCSS);

 % create and assign the 'Nominal.Y' field
SimOptions.Model.Nominal.Y = 0.1;

% create and assign the 'Nominal.X' field
SimOptions.Model.Nominal.X = -.1*[1 1 1 1 1];

% specify the initial state of the true plant
SimOptions.PlantInitialState = [0.1 0 -0.1 0 .05];
```

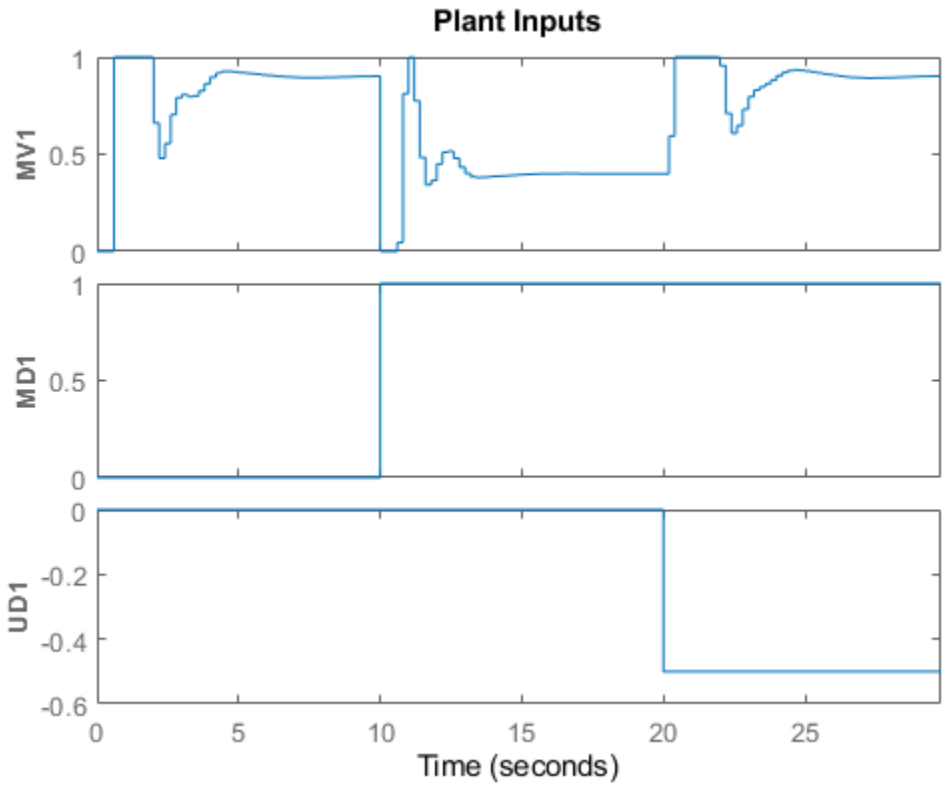
Remove any signal that have been added to the measured output and to the manipulated variable.

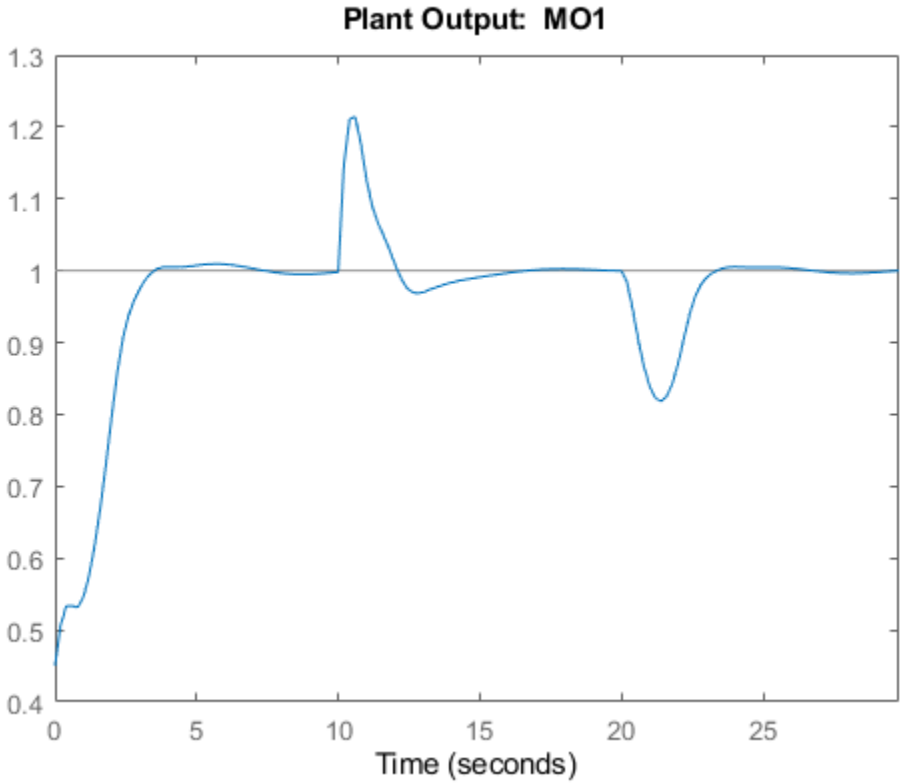
```
SimOptions.OutputNoise = [];            % remove output measurement noise
SimOptions.InputNoise = [];             % remove noise on manipulated variable
```

Run the closed-loop simulation and plot the results. Since `SimOptions.Model` is not empty, `SimOptions.Model.Plant` is converted to discrete time (using zero order hold) and used as the plant in the closed loop simulation, while the plant in `mpcobj.Model.Plant` is only used by the controller to predict the response over the prediction horizon.

```
sim(mpcobj,Nf,r,v,SimOptions)           % simulate the closed loop
```

```
-->Converting model to discrete time.
```





As a result of the model mismatch, some degradation in the response is visible; notably, the controller needs a little more time to achieve tracking and the manipulated variable now settles at about 0.5 to reject the measured disturbance (see values from 5 to 10 seconds) and settles at about 0.9 to reject both input disturbances (from 25 to 30 seconds). Despite this degradation, the controller is still able to track the output reference.

Simulate Open-Loop Response

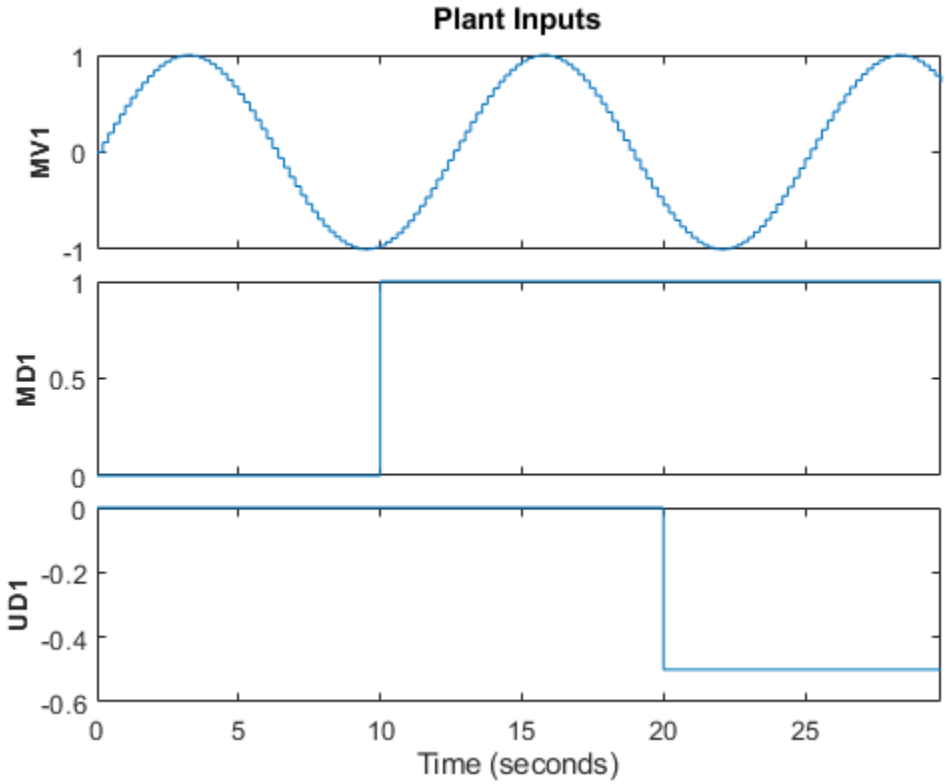
You can also test the behavior of the plant and controller in open-loop, using the `sim` command. Set the `OpenLoop` flag to `on`, and provide a sequence of manipulated variable values to excite the system (the sequence is ignored if `OpenLoop` is set to `off`).

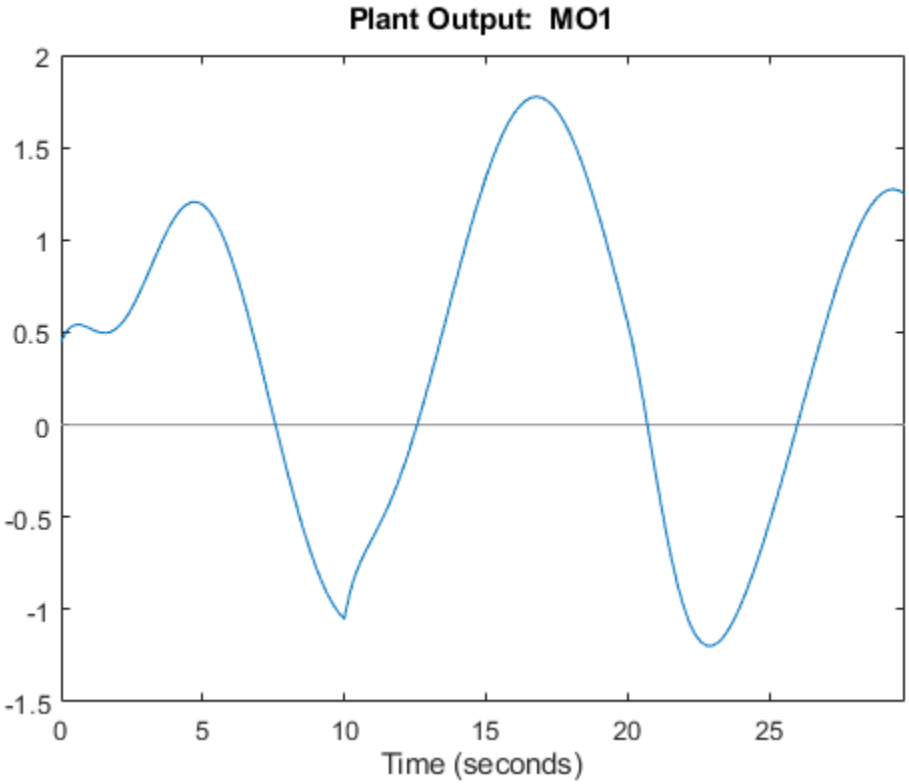
```
SimOptions.OpenLoop = 'on';                % set open loop option
SimOptions.MVSignal = sin((0:Nf-1)'/10);   % define mv signal
```

Simulate the true plant (previously specified in `SimOptions.Model`) in open loop. Since the reference signal is ignored in an open-loop simulation specify it as [].

```
sim(mpcobj,Nf,[],v,SimOptions)             % simulate the open loop system
```

```
-->Converting model to discrete time.
```


Plant Inputs
MV1
MD1
UD1
1
0
-1
1
0.5
0
0
-0.2
-0.4
-0.6
0
5
10
15
20
25
Time (seconds)



Soften Constraints

For an MPC controller, each constraint has an associated dimensionless ECR value. A constraint with a larger ECR value is allowed to be violated more than a constraint with a smaller ECR value. By default all constraints on the manipulated variables have an ECR value of zero, making them hard. You can specify a nonzero ECR value for a constraint to make it soft.

Relax the constraints on manipulated variables from hard to soft.

```
mpcobj.ManipulatedVariables.MinECR = 1;   % ECR for the MV lower bound
mpcobj.ManipulatedVariables.MaxECR = 1;   % ECR for the MV upped bound
```

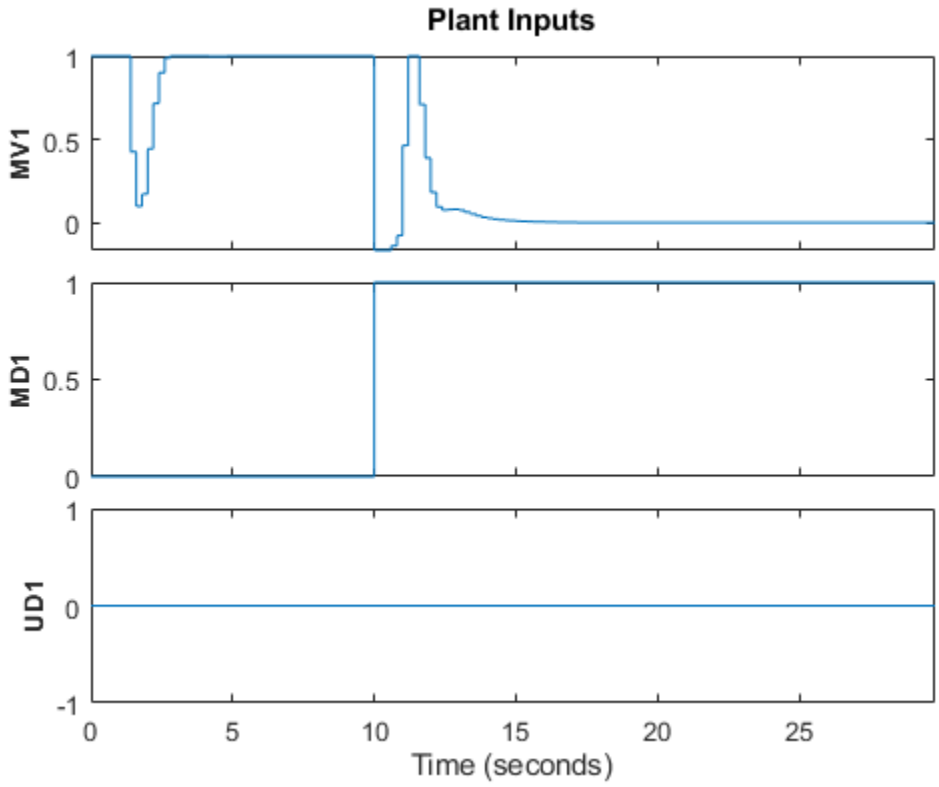
Define an output constraint. By default all constraints on output variables (measured outputs) have an ECR value of one, making them soft. You can reduce the ECR value for an output constraint to make it harder, however best practice is to keep output constraints soft. Soft output constraints are preferred because plant outputs depend on both plant states and measured disturbances; therefore, if a large enough disturbance occurs, the plant outputs constraints can be violated regardless of the plant state (and therefore regardless of any control action taken by the MPC controller). These violation are especially likely when the manipulated variables have hard constraints. Such an unavoidable violation of a hard constraint results in an infeasible MPC problem, for which no manipulated variable can be calculated.

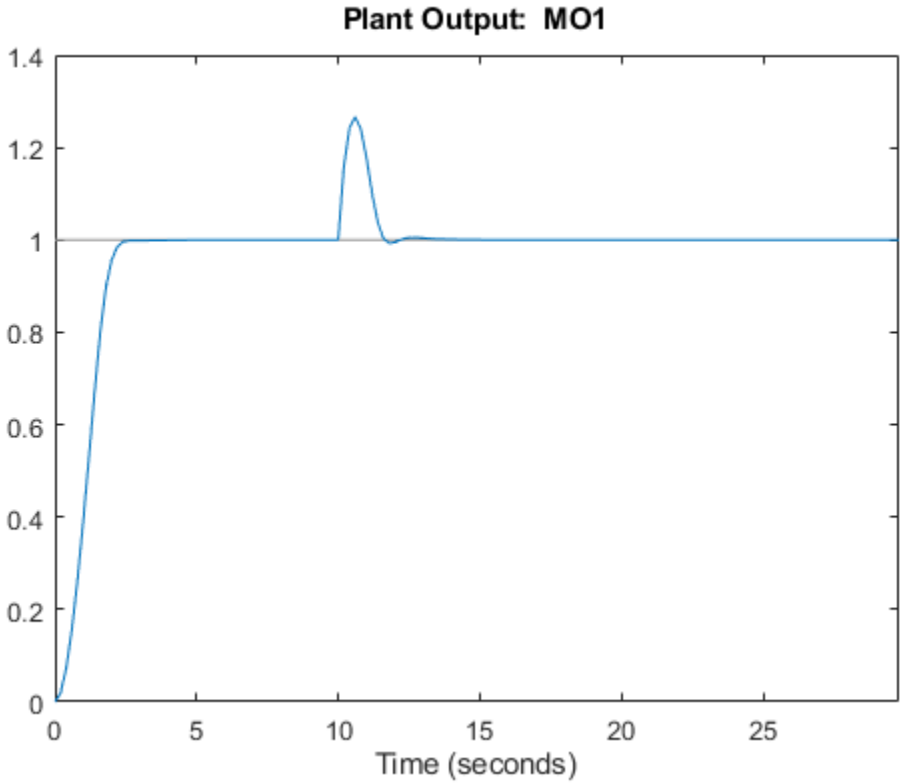
```
mpcobj.OutputVariables.Max = 1.1;    % define the (soft) output constraint
```

Run a new closed-loop simulation, without including the simulation option object, and therefore without any model mismatch, unmeasured disturbance, or noise added to the manipulated variable or measured output.

```
sim(mpcobj,Nf,r,v)          % simulate the closed loop
```

```
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```



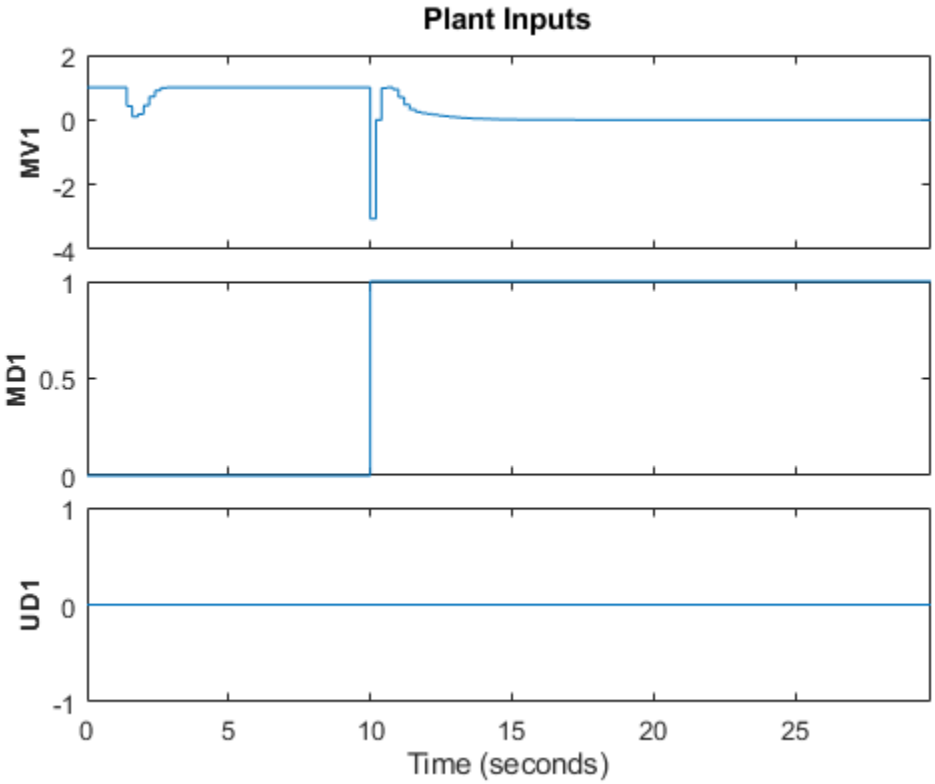


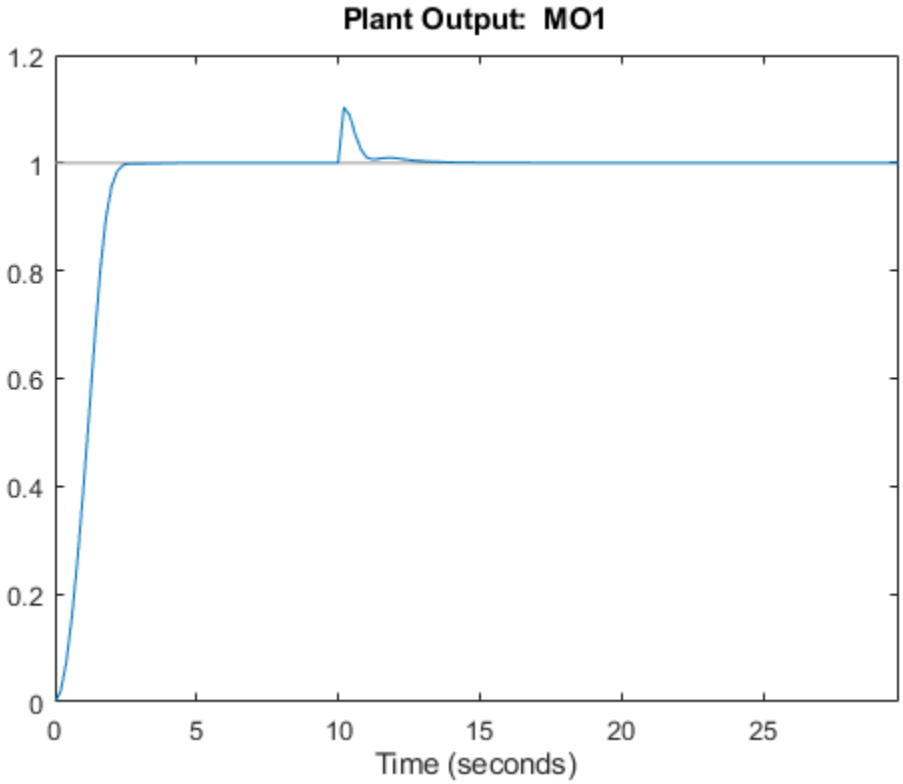
In an effort to reject the measured disturbance, achieve tracking, and prevent the output from rising above its soft constraint of 1.1, the controller slightly violates the soft constraint on the manipulated variable, which reaches small negative values from seconds 10 to 11. The controller violates the constraint on the measured output more than the constraint on the manipulated variable.

Harden the constraint on the output variable and rerun the simulation.

```
mpcobj.OV.MaxECR = 0.001;   % the closer to zero, the harder the constraint
sim(mpcobj,Nf,r,v)          % run a new closed-loop simulation.
```

```
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```


Plant Inputs
MV1
MD1
UD1
2
0
-2
-4
1
0.5
0
1
0
-1
0
5
10
15
20
25
Time (seconds)



Now the controller violates the output constraint only slightly. This output performance improvement comes at the cost of violating the manipulated variable constraint a lot more (the manipulated variable reaches a value of -3).

Change Built-In State Estimator Kalman Gains

At each time step, the MPC controller computes the manipulated variable by solving a constrained quadratic optimization problem that depends on the current state of the plant. Since the plant state is often not directly measurable, by default, the controller uses a linear Kalman filter as an observer to estimate the state of the plant and the disturbance and noise models. Therefore, the states of the controller are the states of this Kalman filter, which are in turn the estimates of the states of the augmented discrete-time plant. Run a closed-loop simulation with model mismatch and unmeasured disturbance, using the default estimator, and return the controller state structure `xc`.

```
SimOptions.OpenLoop = 'off';                    % set closed loop option
[y,t,u,xp,xc] = sim(mpcobj,Nf,r,v,SimOptions);  % run simulation
```

```
-->Simulation is in closed-loop (the "OpenLoop" property of "mpcsimopt" object is "off") and the
-->Converting model to discrete time.
```

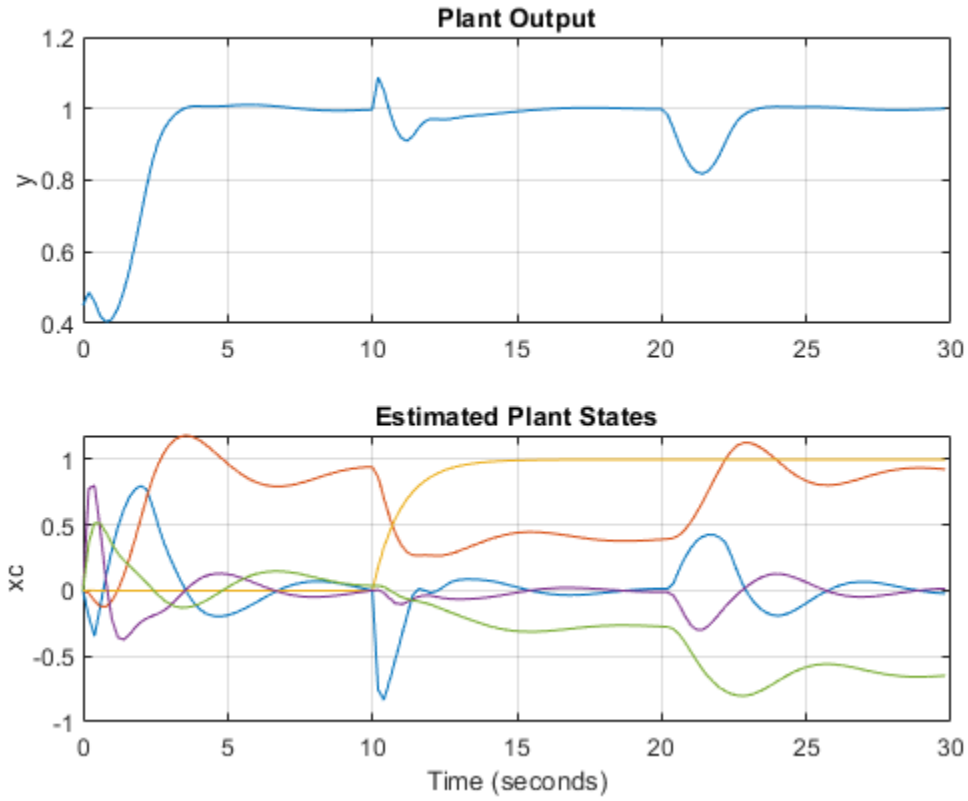
```
xc
```

```
xc =

  struct with fields:
```

```
        Plant: [150x5 double]
    Disturbance: [150x1 double]
        Noise: [150x0 double]
     LastMove: [150x1 double]
```

Plot the plant output response as well as the plant states that have been estimated by the default observer.

```
figure;                                     % create figure

subplot(2,1,1)                              % create upper subplot axis
plot(t,y)                                   % plot y versus time
title('Plant Output');                      % add title to upper plot
ylabel('y')                                 % add a label to the upper y axis
grid                                        % add grid to upper plot

subplot(2,1,2)                              % create lower subplot axis
plot(t,xc.Plant)                            % plot xc.Plant versus time
title('Estimated Plant States');            % add title to lower plot
xlabel('Time (seconds)')                    % add a label to the lower x axis
ylabel('xc')                                % add a label to the lower y axis
grid                                        % add grid to lower plot
```



As expected, the measured and unmeasured disturbances cause sudden changes at 10 and 20 seconds, respectively.

You can change the gains of the Kalman filter. To do so, first, retrieve the default Kalman gains and state-space matrices.

```
[L,M,A1,Cm1] = getEstimator(mpcobj);    % retrieve observer matrices
```

Calculate and display the poles of the default observer. They are all inside the unit circle, though a few of them seem to be relatively close to the border. Note the six states, the first five belonging to the plant model and the sixth belonging to the input disturbance model.

```
e = eig(A1-A1*M*Cm1)                    % eigenvalues of observer state matrix
```

```
e =

   0.5708 + 0.4144i
   0.5708 - 0.4144i
   0.4967 + 0.0000i
   0.9334 + 0.1831i
   0.9334 - 0.1831i
   0.8189 + 0.0000i
```

Design a new state estimator using pole placement. Move the faster poles a little toward the origin and the slowest a little away from the origin. Everything else being equal, this pole placement should result in a slightly slower observer.

```
poles = [.8 .75 .7 .85 .6 .81]; % specify desired positions for the new poles
L = place(A1',Cm1',poles)';     % calculate Kalman gain for time update
M = A1\L;                       % calculate Kalman gain for measurement update
```

Set the new matrix gains in the MPC controller object.

```
setEstimator(mpcobj,L,M);               % set the new estimation gains
```

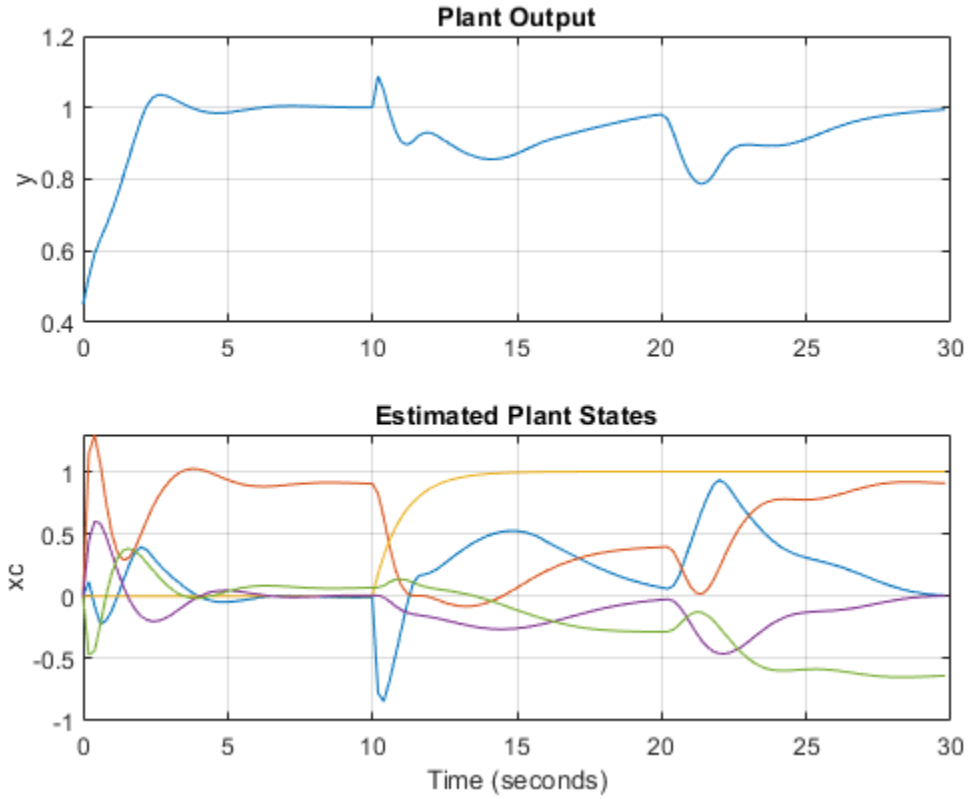
Rerun the closed-loop simulation.

```
[y,t,u,xp,xc] = sim(mpcobj,Nf,r,v,SimOptions);
```

```
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
-->Simulation is in closed-loop (the "OpenLoop" property of "mpcsimopt" object is "off") and the
-->Converting model to discrete time.
```

Plot the plant output response as well as the plant states estimated by the new observer.

```
figure;                                     % create figure

subplot(2,1,1)                              % create upper subplot axis
plot(t,y)                                   % plot y versus time
title('Plant Output');                      % add title to upper plot
ylabel('y')                                 % add a label to the upper y axis
grid                                        % add grid to upper plot

subplot(2,1,2)                              % create lower subplot axis
plot(t,xc.Plant)                            % plot xc.Plant versus time
title('Estimated Plant States');            % add title to lower plot
xlabel('Time (seconds)')                    % add a label to the lower x axis
ylabel('xc')                                % add a label to the lower y axis
grid                                        % add grid to lower plot
```



As expected, the controller states are different from the ones previously plotted, and the overall closed-loop response is somewhat slower.

Simulate Controller in Closed Loop Using `mpcmove`

For more general applications, you can simulate an MPC controller in a for loop using the `mpcmove` function. Using this function, you can run simulations with the following features.

- Nonlinear or time-varying plants
- Constraints or weights that vary at run time
- Disturbance or reference signals that are not known before running the simulation

If your plant is continuous, you can either convert it to discrete time before simulating or you can use a numerical integration algorithm (such as forward Euler or `ode45`) to simulate it in a closed loop using `mpcmove`. For example, you can calculate the plant state at the next control interval using the following methods:

- Discrete time plant x(t+1)=f(x(t),u(t)): `x = f(x,u)`, (typically `x = A*x + B*u` for linear plant models)
- Continuous time plant dx(t)/dt=f(x(t),u(t)), sample time `Ts`, Euler method: `x = x + f(x,u)*Ts`
- Continuous time plant as above, using `ode45`: `[~,xhist] = ode45(@(t,xode) f(xode,u), [0 Ts],x); x = xhist(end);`

In the third case, `ode45` starts from the initial condition `x` and simulates the plant for `Ts` seconds, under a constant control signal `u`. The last value of the resulting state signal `xhist` is the plant state at the next control interval.

First, obtain the discrete-time state-space matrices of the plant, and define the simulation time and initial states for plant and controller.

```
[A,B,C,D] = ssdata(plantDSS);       % discrete-time plant plant ss matrices
Tstop = 30;                         % simulation time
x = [0 0 0 0 0]';                   % initial state of the plant
xmpc = mpcstate(mpcobj);            % get handle to controller state
r = 1;                              % output reference signal
```

Display the initial state of the controller. The 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 models states. * `xmpc.Noise` is the current value of the noise models states. * `xmpc.LastMove` is the last value of the manipulated variable. * `xmpc.Covariance` is the current value of the estimator covariance matrix.

```
xmpc                                % display controller states
```

```
MPCSTATE object with fields
          Plant: [0 0 0 0 0]
    Disturbance: 0
          Noise: [1x0 double]
       LastMove: 0
     Covariance: []
```

Note that `xmpc` is a handle object, which always points to the current state of the controller. Since `mpcmove` updates the internal plant state when a new control move is calculated, you do not need to update `xmpc`, which always points to the current (hence updated) state.

```
isa(xmpc,'handle')
```

```
ans =

  logical

   1
```

Define workspace arrays `YY` and `UU` to store output and input signals, respectively, so that you can plot them after the simulation.

```
YY=[];
UU=[];
```

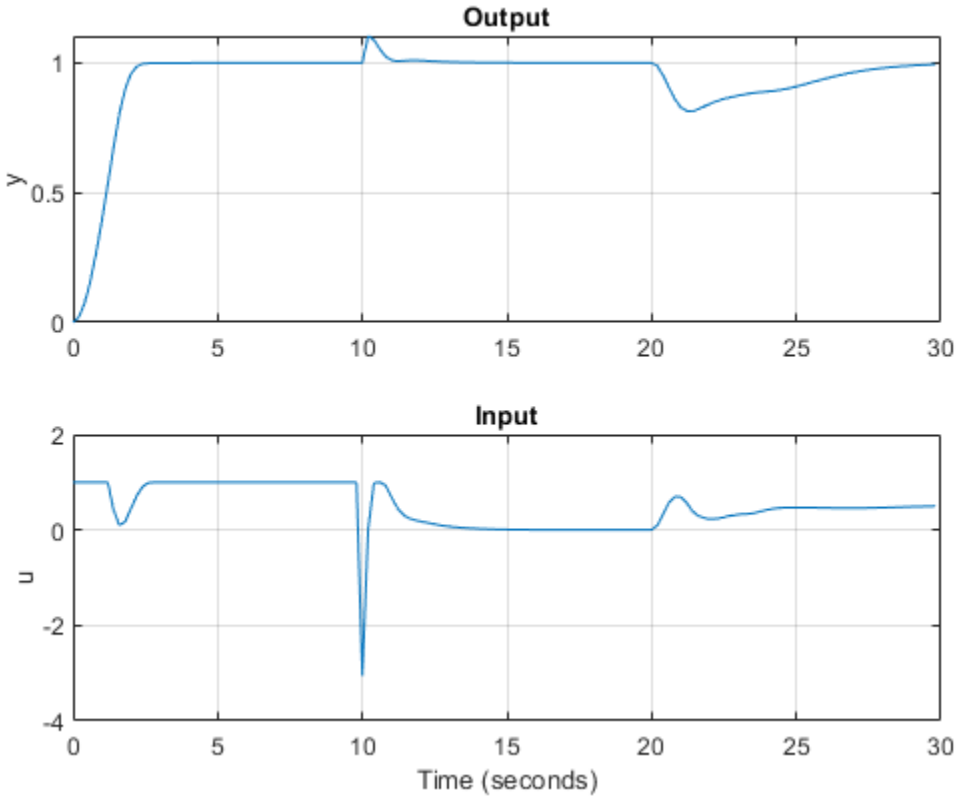
Run the simulation loop.

```
for k=0:round(Tstop/Ts)-1

    % Define measured disturbance signal v(k). You can specify a more
    % complex dependence on time or previous states here, if needed.
    v = 0;
    if k*Ts>=10         % raising to 1 after 10 seconds
```

```
        v = 1;
    end

    % Define unmeasured disturbance signal d(k). You can specify a more
    % complex dependence on time or previous states here, if needed.
    d = 0;
    if k*Ts>=20          % falling to -0.5 after 20 seconds
       d = -0.5;
    end

    % Plant equations: current output
    % If you have a more complex plant output behavior (including, for example,
    % model mismatch or nonlinearities) you can to simulate it here.
    % Note that there cannot be any direct feedthrough between u and y.
    y = C*x + D(:,2)*v + D(:,3)*d;   % calculate current output (D(:,1)=0)
    YY = [YY,y];                     % store current output

    % Note, if the plant had a non-zero operating point the output would be:
    % y = mpcobj.Model.Nominal.Y + C*(x-mpcobj.Model.Nominal.X) + D(:,2)*v + D(:,3)*d;

    % Compute the MPC action (u) and update the internal controller states.
    % Note that you do not need the update xmpc because it always points
    % to the current controller states.
    u = mpcmove(mpcobj,xmpc,y,r,v);     % xmpc,y,r,v are values at current step k
    UU = [UU,u];                        % store current input

    % Plant equations: state update
    % You can simulate a more complex plant state behavior here, if needed.
    x = A*x + B(:,1)*u + B(:,2)*v + B(:,3)*d;   % update state

    % Note, if the plant had a non-zero operating point the state update would be:
    % x = mpcobj.Model.Nominal.X + mpcobj.Model.Nominal.DX + ...
    % A*(x-mpcobj.Model.Nominal.X) + B(:,1)*(u-mpcobj.Model.Nominal.U(1)) + ...
    % B(:,2)*v + B(:,3)*d;

end
```

Plot the results.

```
figure                                  % create figure

subplot(2,1,1)                          % create upper subplot axis
plot(0:Ts:Tstop-Ts,YY)                  % plot YY versus time
ylabel('y')                             % add a label to the upper y axis
grid                                    % add grid to upper plot
title('Output')                         % add title to upper plot

subplot(2,1,2)                          % create lower subplot axis
plot(0:Ts:Tstop-Ts,UU)                  % plot UU versus time
ylabel('u')                             % add a label to the lower y axis
xlabel('Time (seconds)')                % add a label to the lower x axis
grid                                    % add grid to lower plot
title('Input')                          % add title to lower plot
```



To check the optimal predicted trajectories at any point during the simulation, you can use the second output argument of `mpcmove`. For this example, assume you start from the current state (`x` and `xmpc`). Also assume that, from this point until the end of the horizon, the reference set-point is `0.5` and the disturbance is `0`. Simulate the controller and return the info structure.

```
r = 0.5;                                        % reference
v = 0;                                          % disturbance
[~,info] = mpcmove(mpcobj,xmpc,y,r,v);          % solve over prediction horizon
```

Display the info variable.

```
info
```

```
info =

  struct with fields:

         Uopt: [11x1 double]
         Yopt: [11x1 double]
         Xopt: [11x6 double]
         Topt: [11x1 double]
        Slack: 0
   Iterations: 1
       QPCode: 'feasible'
         Cost: 0.1399
```

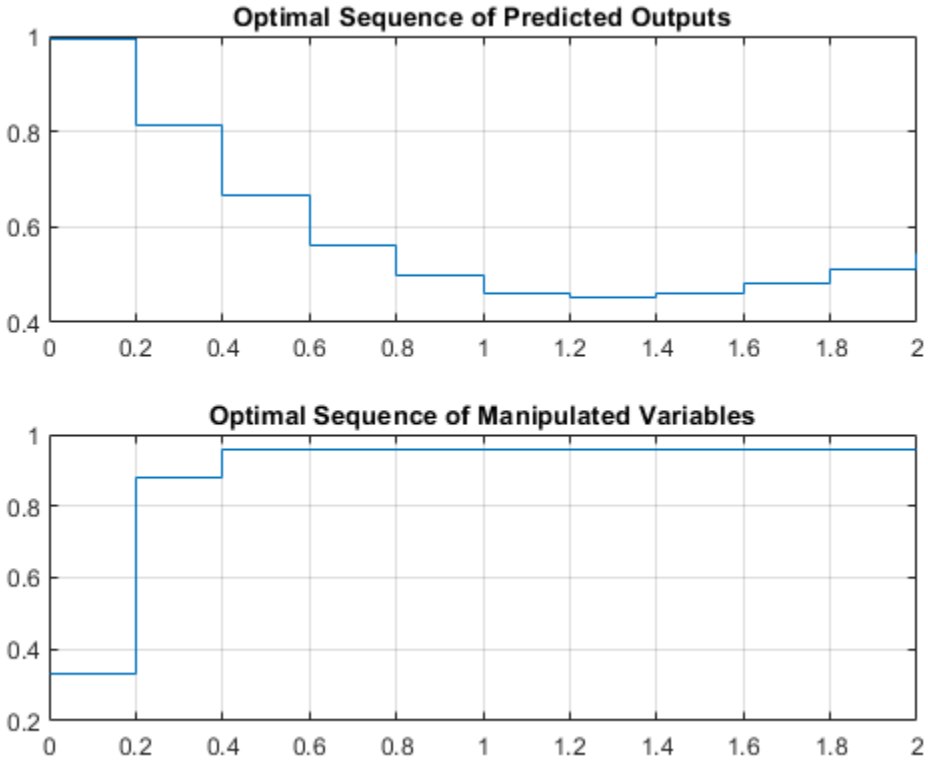
`info` is a structure containing the predicted optimal sequences of manipulated variables, plant states, and outputs over the prediction horizon. `mpcmove` calculated this sequence, together with the optimal first move, by solving a quadratic optimization problem to minimize the cost function. The plant states and outputs in `info` result from applying the optimal manipulated variable sequence directly to `mpcobj.Model.Plant`, in an open-loop fashion. Due to the presence of noise, unmeasured disturbances, and uncertainties, this open-loop optimization process is not equivalent to simulating the closed loop consisting of the plant, estimator and controller using either the `sim` command or `mpcmove` iteratively in a for loop.

Extract the predicted optimal trajectories.

```
topt = info.Topt;                    % time
yopt = info.Yopt;                    % predicted optimal plant model outputs
uopt = info.Uopt;                    % predicted optimal mv sequence
```

Since the optimal sequence values are constant across each control step, plot the trajectories using a stairstep plot.

```
figure                                                  % create new figure

subplot(2,1,1)                                          % create upper subplot
stairs(topt,yopt)                                       % plot yopt in a stairstep graph
title('Optimal Sequence of Predicted Outputs')          % add title to upper subplot
grid                                                    % add grid to upper subplot

subplot(2,1,2)                                          % create lower subplot
stairs(topt,uopt)                                       % plot uopt in a stairstep graph
title('Optimal Sequence of Manipulated Variables')      % add title to upper subplot
grid                                                    % add grid to upper subplot
```



Linear Representation of MPC Controller

When the constraints are not active, the MPC controller behaves like a linear controller. Note that for a finite-time unconstrained Linear Quadratic Regulator problem with a finite non-receding horizon, the value function is time-dependent, which causes the optimal feedback gain to be time varying. In contrast, in MPC the horizon has a constant length because it is always receding, resulting in a time-invariant value function and consequently a time-invariant optimal feedback gain.

You can get the state-space form of the MPC controller.

```
LTI = ss(mpcobj,'rv');                  % get state-space representation
```

Get the state-space matrices to simulate the linearized controller.

```
[AL,BL,CL,DL] = ssdata(LTI);            % get state-space matrices
```

Initialize variables for a closed-loop simulation of both the original MPC controller without constraints and the linearized controller.

```
mpcobj.MV = [];             % remove input constraints
mpcobj.OV = [];             % remove output constraints

Tstop = 5;                  % set simulation time
y = 0;                      % set nitial measured output
r = 1;                      % set output reference set-point (constant)
u = 0;                      % set previous (initial) input command
```

```
x = [0 0 0 0 0]';           % set initial state of plant
xmpc = mpcstate(mpcobj);    % set initial state of unconstrained MPC controller
xL = zeros(size(BL,1),1);   % set initial state of linearized MPC controller

YY = [];                    % define workspace array to store plant outputs
```

```
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

Simulate both controllers in a closed loop with the same plant model.

```
for k = 0:round(Tstop/Ts)-1

    YY = [YY,y];            % store current output for plotting purposes

    % Define measured disturbance signal v(k).
    v = 0;
    if k*Ts>=10
        v = 1;              % raising to 1 after 10 seconds
    end

    % Define unmeasured disturbance signal d(k).
    d = 0;
    if k*Ts>=20
        d = -0.5;           % falling to -0.5 after 20 seconds
    end

    % Compute the control actions of both (unconstrained) MPC and linearized MPC
    uMPC = mpcmove(mpcobj,xmpc,y,r,v);   % unconstrained MPC (also updates xmpc)
    u = CL*xL + DL*[y;r;v];              % unconstrained linearized MPC

    % Compare the two control actions
    dispStr(k+1) = {sprintf(['t=%5.2f, u=%7.4f (provided by LTI), u=%7.4f' ...
        ' (provided by MPCOBJ)'],k*Ts,u,uMPC)}; %#ok<*SAGROW>

    % Update state of unconstrained linearized MPC controller
    xL = AL*xL + BL*[y;r;v];

    % Update plant state
    x = A*x + B(:,1)*u + B(:,2)*v + B(:,3)*d;

    % Calculate plant output
    y = C*x + D(:,1)*u + D(:,2)*v + D(:,3)*d;       % D(:,1)=0

end
```

Display the character arrays containing the control actions.

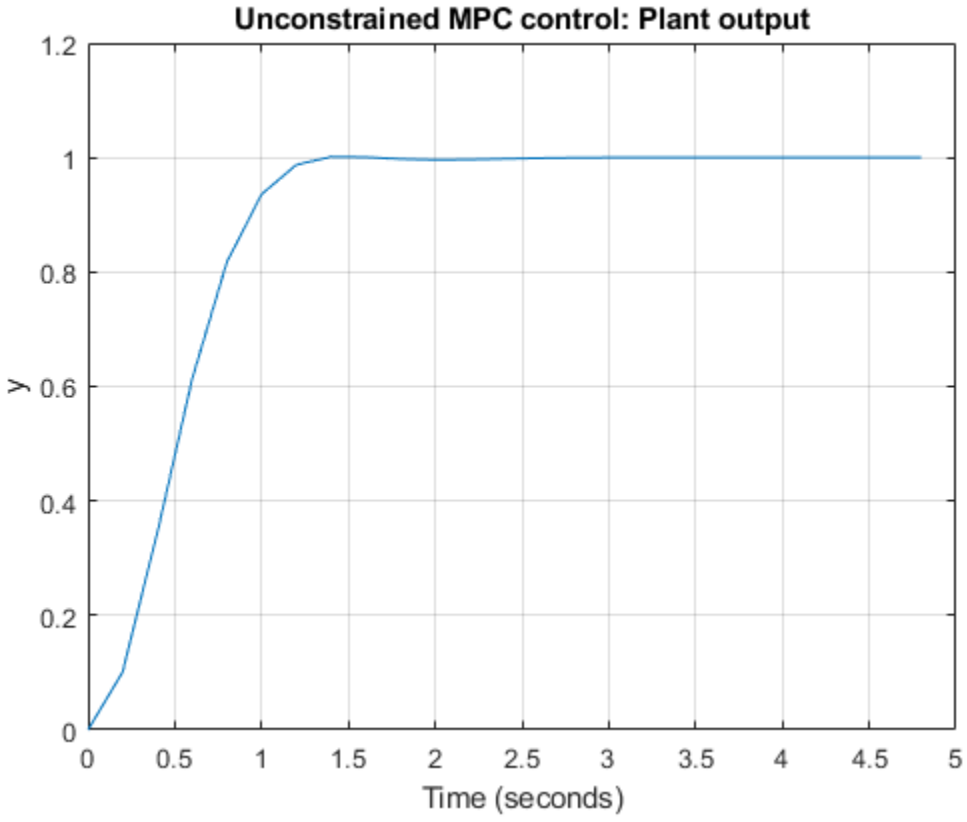
```
for k=0:round(Tstop/Ts)-1
    disp(dispStr{k+1});             % display each string as k increases
end
```

```
t= 0.00, u= 5.2478 (provided by LTI), u= 5.2478 (provided by MPCOBJ)
t= 0.20, u= 3.0134 (provided by LTI), u= 3.0134 (provided by MPCOBJ)
t= 0.40, u= 0.2281 (provided by LTI), u= 0.2281 (provided by MPCOBJ)
t= 0.60, u=-0.9952 (provided by LTI), u=-0.9952 (provided by MPCOBJ)
t= 0.80, u=-0.8749 (provided by LTI), u=-0.8749 (provided by MPCOBJ)
t= 1.00, u=-0.2022 (provided by LTI), u=-0.2022 (provided by MPCOBJ)
t= 1.20, u= 0.4459 (provided by LTI), u= 0.4459 (provided by MPCOBJ)
```

```
t= 1.40, u= 0.8489 (provided by LTI), u= 0.8489 (provided by MPCOBJ)
t= 1.60, u= 1.0192 (provided by LTI), u= 1.0192 (provided by MPCOBJ)
t= 1.80, u= 1.0511 (provided by LTI), u= 1.0511 (provided by MPCOBJ)
t= 2.00, u= 1.0304 (provided by LTI), u= 1.0304 (provided by MPCOBJ)
t= 2.20, u= 1.0053 (provided by LTI), u= 1.0053 (provided by MPCOBJ)
t= 2.40, u= 0.9920 (provided by LTI), u= 0.9920 (provided by MPCOBJ)
t= 2.60, u= 0.9896 (provided by LTI), u= 0.9896 (provided by MPCOBJ)
t= 2.80, u= 0.9925 (provided by LTI), u= 0.9925 (provided by MPCOBJ)
t= 3.00, u= 0.9964 (provided by LTI), u= 0.9964 (provided by MPCOBJ)
t= 3.20, u= 0.9990 (provided by LTI), u= 0.9990 (provided by MPCOBJ)
t= 3.40, u= 1.0002 (provided by LTI), u= 1.0002 (provided by MPCOBJ)
t= 3.60, u= 1.0004 (provided by LTI), u= 1.0004 (provided by MPCOBJ)
t= 3.80, u= 1.0003 (provided by LTI), u= 1.0003 (provided by MPCOBJ)
t= 4.00, u= 1.0001 (provided by LTI), u= 1.0001 (provided by MPCOBJ)
t= 4.20, u= 1.0000 (provided by LTI), u= 1.0000 (provided by MPCOBJ)
t= 4.40, u= 0.9999 (provided by LTI), u= 0.9999 (provided by MPCOBJ)
t= 4.60, u= 1.0000 (provided by LTI), u= 1.0000 (provided by MPCOBJ)
t= 4.80, u= 1.0000 (provided by LTI), u= 1.0000 (provided by MPCOBJ)
```

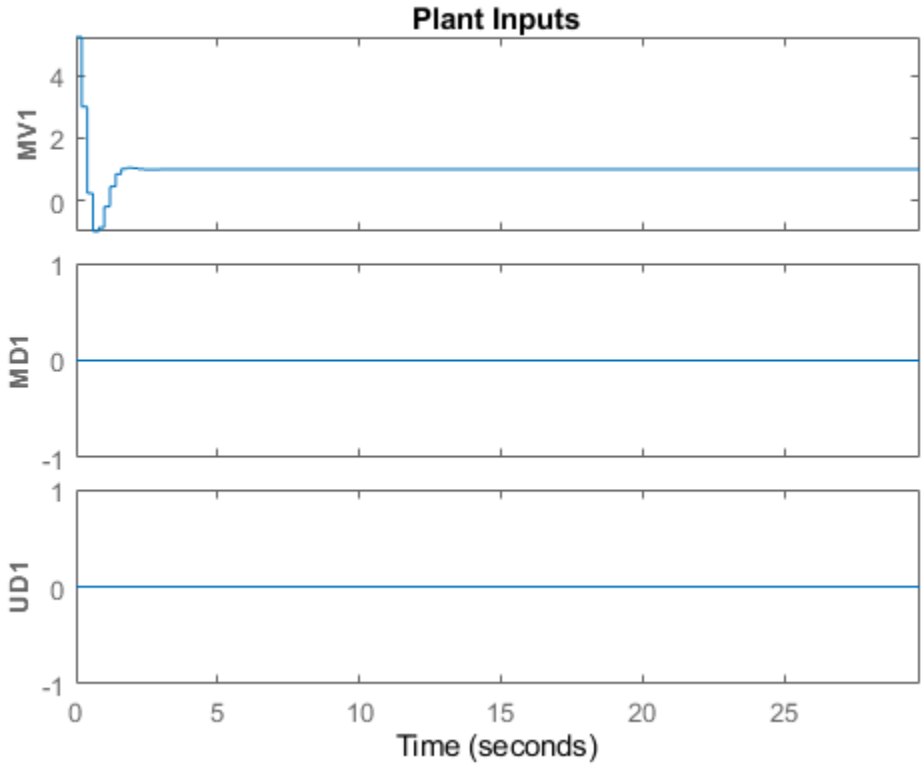
Plot the results.

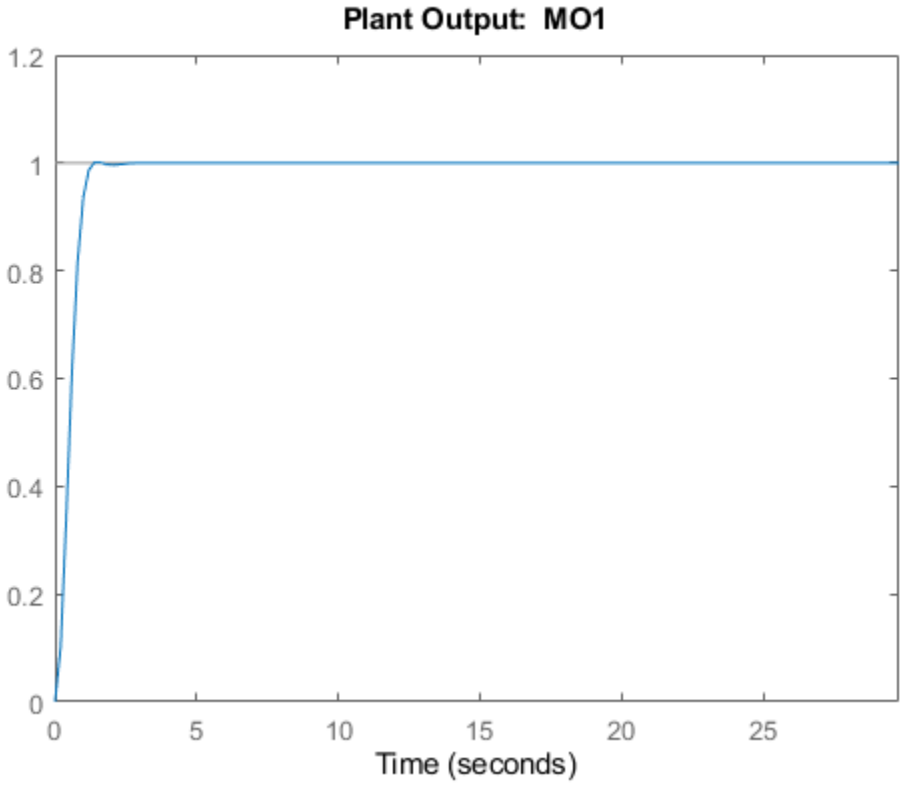
```
figure                                                  % create figure
plot(0:Ts:Tstop-Ts,YY)                                  % plot plant outputs
grid                                                    % add grid
title('Unconstrained MPC control: Plant output')        % add title
xlabel('Time (seconds)')                                % add label to x axis
ylabel('y')                                             % add label to y axis
```



Running a closed-loop simulation in which all controller constraints are turned off is easier using `sim`, as you just need to specify `'off'` in the `Constraint` field of the related `mpcsimopt` simulation option object.

```
SimOptions = mpcsimopt;                    % create simulation options object
SimOptions.Constraints = 'off';            % remove all MPC constraints
SimOptions.UnmeasuredDisturbance = d;      % unmeasured input disturbance
sim(mpcobj,Nf,r,v,SimOptions);             % run closed-loop simulation
```





Simulate Using Simulink

You can also simulate your MPC controller in Simulink.

To compare results, recreate the MPC object with the constraints you use in the Design MPC Controller section, and the default estimator.

```
mpcobj = mpc(plantDSS,Ts,10,3);
mpcobj.MV = struct('Min',0,'Max',1,'RateMin',-10,'RateMax',10);
mpcobj.Model.Disturbance = tf(sqrt(1000),[1 0]);
```

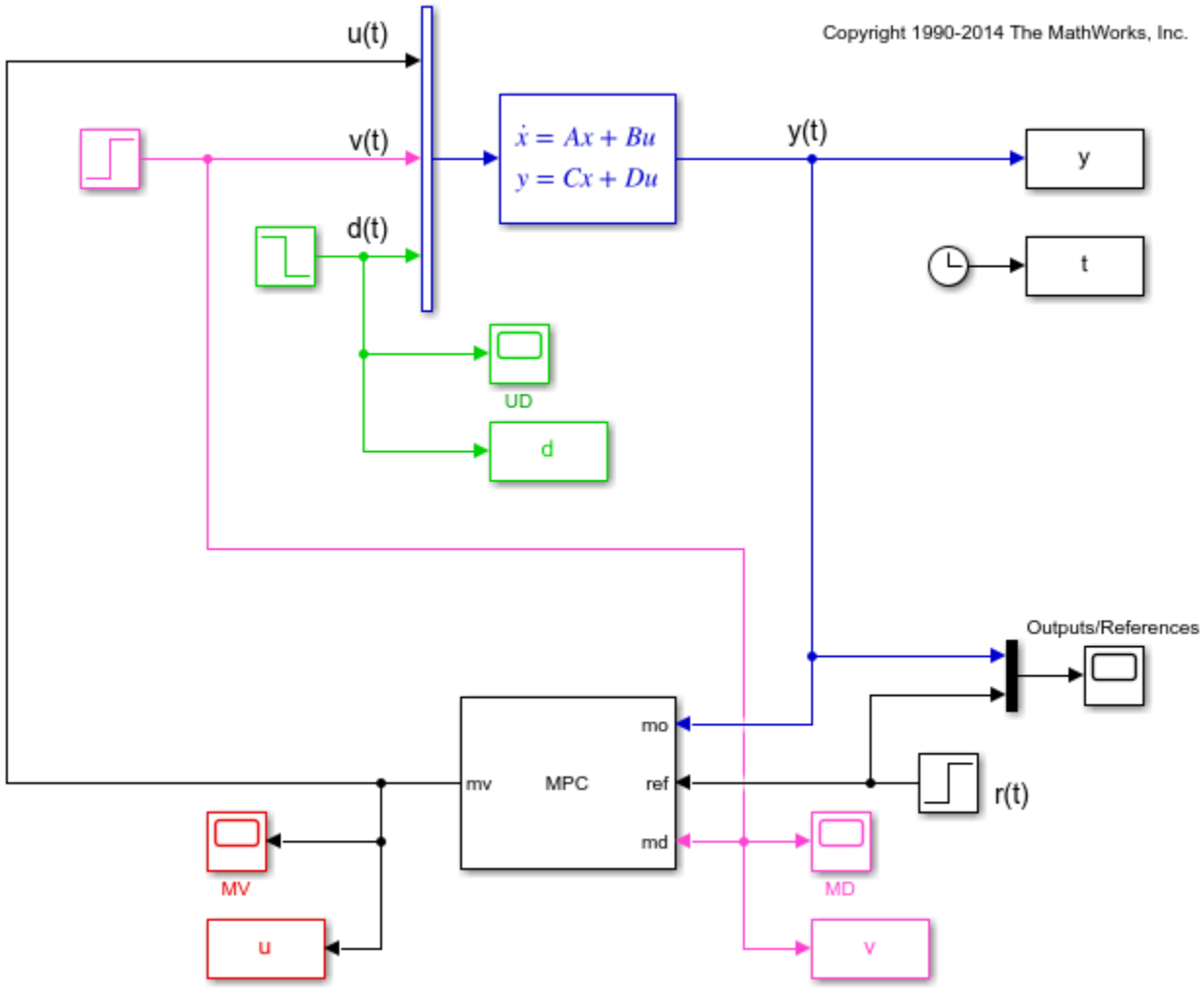
```
-->"Weights.ManipulatedVariables" is empty. Assuming default 0.00000.
-->"Weights.ManipulatedVariablesRate" is empty. Assuming default 0.10000.
-->"Weights.OutputVariables" is empty. Assuming default 1.00000.
```

Obtain the state-space matrices of the continuous-time plant.

```
[A,B,C,D] = ssdata(plantCSS);       % get state-space realization
```

Open the `mpc_miso` Simulink model for closed-loop simulation. The plant model is implemented with a continuous state-space block.

```
open_system('mpc_miso')
```



The plant input signals u(t), v(t), and d(t) represent the manipulated variable, measured input disturbance, and unmeasured input disturbance, respectively, while y(t) is the measured output. The block parameters are the matrices forming the state-space realization of the continuous-time plant, and the initial conditions for the five states. The MPC controller is implemented with an MPC Controller block, which has the workspace MPC object `mpcobj` as a parameter, the manipulated variable as the output, and the measured plant output, reference signal, and measured plant input disturbance, respectively, as inputs. The four Scope blocks plot the five loop signals, which are also saved (except for the reference signal) by four To-Workspace blocks.

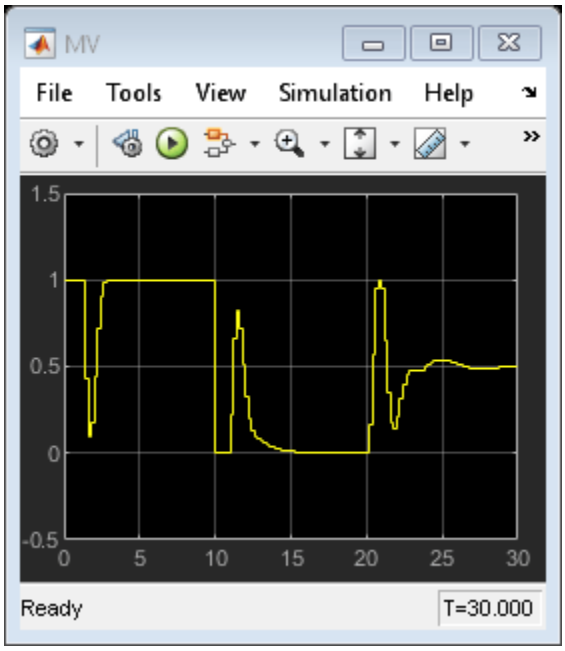
Simulate the closed loop system using the simulink `sim` command. Note that this command (which simulates a Simulink model, and is equivalent to clicking the "Run" button in the model) is different from the `sim` command provided by the MPC toolbox (which instead simulates an MPC controller in a loop with an LTI plant).

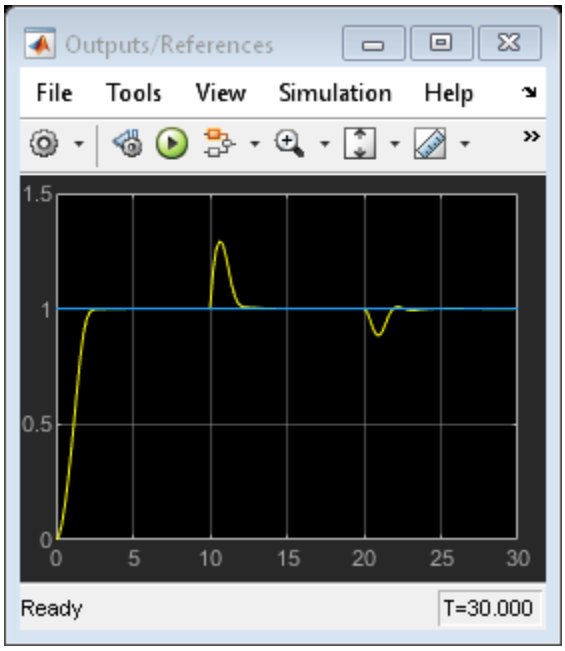
```
sim('mpc_miso')
```

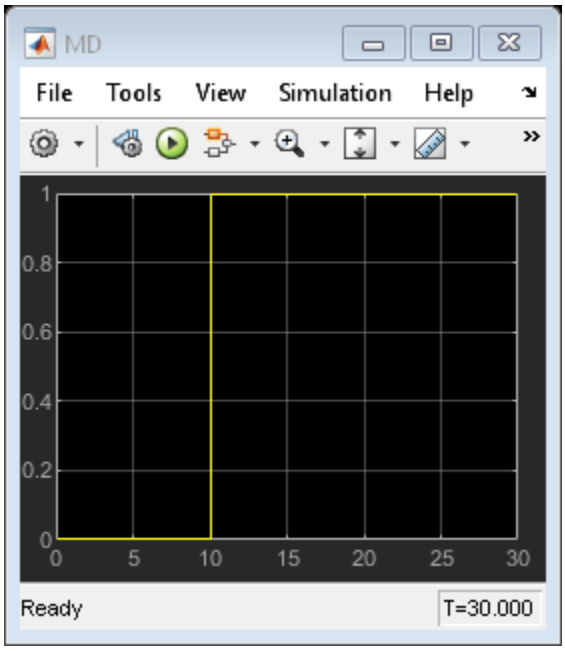
```
   Assuming no disturbance added to measured output #1.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

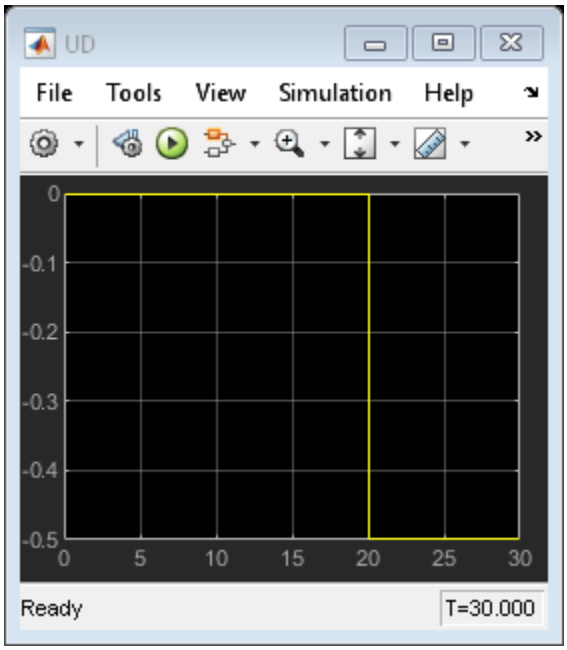
To show the simulation results, open the four Scope windows

```
open_system('mpc_miso/MV')
open_system('mpc_miso/Outputs//References')
open_system('mpc_miso/MD')
open_system('mpc_miso/UD')
```









The plots in the Scope windows are equivalent to the ones in the Simulate Closed-Loop Response Using the `sim` Command and Simulate Closed-Loop Response with Model Mismatch sections, with minor differences due to the fact that in Simulate Closed-Loop Response Using the `sim` Command the unmeasured disturbance signal is zero, and that in Simulate Closed-Loop Response with Model Mismatch you add noise to the plant input and output. Also note that, while the MPC `sim` command internally discretizes any continuous plant model using the ZOH method, Simulink typically uses an integration algorithm (in this example `ode45`) to simulate the closed loop when a continuous-time block is present.

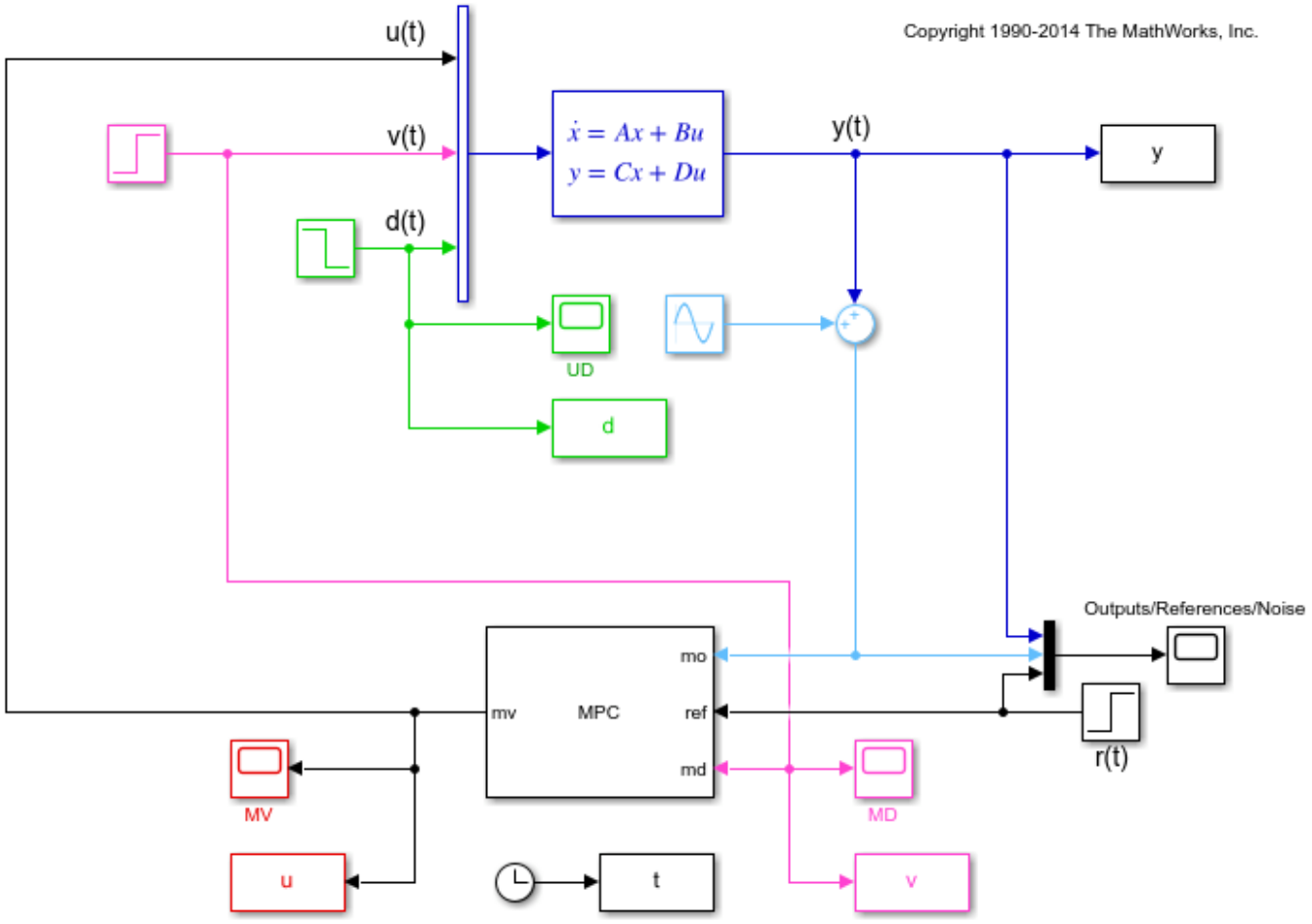
Run Simulation with Sinusoidal Output Noise.

Assume output measurements are affected by a sinusoidal disturbance (a single tone sensor noise) of frequency 0.1 Hz.

```
omega = 2*pi/10;                                % disturbance radial frequency
```

Open the `mpc_misonoise` Simulink model, which is similar to the `mpc_miso` model except for the sinusoidal disturbance added to the measured output. Also, the simulation time is longer and the unmeasured disturbance begins before the measured disturbance.

```
open_system('mpc_misonoise')                    % open new Simulink model
```



Since this noise is expected, you can specify a noise model to help the state estimator ignore it. Doing so improves the disturbance rejection capabilities of the controller.

```
mpcobj.Model.Noise = 0.5*tf(omega^2,[1 0 omega^2]); % measurement noise model
```

Revise the MPC design by specifying a disturbance model on the unmeasured input as a white Gaussian noise with zero mean and variance `0.1`.

```
setindist(mpcobj,tf(0.1));                      % static gain
```

In this case, you cannot have integrators as disturbance model on both the unmeasured input and the output, because this violates state observability. Therefore when you specify a static gain for the input disturbance model, an output disturbance model consisting in a discretized integrator is automatically added to the controller. This output disturbance model helps the controller to reject step-like and slowly varying disturbances at the output.

```
getoutdist(mpcobj)
```

```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->A feedthrough channel in NoiseModel was inserted to prevent problems with estimator design.

ans =

  A =
       x1
   x1   1

  B =
        u1
   x1  0.2

  C =
        x1
   MO1   1

  D =
        u1
   MO1   0

Sample time: 0.2 seconds
Discrete-time state-space model.
```

Large measurement noise can decrease the accuracy of the state estimates. To make the controller less aggressive, and decrease its noise sensitivity, decrease the weight on the output variable tracking.

```
mpcobj.weights = struct('MV',0,'MVRate',0.1,'OV',0.005);    % new weights
```

To give the Kalman filter more time to successfully estimate the states, increase the prediction horizon to 40.

```
mpcobj.predictionhorizon = 40;                  % new prediction horizon
```

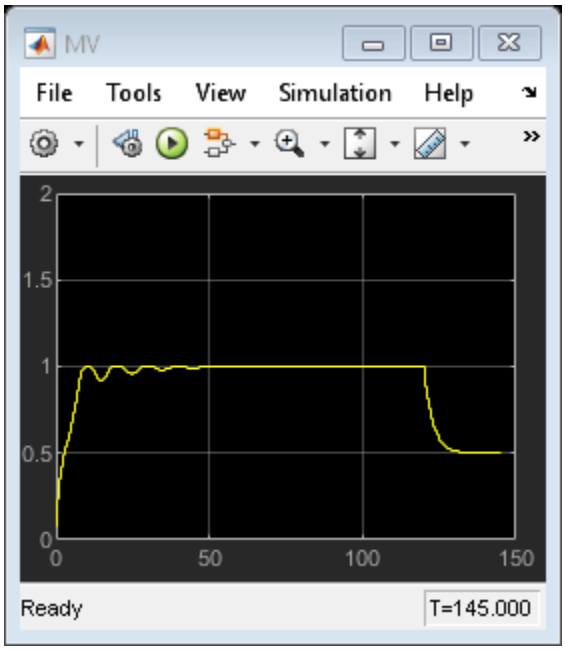
Run the simulation for 145 seconds.

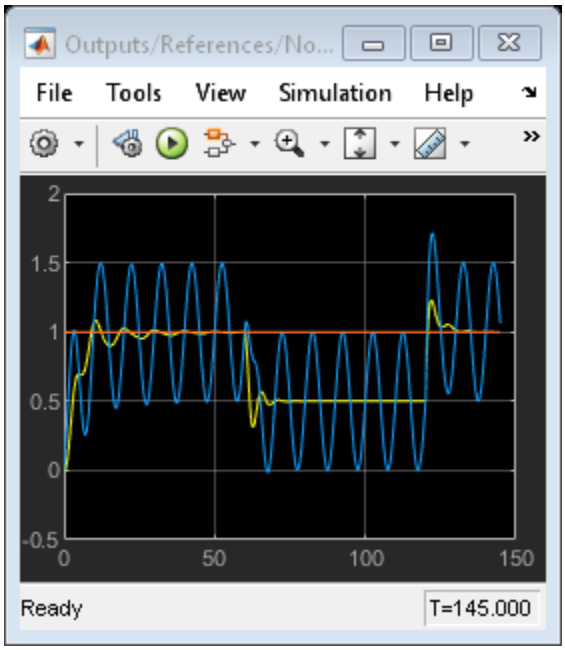
```
sim('mpc_misonoise',145)    % the second argument is the simulation duration
```

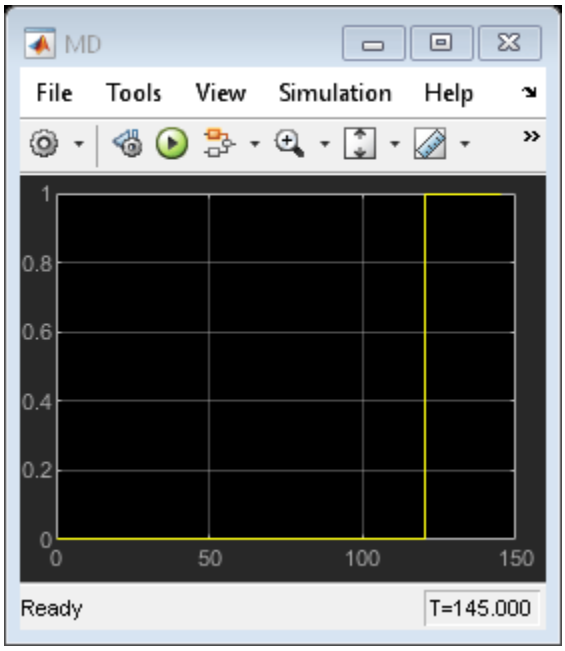
```
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->A feedthrough channel in NoiseModel was inserted to prevent problems with estimator design.
```

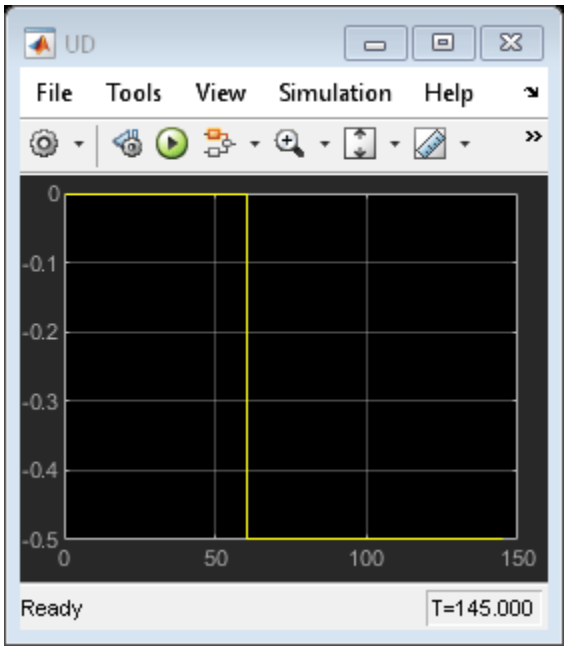
To show the simulation results, open the four Scope windows

```
open_system('mpc_misonoise/MV')
open_system('mpc_misonoise/Outputs//References//Noise')
open_system('mpc_misonoise/MD')
open_system('mpc_misonoise/UD')
```


MV
File Tools View Simulation Help
2
1.5
1
0.5
0
0 50 100 150
Ready
T=145.000


Outputs/References/No...
File Tools View Simulation Help
2
1.5
1
0.5
0
-0.5
0 50 100 150
Ready
T=145.000





The Kalman filter successfully learns to ignore the measurement noise after 50 seconds. The unmeasured and measured disturbances get rejected in a 10 to 20 second timespan. As expected, the manipulated variable stays in the interval between 0 and 1.

```
bdclose all    % close all open Simulink models without saving any change
close all      % close all open figures
```

See Also

Apps
MPC Designer

Functions
`sim` | `review` | `cloffset` | `mpcmove`

Objects
`mpc` | `mpcstate` | `mpcsimopt` | `mpcmoveopt`

Blocks
MPC Controller

Related Examples

- “Model Predictive Control of a Single-Input-Single-Output Plant” on page 3-52
- “Model Predictive Control of a Multi-Input Multi-Output Nonlinear Plant” on page 3-93

More About

- “MPC Signal Types” on page 2-2
- “MPC Prediction Models” on page 2-3
- “What is Model Predictive Control?” on page 1-3

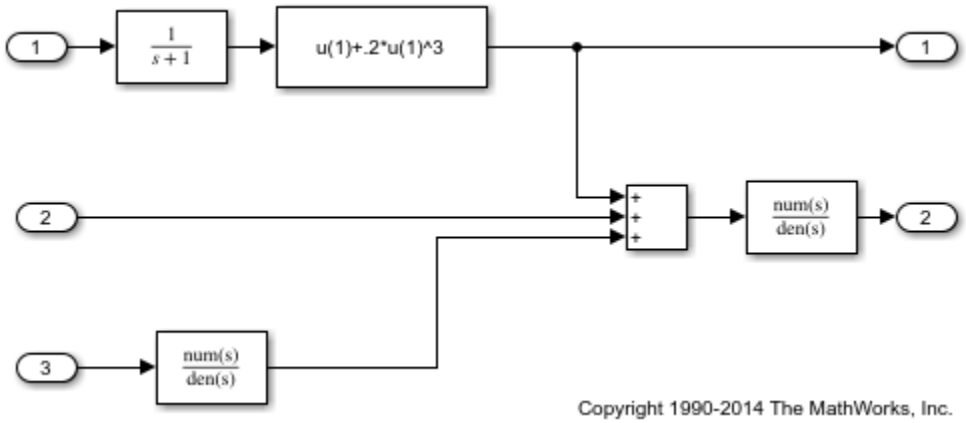
Model Predictive Control of a Multi-Input Multi-Output Nonlinear Plant

This example shows how to design a model predictive controller for a multi-input multi-output nonlinear plant defined in Simulink® and simulate the closed loop. The plant has three manipulated variables and two measured outputs.

Linearize the Nonlinear Plant

The nonlinear plant is implemented in the Simulink model `mpc_nonlinmodel`. Notice the nonlinearity `0.2*u(1)^3` from the first input to the first output.

```
open('mpc_nonlinmodel')
```



Linearize the plant at the default operating conditions (the initial states of the transfer function blocks are all zero) using the `linearize` command from Simulink® Control Design™.

```
plant = linearize('mpc_nonlinmodel');
```

Assign names to I/O variables.

```
plant.InputName = {'Mass Flow';'Heat Flow';'Pressure'};
plant.OutputName = {'Temperature';'Level'};
plant.InputUnit = {'kg/s' 'J/s' 'Pa'};
plant.OutputUnit = {'K' 'm'};
```

Note that since you have not defined any measured or unmeasured disturbance, or any an unmeasured output, when an MPC controller is created based on `plant`, by default all plant inputs are assumed to be manipulated variables and all plant outputs are assumed to be measured outputs.

Design the MPC Controller

Create the controller object with sampling period, prediction and control horizons of 0.2 sec, 5 steps, and 2 moves, respectively;

```
mpcobj = mpc(plant,0.2,5,2);
```

```
-->"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 hard constraints on the manipulated variable.

```
mpcobj.MV = struct('Min',{-3;-2;-2},'Max',{3;2;2},'RateMin',{-1000;-1000;-1000});
```

Define weights on manipulated variables and output signals.

```
mpcobj.Weights = struct('MV',[0 0 0],'MVRate',[.1 .1 .1],'OV',[1 1]);
```

Display the MPC object to review its properties.

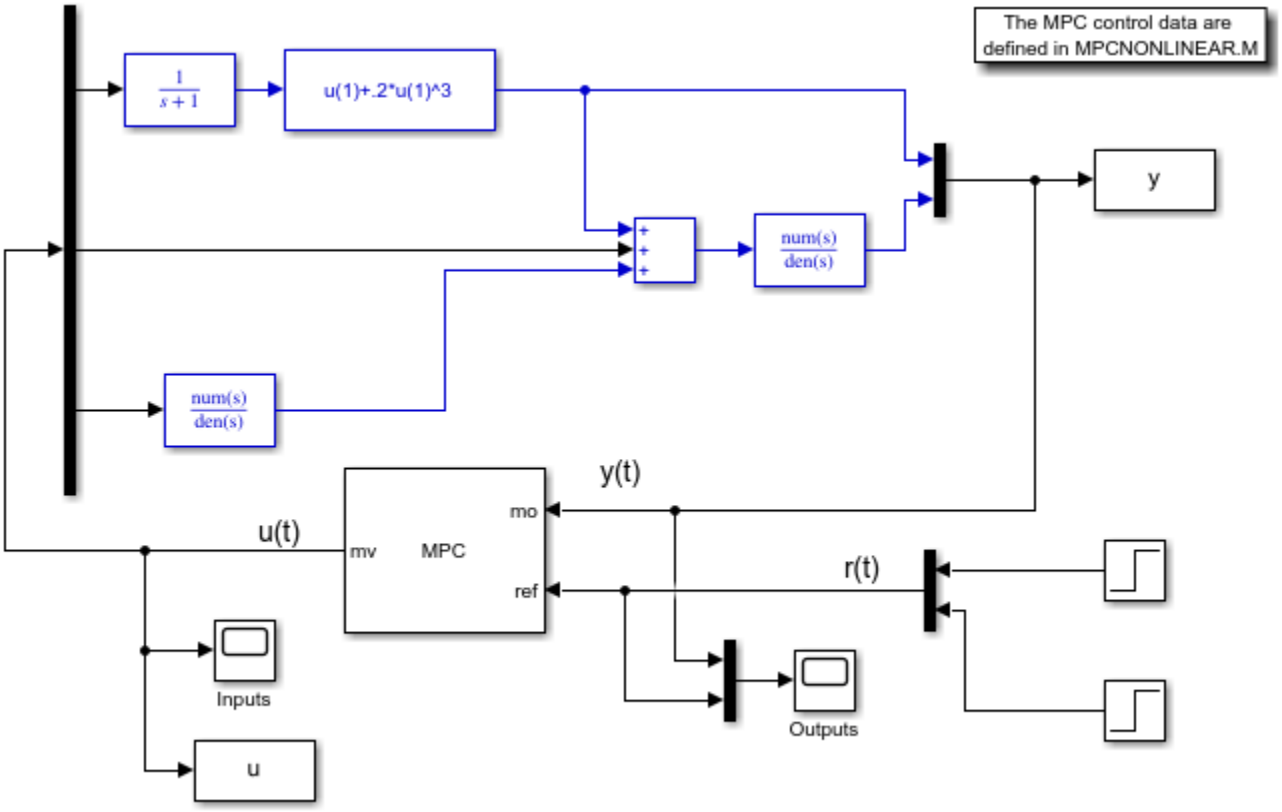
```
mpcobj
```

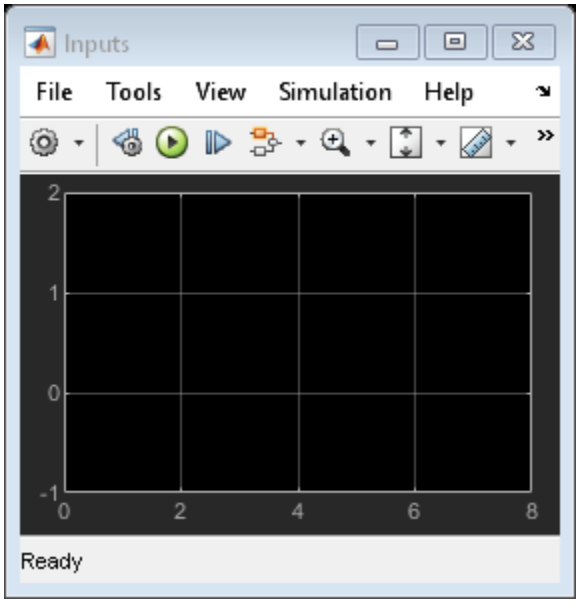
```
MPC object (created on 03-Mar-2023 21:37:07):
---------------------------------------------
Sampling time:      0.2 (seconds)
Prediction Horizon: 5
Control Horizon:    2

Plant Model:
                                    --------------
      3  manipulated variable(s)   -->|  5 states  |
                                     |             |-->  2 measured output(s)
      0  measured disturbance(s)   -->|  3 inputs  |
                                     |             |-->  0 unmeasured output(s)
      0  unmeasured disturbance(s) -->|  2 outputs |
                                     --------------
Disturbance and Noise Models:
        Output disturbance model: default (type "getoutdist(mpcobj)" for details)
         Measurement noise model: default (unity gain after scaling)

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

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

Constraints:
 -3 <= Mass Flow (kg/s) <= 3, -1000 <= Mass Flow/rate (kg/s) <= Inf, Temperature (K) is unconstra
  -2 <= Heat Flow (J/s) <= 2,  -1000 <= Heat Flow/rate (J/s) <= Inf,       Level (m) is unconstra
    -2 <= Pressure (Pa) <= 2,    -1000 <= Pressure/rate (Pa) <= Inf

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

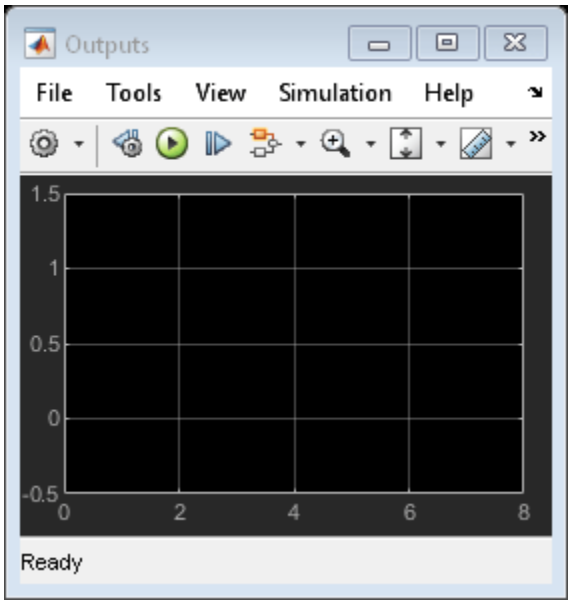
Simulate the Closed Loop Using Simulink

Open the pre-existing Simulink model for the closed-loop simulation. The plant model is identical to the one used for linearization, while the MPC controller is implemented with an MPC controller block, which has the workspace MPC object `mpcobj` as parameter. The reference for the first output is a step signal rising from zero to one for t=0, as soon as the simulation starts. The reference for the second output

```
mdl1 = 'mpc_nonlinear';
open_system(mdl1)
```



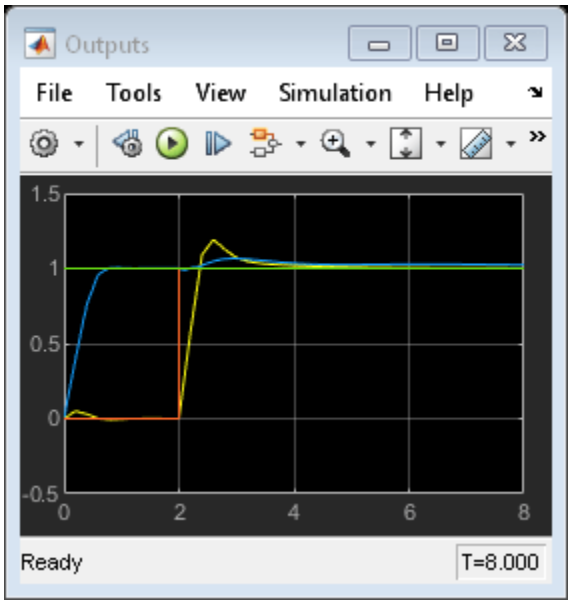




Run the closed loop simulation.

```
sim(mdl1)
```

```
-->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.
```



Despite the presence of the nonlinearity, both outputs track their references well after a few seconds, while, as expected, the manipulated variables stay within the preset hard constraints.

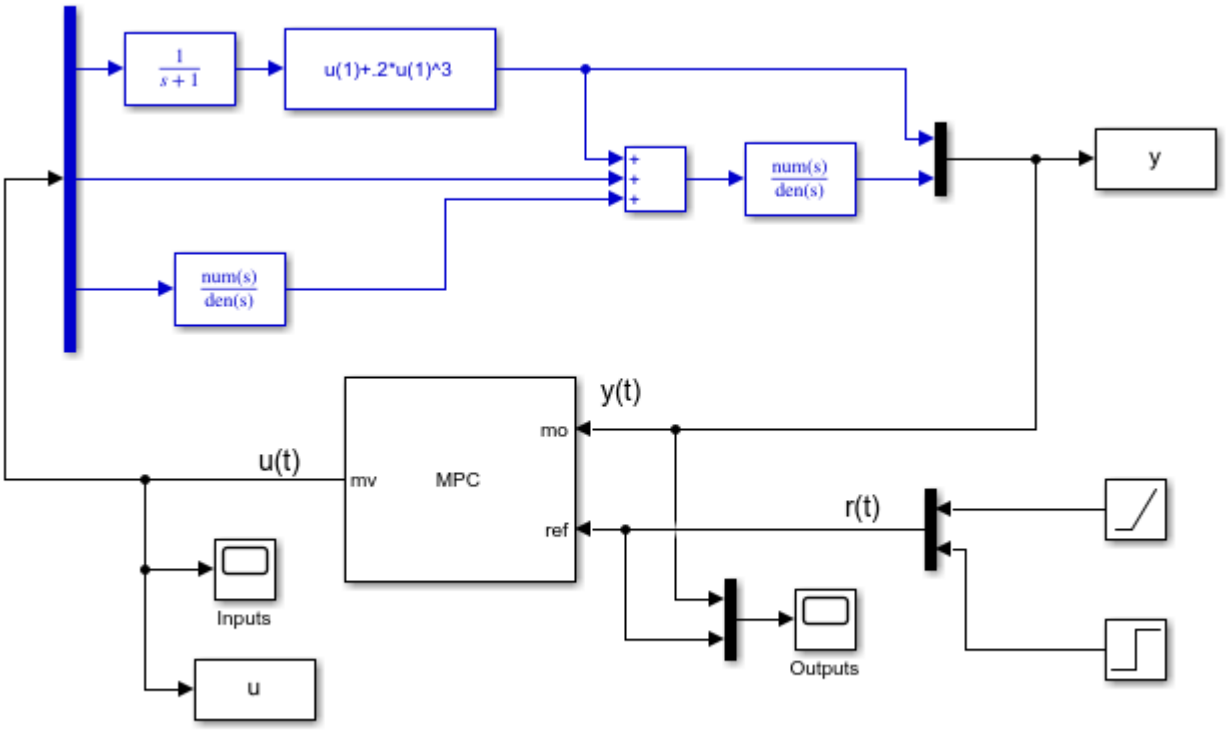
Modify MPC Design to Track Ramp Signals

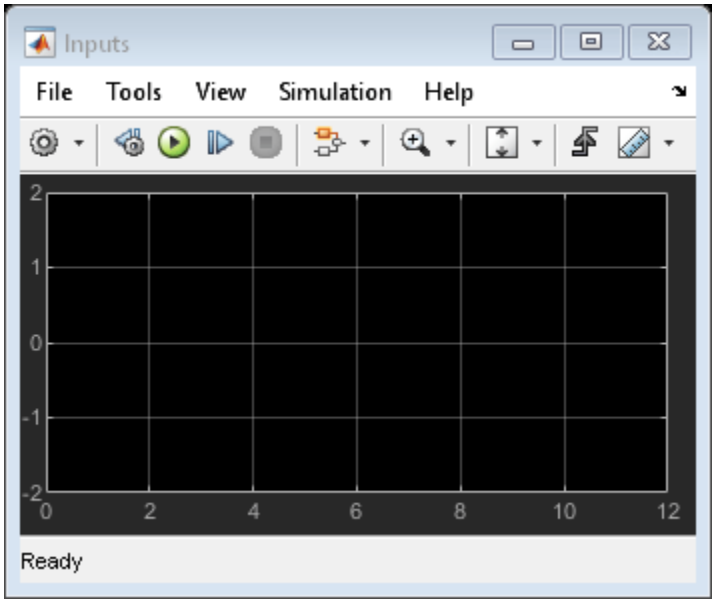
In order to both track a ramp while compensating for the nonlinearity, define a disturbance model on both outputs as a triple integrator (without the nonlinearity a double integrator would suffice).

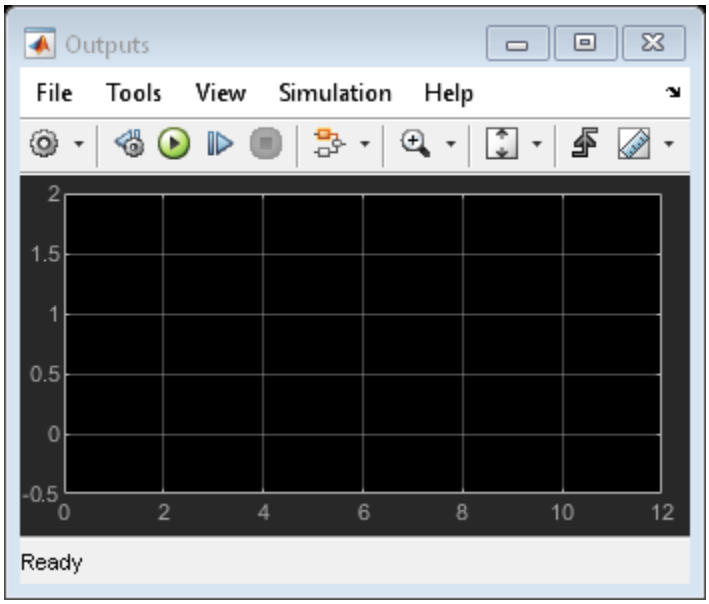
```
outdistmodel = tf({1 0;0 1},{[1 0 0 0],1;1,[1 0 0 0]});
setoutdist(mpcobj,'model',outdistmodel);
```

Open the pre-existing Simulink model for the closed-loop simulation. It is identical to the previous closed loop model, except for the fact that the reference for the first plant output is no longer a step but a ramp signal that rises with slope of 0.2 after 3 seconds.

```
mdl2 = 'mpc_nonlinear_setoutdist';
open_system(mdl2)
```



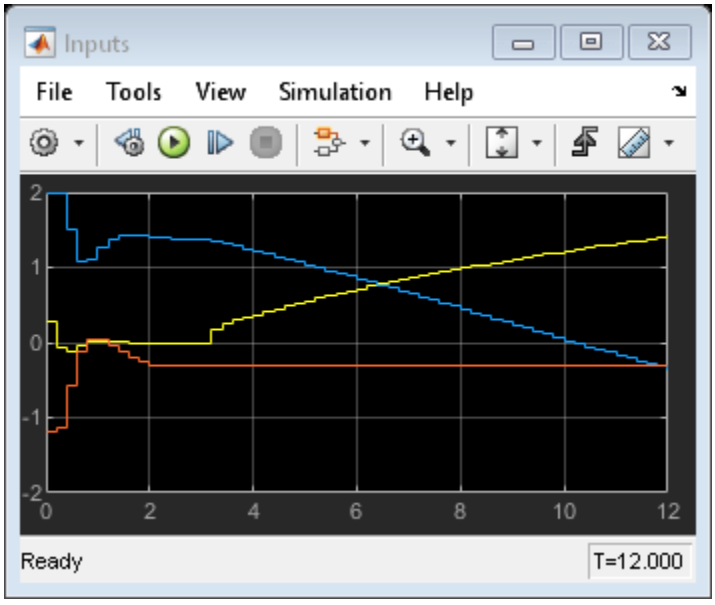


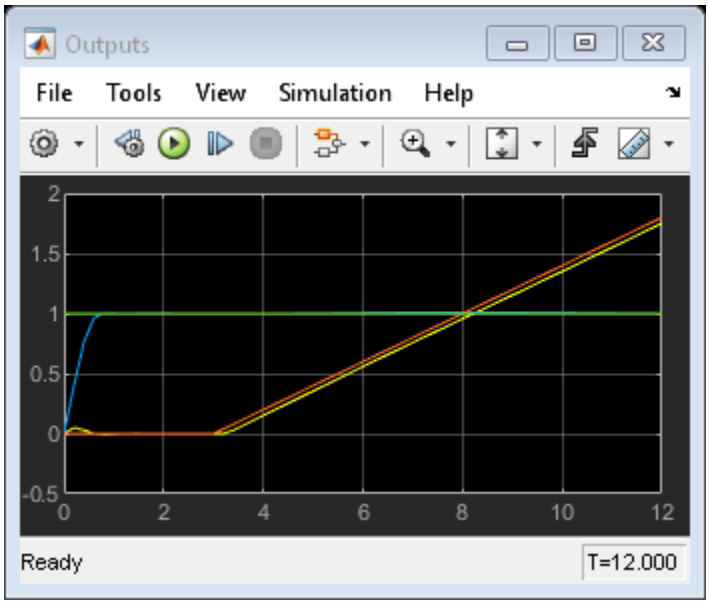


Run the closed loop simulation for 12 seconds.

```
sim(mdl2,12)
```

```
-->Converting model to discrete time.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```





Simulate without Constraints

When the constraints are not active, the MPC controller behaves like a linear controller. Simulate two versions of an unconstrained MPC controller in closed loop to illustrate this fact.

First, remove the constraints from `mcpobj`.

```
mpcobj.MV = [];
```

Then reset the output disturbance model to default, (this is only done to get a simpler version of a linear MPC controller in the next step).

```
setoutdist(mpcobj,'integrators');
```

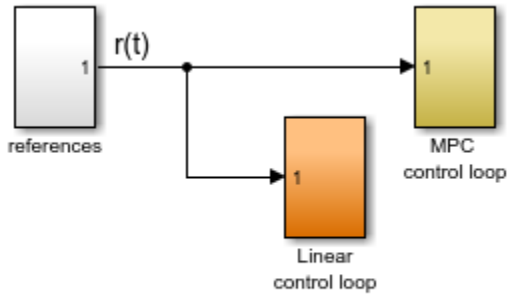
Convert the unconstrained MPC controller to a linear time invariant (LTI) state space dynamical system, having the vector `[ym;r]` as input, where `ym` is the vector of measured output signals (at a given step), and `r` is the vector of output references (at the same given step).

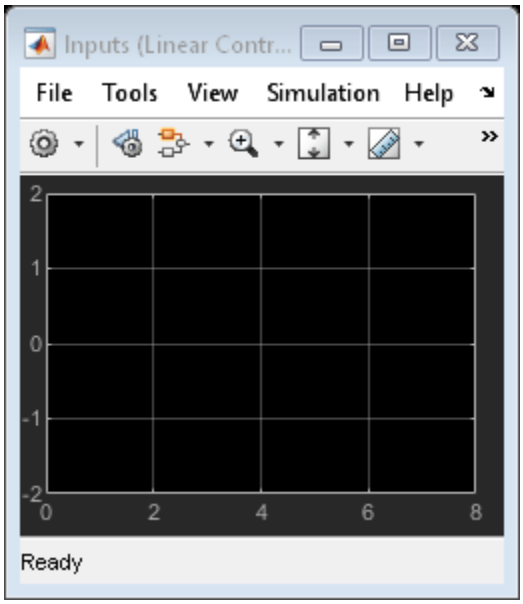
```
LTI = ss(mpcobj,'r');          % use reference as additional input signal
```

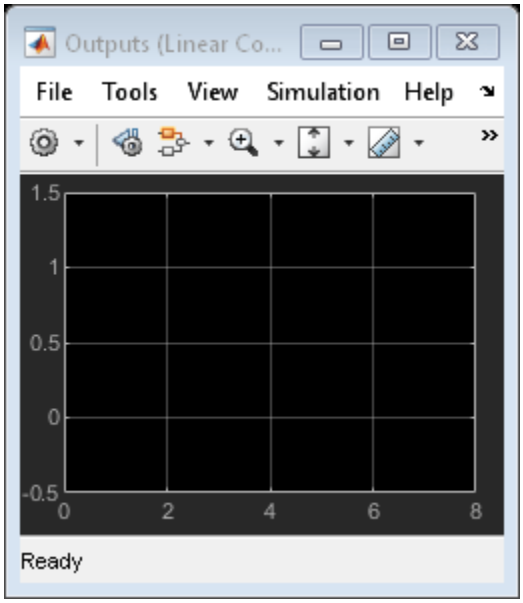
```
-->Converting model to discrete time.
-->Assuming output disturbance added to measured output #1 is integrated white noise.
-->Assuming output disturbance added to measured output #2 is integrated white noise.
-->"Model.Noise" is empty. Assuming white noise on each measured output.
```

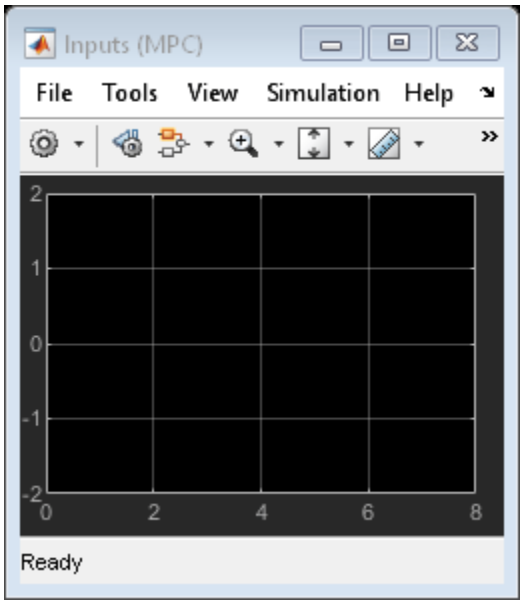
Open the pre-existing Simulink model for the closed-loop simulation. The "references" block contains two step signals (acting after 4 and 0 seconds, respectively) that are used as a reference. The "MPC control loop" block is equivalent to the first closed loop, except for the fact that the reference signals are supplied to it as input. The "Linear control loop" block is equivalent to the "MPC control loop" block except for the fact that the controller is an LTI block having the workspace `ss` object `LTI` as parameter.

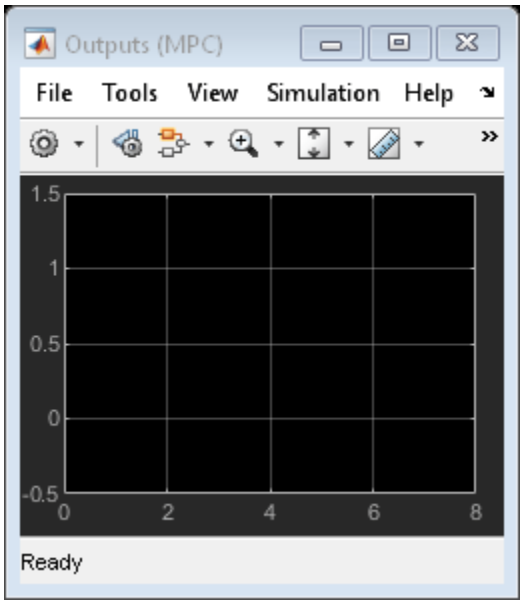
```
refs = [1;1];                   % set values for step signal references
mdl3 = 'mpc_nonlinear_ss';
open_system(mdl3)
```





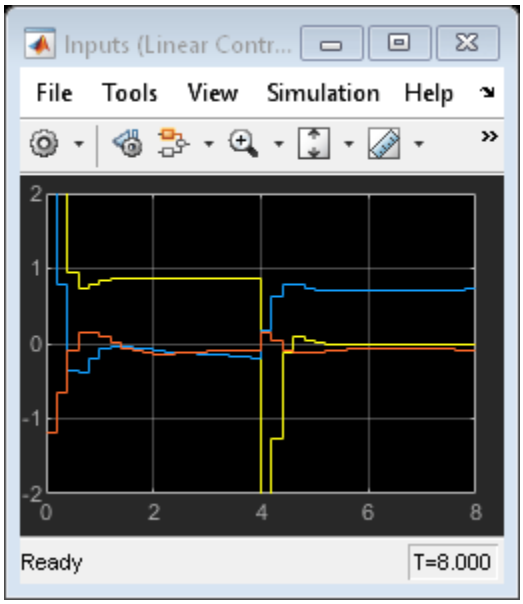


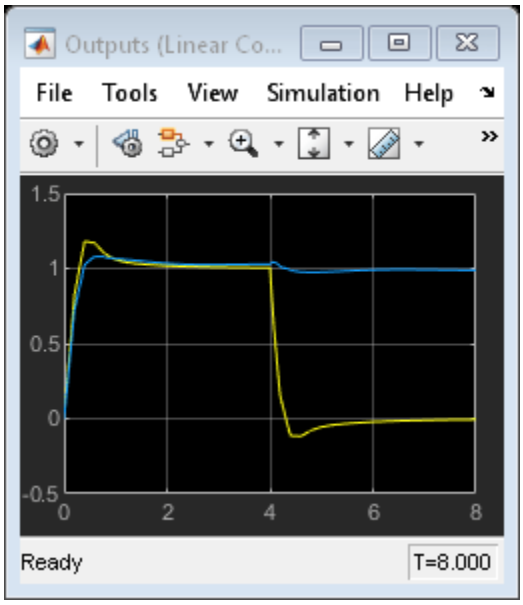


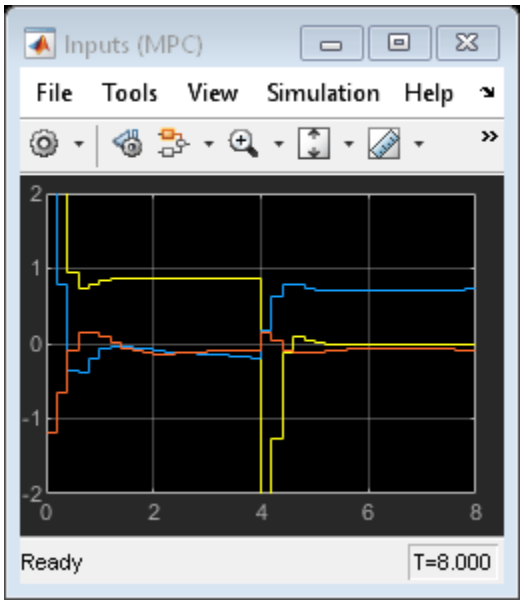


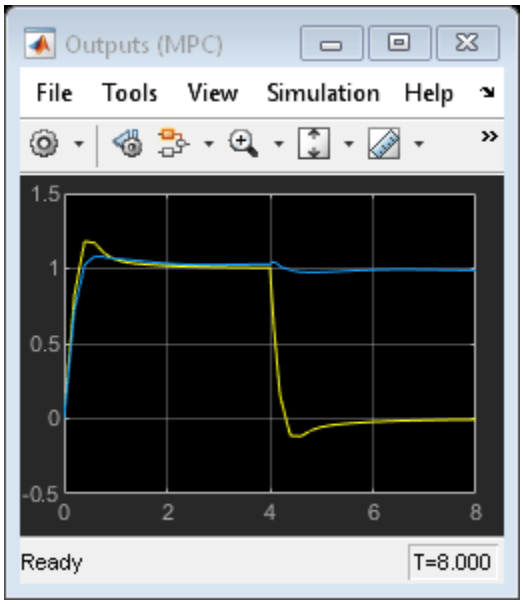
Run the closed loop simulation for 12 seconds.

```
sim(mdl3)
```


Inputs (Linear Contr...
File Tools View Simulation Help
2
1
0
-1
-2
0 2 4 6 8
Ready
T=8.000


Outputs (Linear Co...
File Tools View Simulation Help
1.5
1
0.5
0
-0.5
0 2 4 6 8
Ready
T=8.000





The inputs and outputs signals look identical for both loops. Also note that the manipulated variable are no longer bounded by the previous constraints.

Compare Simulation Results

```
fprintf('Compare output trajectories: ||ympc-ylin|| = %g\n',norm(ympc-ylin));
disp('The MPC controller and the linear controller produce the same closed-loop trajectories.');
```

```
Compare output trajectories: ||ympc-ylin|| = 1.16677e-14
The MPC controller and the linear controller produce the same closed-loop trajectories.
```

As expected, there is only a negligible difference due to numerical errors.

See Also

Apps
MPC Designer

Functions
`linearize`

Objects
`mpc`

Blocks
MPC Controller

Related Examples

- “Model Predictive Control of a Single-Input-Single-Output Plant” on page 3-52
- “Model Predictive Control of Multi-Input Single-Output Plant” on page 3-56

More About

- “What is Model Predictive Control?” on page 1-3