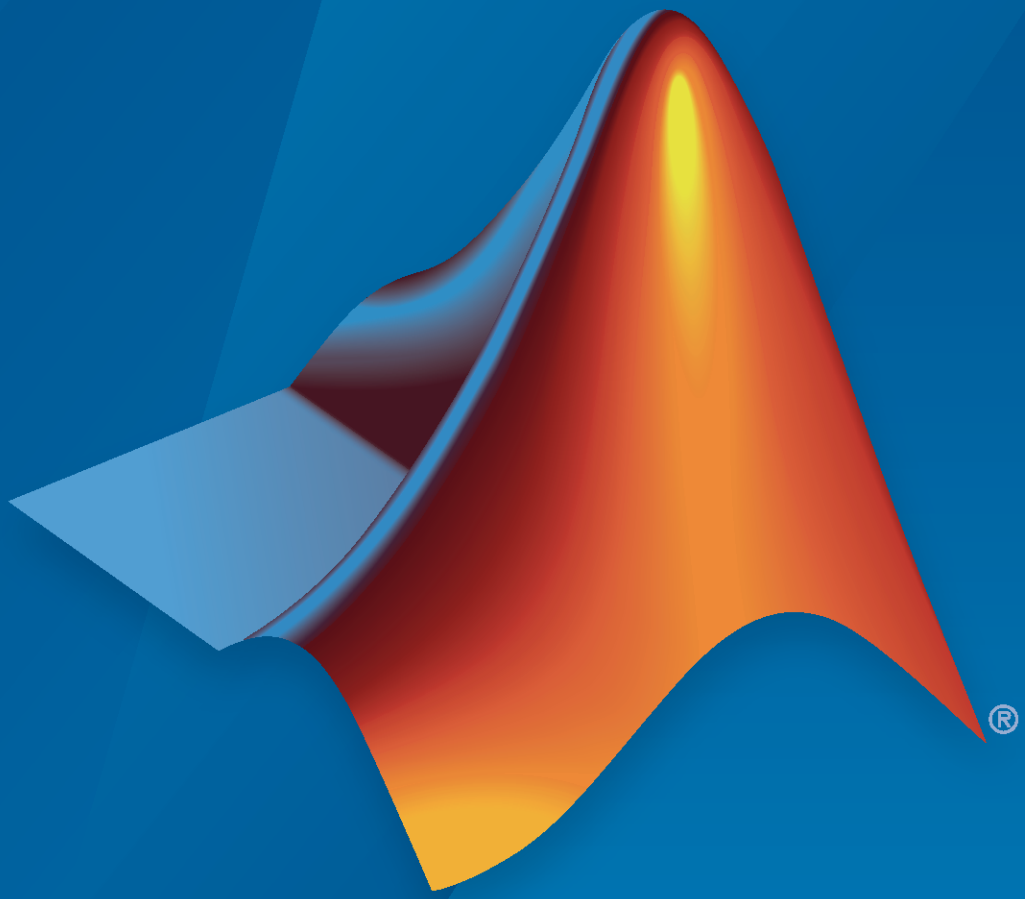
# Control System Toolbox™ Release Notes



MATLAB®

MathWorks®

# How to Contact MathWorks

Latest news: `www.mathworks.com`

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

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

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

Phone: 508-647-7000

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

*Control System Toolbox™ Release Notes*

# Contents

## R2020b



## R2020a



## R2019b

## R2019a



## R2018b



## R2018a



## R2017b

## R2017a



## R2016b

## R2016a



## R2015b

## R2015a



## R2014b

## R2014a



## R2013b



## R2013a



## R2012b

## R2012a



## R2011b



## R2011a

## R2010b



## R2010a



## R2009b

## R2009a



## R2008b



## R2008a



## R2007b



## R2007a

## R2006b



## R2006a



## R14SP3



## R14SP2

# R2020b

## sparss and mechss: Support for sparse state-space models

Use the new `sparss` and `mechss` objects to represent first-order and second-order sparse state-space models, respectively. Using the sparse representation for state-space models increases computational efficiency while reducing the amount of memory required for data storage. For more information, see the `sparss` and `mechss` reference pages.

You can perform time-domain and frequency-domain analysis on `sparss` and `mechss` model objects with functions listed in “Time and Frequency Domain Analysis”. Signal-based interconnections are also supported for sparse model objects. Use the functions listed in “Model Interconnection” to interface signals between sparse models and with other LTI models. For more information, see “Sparse Model Basics”. For an example, see “Transient Modeling and Linear Analysis of a Cantilever Beam”.

Additionally, you can:

- Access the sparse matrices and sample time of the sparse model objects using `sparssdata` and `mechssdata`.
- Obtain a summary of how the state vector is partitioned into components, interfaces, and signals in your `sparss` and `mechss` model objects using `showStateInfo`.
- Sort states based on state partition using `xsort`.
- Specify the physical interface between two sparse models or between subcomponents of a `mechss` object using the `interface` command. For an example, see “Rigid Assembly of Model Components”.
- Use the Sparse Second Order block to represent your second-order sparse state-space model in Simulink®. For an example, see “Linearize Simulink Model to a Sparse Second-Order Model Object”.
- Linearize and obtain a sparse model from a Simulink model when a Descriptor State-Space or a Sparse Second Order block is present. For more information, see “Sparse Model Basics”. For an example, see “Linearize Simulink Model to a Sparse Second-Order Model Object”.
- Convert between continuous-time and discrete-time and resample sparse models using `c2d`, `d2c` and `d2d`. For available methods, see “Sparse Model Basics”.

## New Example: Tune Phase-Locked Loop Using Loop-Shaping Design

The new example, “Tune Phase-Locked Loop Using Loop-Shaping Design”, shows how to tune the components of a loop filter to improve the loop bandwidth of a phase-locked loop (PLL) system, using `systune` to achieve the specified loop shape. The example uses the Mixed-Signal Blockset™ library to model the PLL system.

## Functionality being removed or changed

### Extended and Unscented Kalman Filter Algorithms: Numerical changes

*Behavior change*

Numerical improvements in the algorithms used by the Extended Kalman Filter and Unscented Kalman Filter blocks and the `extendedKalmanFilter` and `unscentedKalmanFilter` functions might produce results that are different from the results you obtained using previous versions.

# R2020a

**Version: 10.8**

**New Features**

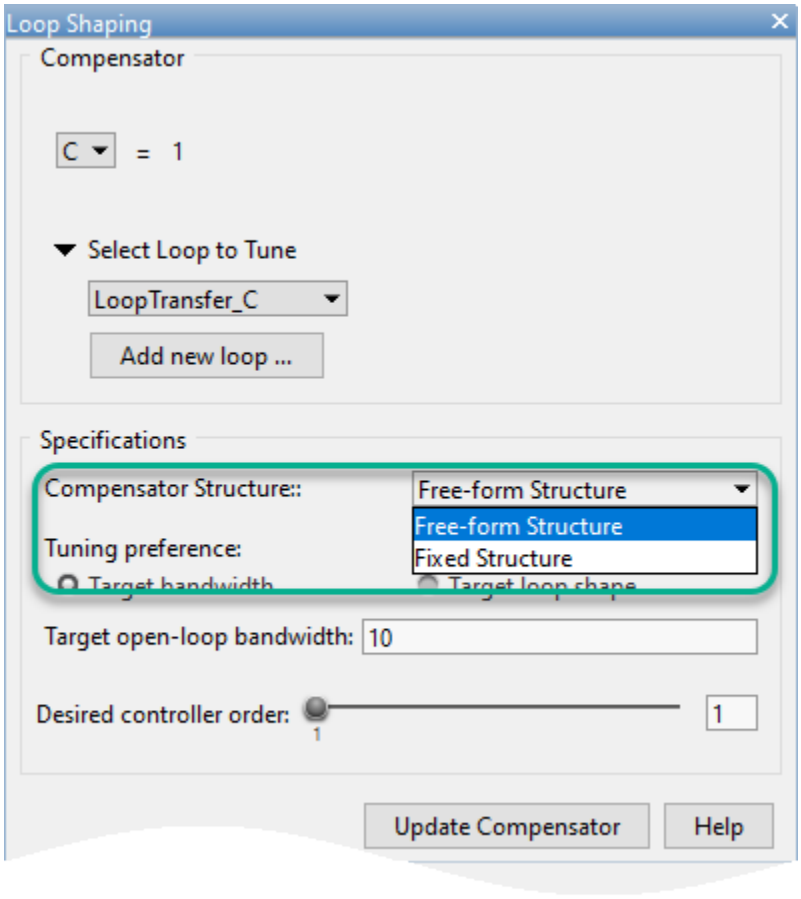
**Bug Fixes**

## Control System Designer: Perform automated loop shaping without a Robust Control Toolbox license

You can now perform automated loop shaping in the Control System Designer without the requirement of a license for Robust Control Toolbox™. Control System Designer uses `looptune` instead of `loopsyn` in the absence of a Robust Control Toolbox license.

If you already have a Robust Control Toolbox license, you can choose one of the following options from the **Compensator Structure** dropdown menu:

- `Free-form Structure` — Uses `loopsyn` to perform the automated loop shaping with the option to specify your controller order preference using the **Desired controller order** slider.
- `Fixed Structure` — Uses `looptune` to perform the automated loop shaping without the option to specify your controller order preference. When you select `Fixed Structure`, Control System Designer will use the supplied controller order to perform the automated loop shaping.




For more information, see Design Compensator Using Automated Tuning Methods.

## frdfun: Utility to apply a function across all frequencies of an frd model

Use `frdfun` to apply a function across all frequencies of a frequency-response data model `frd`. The function to be used on the `frd` model must take a single matrix and return a scalar, vector, or matrix of fixed size across the frequencies.

For more information, see `frdfun`.

## zgrid: Specify Sample Time to Plot True Frequencies on the Grid

`zgrid` now includes a new optional argument to specify the sample time `T`. You can use the syntax `zgrid(...,T)` to plot true frequencies on the z-plane grid.

For more information, see `zgrid`.

# R2019b

**Version: 10.7**

**New Features**

**Bug Fixes**

## Live Editor Tasks: Perform model transformation and control design tasks interactively and generate MATLAB code in a live script

Use new Live Editor tasks to perform model reduction, model rate conversion, and PID controller tuning without writing code. The tasks generate plots that let you interactively explore the effects of changing parameter values and options. The tasks also automatically generate code that becomes part of your live script.

In R2019b, Control System Toolbox includes three tasks:

- **Reduce Model Order** — Reduce complexity of linear time-invariant (LTI) models
- **Convert Model Rate** — Convert LTI models between continuous time and discrete time; resample discrete-time models
- **Tune PID Controller** — Tune PID controller gains for LTI plants

To use tasks in the Live Editor, on the **Live Editor** tab, in the **Task** menu, select a task. Alternatively, in a code block in a live script, begin typing the task name and select the task from the suggested command completions. For more information about Live Editor tasks generally, see Add Interactive Tasks to a Live Script (MATLAB).

## Residuals for Extended and Unscented Kalman Filters: Calculate residuals and residual covariances of filter predictions

The new function `residual` computes residuals for extended Kalman filter (EKF) and unscented Kalman filter (UKF) objects. These residuals represent the error between the filter predictions and the measurements. Use `residual` to help validate filter performance.

For more information, see `residual`.

For more information on extended and unscented Kalman filter objects, see `extendedKalmanFilter` and `unscentedKalmanFilter`.

## ngrid, sgrid, and zgrid: Specify target axes as Axes and UIAxes to create apps in App Designer

You can now use an `Axes` or `UIAxes` object with `ngrid`, `sgrid`, and `zgrid` to specify target axes when creating apps in App Designer.

For more information, see `ngrid`, `sgrid` and `zgrid`.

## Nyquist Plots: Programmatically Zoom on Critical Point

You can now use the `zoomcp` command to zoom programmatically on the critical point of a Nyquist plot that you create with `nyquistplot`. Previously, you could zoom the plot to the critical point by right-clicking and selecting **Zoom on (-1,0)**. Now, for a plot with handle `h`, entering the command `zoomcp(h)` achieves the same result as the right-click option. For an example, see the `nyquistplot` reference page.

# R2019a

## getPIDLoopResponse: Obtain closed-loop and open-loop responses of plant with PID controller

The new `getPIDLoopResponse` command generates responses for a control system formed by a PID controller and plant model. The function returns the closed-loop, open-loop, controller action, or disturbance response that you specify. Previously, you had to construct each response transfer function manually, using interconnection commands such as `feedback(G*C,1)`. For more information, see `getPIDLoopResponse`.

## icare and idare Commands: Solve continuous-time and discrete-time Riccati equations

Use the new `icare` and `idare` commands to solve continuous-time and discrete-time Riccati equations. `icare` replaces `care` and `gcare` for solving continuous-time Riccati equations while, `idare` replaces `dare` and `gdare` for solving discrete-time Riccati equations. The new commands have improved accuracy through better scaling and the computation of gain `K` is more accurate when `R` is ill-conditioned.

For more information, see `icare` and `idare`.

## Compatibility Considerations

- `icare` replaces `care` and `gcare` for solving continuous-time Riccati equations. The `care` and `gcare` commands are not recommended. For more information, see “care and gcare are not recommended” on page 4-2.
- `idare` replaces `dare` and `gdare` for solving discrete-time Riccati equations. The `dare` and `gdare` commands are not recommended. For more information, see “dare and gdare are not recommended” on page 4-3.

## Functionality being removed or changed

**care and gcare are not recommended**
*Still runs*

The `care` and `gcare` commands are not recommended. Use `icare` to solve continuous-time implicit Riccati equations instead. This approach has improved accuracy through better scaling and the computation of `K` is more accurate when `R` is ill-conditioned relative to `care` and `gcare`. Furthermore, `icare` includes an optional `info` structure to gather the implicit solution data of the Riccati equation.

The following table shows some typical uses of `care` and `gcare`, and how to update your code to use `icare` instead.

| Not Recommended | Recommended |
| --- | --- |
| • `[X,L,G] = care(A,B,Q,R,S,E)`<br>• `[X,L] = gcare(H,J,NS)` | `[X,K,L] = icare(A,B,Q,R,S,E,G)` computes the stabilizing solution `X`, the state-feedback gain `K` and the closed-loop eigenvalues `L` of the continuous-time algebraic Riccati equation. For more information, see `icare`. |

| Not Recommended | Recommended |
| --- | --- |
| • `[X,L,G,report] = care(A,B,Q,R,S,E)`<br>• `[X,L,report] = gcare(H,J,NS)` | `[X,K,L,info] = icare(A,B,Q,R,S,E,G)` computes the stabilizing solution `X`, the state-feedback gain `K`, the closed-loop eigenvalues `L` of the continuous-time algebraic Riccati equation. The `info` structure contains the implicit solution data. For more information, see `icare`. |

There are no plans to remove `care` and `gcare` at this time.

**dare and gdare are not recommended**
*Still runs*

The `dare` and `gdare` commands are not recommended. Use `idare` to solve discrete-time implicit Riccati equations instead. This approach has improved accuracy through better scaling and the computation of `K` is more accurate when `R` is ill-conditioned relative to `dare` and `gdare`. Furthermore, `idare` includes an optional `info` structure to gather the implicit solution data of the Riccati equation.

The following table shows some typical uses of `dare` and `gdare`, and how to update your code to use `idare` instead.

| Not Recommended | Recommended |
| --- | --- |
| • `[X,L,G] = dare(A,B,Q,R,S,E)`<br>• `[X,L] = gdare(H,J,NS)` | `[X,K,L] = idare(A,B,Q,R,S,E)` computes the stabilizing solution `X`, the state-feedback gain `K` and the closed-loop eigenvalues `L` of the discrete-time algebraic Riccati equation. For more information, see `idare`. |
| • `[X,L,G,report] = dare(A,B,Q,R,S,E)`<br>• `[X,L,report] = gdare(H,J,NS)` | `[X,K,L,info] = idare(A,B,Q,R,S,E)` computes the stabilizing solution `X`, the state-feedback gain `K`, the closed-loop eigenvalues `L` of the discrete-time algebraic Riccati equation. The `info` structure contains the implicit solution data. For more information, see `idare`. |

There are no plans to remove `dare` and `gdare` at this time.

# R2018b

**Version: 10.5**

**New Features**

**Bug Fixes**

## allmargin Function: New MIMO syntax for loop-at-a-time analysis

You can now use `allmargin` to compute loop-at-a-time classical stability margins of MIMO systems. Previously, `allmargin` computed stability margins for SISO systems only.

For more information, see the `allmargin` reference page.

# R2018a

**Version: 10.4**

**New Features**

**Bug Fixes**

## Particle Filter Simulink Block: Estimate states of nonlinear systems for online tracking and control system design

Perform state estimation for arbitrary nonlinear models using the new Particle Filter block in Simulink. Particle filters are flexible in comparison to Kalman filters, that is, they can also perform state estimation for nonlinear systems with non-Gaussian distributions.

Particle Filter block uses particles and sensor data to estimate the posterior distribution of the current state. The filter predicts the states using the nonlinear state transition function. Then, it corrects the estimate based on sensor data and measurement likelihood model. You can specify a fixed number of particles to use, a fixed number of state variables to estimate, and your state estimation method.

You can find the Particle Filter block in the **Control System Toolbox > State Estimation** block library in Simulink.

You can use Simulink Coder™ to deploy particle filters with multiple measurement models and fixed-size arrays for your application.

For more information on the Particle Filter block, see Particle Filter. For more information on the detailed workflow, see Parameter and State Estimation in Simulink Using Particle Filter Block.

## c2d Function: Convert models to discrete time using least-squares optimization

You can now convert continuous-time dynamic system models to discrete time using a new least-squares optimization method. This algorithm minimizes the error between the frequency responses of the continuous-time and discrete-time systems up to the Nyquist frequency. This method is useful when you want to capture fast system dynamics but must use a larger sample time, for example, when computational resources are limited.

To convert a model using this approach, specify the discretization method as `'least-squares'`.

```
discreteModel = c2d(contModel,Ts,'least-squares');
```

Alternatively, you can create a `c2dOptions` option set, and set the `Method` property to `'least-squares'`. You can then use this option set with the `c2d` function.

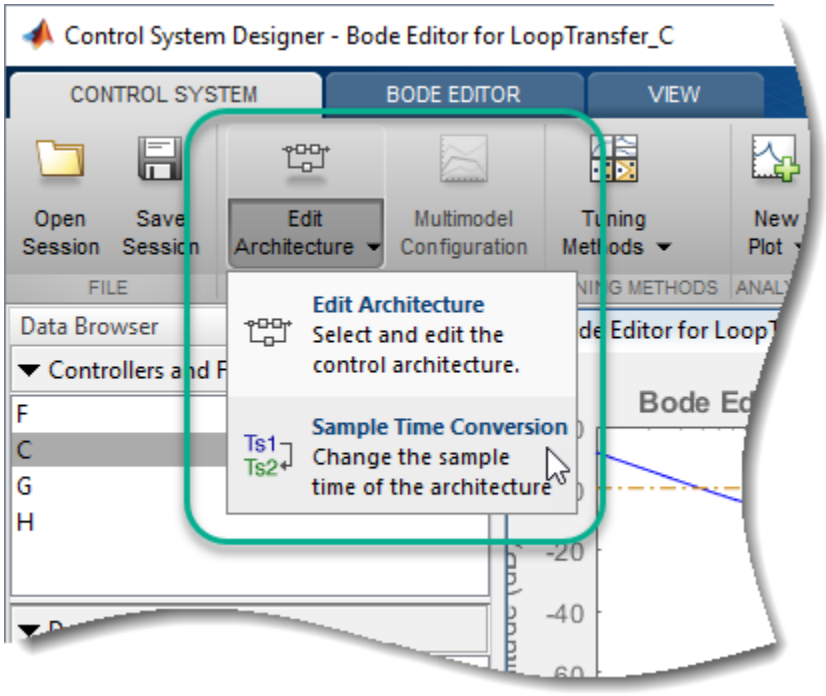
```
options = c2dOptions('Method','least-squares');
discreteModel = c2d(contModel,Ts,options);
```

This conversion method supports only SISO models.

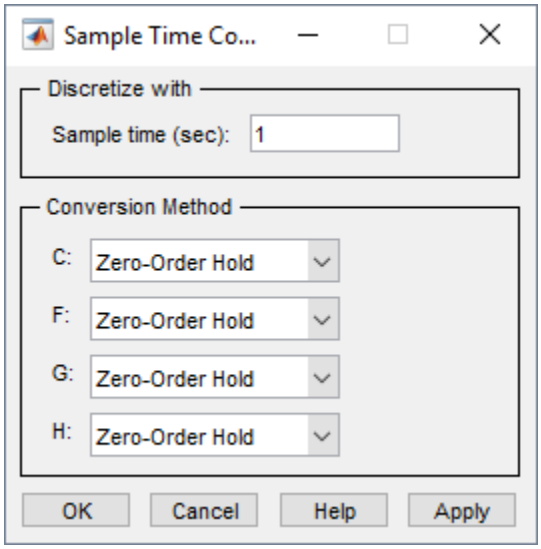
For more information, see `c2d` and `c2dOptions`.

## Control System Designer: Change sample time of control system

You can now modify the sample time of your control system in **Control System Designer**. To do so, on the **Control System** tab, under **Edit Architecture**, click **Sample Time Conversion**.




In the Sample Time Conversion dialog box, specify the **Sample time**, and select a **Conversion method** for each block in the control system. For more information on the available conversion methods, see Continuous-Discrete Conversion Methods. **Control System Designer** does not support the new least-squares vector fitting approach.




Click **OK**.

The app converts the dynamic model of each block to discrete time using the specified sample time and conversion method.

If your model is already in discrete time, you can choose to convert it to continuous time or to resample the system using a different sample time.

## Control System Designer: Create Simulink model for control architecture

You can now generate a Simulink model for your tuned control system architecture in **Control System Designer**.

For more information, see Generate Simulink Model for Control Architecture.

# R2017b

## Gain Scheduling: Implement gain-scheduled controllers using a new library of blocks configured to take scheduled parameters as inputs

A new library of Simulink blocks lets you implement common control-system elements with variable parameters. The new blocks in the Linear Parameter Varying library take parameter values as inputs, letting you compute those values at run-time in your model.

These blocks are useful for implementing gain-scheduled controller elements in which the parameter values vary as a function of scheduling variables. For instance, the new Varying Notch Filter block accepts as inputs the notch frequency, the gain at the notch frequency, and the damping ratio of the filter poles. In your model, you connect each of these inputs to an element such as a lookup table or MATLAB function block that computes the parameter value from the scheduling variables.

The available variable-parameter control-system elements include:

- Varying Transfer Function, Discrete Varying Transfer Function
- Varying Lowpass Filter, Discrete Varying Lowpass
- Varying Notch Filter, Discrete Varying Notch
- Varying State Space, Discrete Varying State Space
- Varying Observer Form, Discrete Varying Observer Form

The library also includes versions of the PID Controller and PID Controller (2DOF) blocks that are preconfigured to accept PID coefficients as inputs.

To access the new library, in the Simulink Library Browser, select **Control System Toolbox > Linear Parameter Varying**. For more information about using the blocks to implement gain-scheduled control systems, see Model Gain-Scheduled Control Systems in Simulink.

## Gain Scheduling: Achieve smooth and memory-efficient implementation by turning gain surfaces into embedded equations

In a gain-scheduled control system, controller gains vary as a function of one or more scheduling variables. In a Simulink model, one way to implement a scheduled gain is to use a MATLAB function block to represent the equations relating controller gains to scheduling-variable values. You can now use `systune` to tune those equations automatically and write the resulting relationship back to the MATLAB function block. Previously, you had to convert the tuned relationship to a discrete lookup table relating scheduling variables to gain values. While you can still use this approach, the new functionality can achieve smoother variation of scheduled gains by evaluating the equation that relates gains to scheduling variables.

To tune MATLAB function blocks representing scheduled gain values, you parameterize each MATLAB function block with a `tunableSurface` object that represents the equation relating gain to scheduling parameters. When you write the tuned parameters back to your model, the MATLAB function is automatically updated with MATLAB® code for the tuned gain surface. Use the new `codegen` command to examine the MATLAB code for the gain schedule.

If you have a code-generation product such as Simulink Coder, you can implement the tuned gain schedule in hardware. The new functionality can result in more memory-efficient implementation, storing only the gain surface coefficients rather than a potentially long list of lookup-table values.

For more information about modeling and tuning gain-scheduled control systems in Simulink, see:

- Model Gain-Scheduled Control Systems in Simulink
- Tune Gain Schedules in Simulink

Tuning control systems modeled in Simulink requires Simulink Control Design™ software.

## Gain-Scheduled Controller Tuning: Automatically tune gain-scheduled state observer gain, LQR gain, and other controller architectures expressed as matrices

You can now use `systune` for automatic tuning of matrix-valued gain schedules and implement them in MATLAB function blocks or Matrix Interpolation blocks. For instance, suppose that you want to implement a time-varying LQG controller of the form:

$$dx_e = Ax_e + Bu + L(t)(y - Cx_e - Du)$$
$$u = -K(t)x_e,$$

where the state-feedback matrix $K$ and the observer-gain matrix $L$ vary with time. In your Simulink model, use the new Varying Observer Form block to represent the LQG controller. Then use MATLAB function blocks to implement the time-varying matrices as inputs to the Varying Observer Form block. When you set up the model for tuning, parameterize the MATLAB function block using a `tunableSurface` that computes a matrix value as a function of time. When you tune the control system with `systune` and write the tuned controller parameters back to the model, the MATLAB function block is automatically updated with MATLAB code for the tuned gain surface.

For more information about:

- The Varying Observer Form block and other new variable-parameter blocks for gain scheduling, see “Gain Scheduling: Implement gain-scheduled controllers using a new library of blocks configured to take scheduled parameters as inputs” on page 7-2.
- Modeling gain-scheduled control systems in Simulink, see Model Gain-Scheduled Control Systems in Simulink.
- Creating gain surfaces for tuning gain schedules, see Parameterize Gain Schedules.

Tuning control systems modeled in Simulink requires Simulink Control Design software.

## Gain-Scheduled Controller Tuning: Specify tuning goals that vary with operating condition

When tuning fixed or gain-scheduled controllers at multiple design points representing different operating conditions, it is sometimes useful to adjust the design objectives as a function of operating condition. For example, you might want to relax design requirements in some regions of the operating range. The new `varyingGoal` function lets you construct tuning goals that depend implicitly or explicitly on the design point.

For more information about configuring varying requirements for gain scheduling, see:

- The `varyingGoal` reference page
- Change Requirements with Operating Condition

## Tuning Gain Surfaces: Custom normalization, lookup-table updates, and other enhancements

This release includes enhancements to tunable gain surfaces that improve the workflow for tuning gain schedules.

- Custom normalization of scheduling variables in tunable surface — By default, the `tunableSurface` representation of a gain surface normalizes the scheduling variables so that the design range of each variable falls within the interval [–1,1]. You can change this normalization using the new `Normalization` property of `tunableSurface`. Changing the normalization is useful, for example, when you have a known gain value for a design point, or want to restrict a scheduling variable to positive values. For more information, see `tunableSurface`.
- Update portion of lookup table — You can now update a single point or a portion of a lookup table in a Simulink model that you tuned using `slTuner` and `systune`. To perform the update, use the new `writeLookupTableData` command. This command is useful when you are retuning a single design point or a subset of design points covered by the lookup table. Previously, you could only use `writeBlockValue` to update the entire lookup table.
- Name scheduling variables in basis functions — New syntaxes for `polyBasis` and `fourierBasis` let you assign names to the input variables of the basis functions generated by these commands. The names are preserved when you combine basis functions using `ndBasis`. Naming basis-function variables can improve readability of the generated basis functions and of code generated from a `tunableSurface` object that you create with the basis functions. For more information, see the reference pages for these commands.
- Tunable surface with constant gain — A new syntax of `tunableSurface` lets you create a flat gain surface with constant, tunable gain. For more information, see `tunableSurface`.

## Gain-Scheduled Controller Tuning: Exclude design points from tuning or analysis

When you have created a design grid of tunable, linearized models for gain-scheduled controller tuning, you can now exclude one or more design points from tuning without removing the corresponding model from the array. Doing so can be useful, for example, to identify problematic design points when tuning over the entire design grid fails to meet your design requirements. It can also be useful when there are design points that you want to exclude from a particular tuning run, but preserve for performance analysis or further tuning. To exclude design points from tuning, use the new `SkipModels` option of `systuneOptions`, which lets you specify models in the design grid to exclude from tuning.

As an alternative, you can eliminate design points from the model grid entirely, so that they do not contribute to any stage of tuning or analysis. To do so, use the new `voidModel` command, which replaces specified models in a model array with `NaN`. This approach is useful when your sampling grid includes points that represent irrelevant or unphysical design points. Using `voidModel` lets you design over a grid of design points that is almost regular.

For more information about controlling how different design points contribute to tuning, see Change Requirements with Operating Condition.

## Particle Filters: Estimate states of nonlinear systems

Perform state estimation for arbitrary nonlinear system models using the new `particleFilter` command. Particle filters are flexible, that is, they can also perform state estimation for nonlinear

systems with non-Gaussian distributions. Previously, you could perform state estimation only for systems with unimodal distributions using extended or unscented Kalman filters.

`particleFilter` uses particles and sensor data to estimate the posterior distribution of the current state. The filter predicts the states using the nonlinear state transition function. Then, it corrects the estimate based on sensor data and measurement likelihood model. You can specify a fixed number of particles to use, a fixed number of state variables to estimate, and your state estimation method based on the particle weights.

To use a particle filter for state estimation:

1. Create a particle filter, and set the transition and measurement likelihood functions.
2. Initialize the particle filter by specifying the number of particles to be used and your initial state guess. Also specify state bounds or covariance of the initial particle distribution.
3. Specify the state estimation and resampling method.
4. Perform state estimation.

You can use MATLAB Compiler™ or MATLAB Coder software to deploy the particle filter for your application.

For more information and examples, see the `particleFilter` reference page.

## Improved lqg Function: Compute gain matrices and optimal controller in discrete time using current Kalman Filter estimator

When designing an LQG controller for a discrete-time plant using the `lqg` function, you can now use the current type of Kalman estimator, which uses `x[n|n]` as the state estimate. Previously, the `lqg` function supported using only the delayed type of Kalman estimator; that is, using `x[n|n-1]` as the state estimate. For more information about the types of Kalman estimators, see `kalman`.

Also, you can now return the controller and estimator gain matrices when using the `lqg` function. You can use the controller and estimator gains to, for example, implement the controller in observer form.

For more information, see `lqg`.

## Model Reduction: balred no longer ignores MatchDC option when specified frequency or time intervals exclude DC

When you use `balred` for model reduction, you can use `balredOptions` to restrict the computation to specified frequency or time intervals. If the `StateElimMethod` option of `balredOptions` is set to `'MatchDC'` (the default value), then `balred` attempts to match the DC gain of the original and reduced models, even if the specified intervals exclude DC (frequency = 0 or time = `Inf`). This behavior might reduce the quality of the match in the specified intervals. To improve the match within intervals that exclude DC, set `StateElimMethod = 'Truncate'`. For more information, see `balredOptions`.

In the Model Reducer app, there is no change in behavior.

## Compatibility Considerations

Previously, if time or frequency intervals excluded DC, `balred` did not attempt to match the DC gain of the original and reduced models, even if `StateElimMethod = 'MatchDC'`. If you have scripts or

functions that use `balred` with restricted time or frequency intervals that exclude DC, consider updating them to set `StateElimMethod = 'Truncate'`. The `balred` command now issues a warning when `StateElimMethod = 'MatchDC'` and the specified time or frequency intervals exclude DC.

## Regularization of conic-sector tuning goal in Control System Tuner

The Conic Sector Goal in the **Control System Tuner** app has a new **Regularization** option that allows you to specify a nonzero regularization parameter. This option is useful when other tuning goals tend to make the sector bound ill-conditioned at some frequencies. For more information, see Conic Sector Goal.

This new option is equivalent to the `Regularization` property of `TuningGoal.ConicSector`, introduced in R2017a for command-line tuning.

## Dynamic system models store Notes property as string or character vector

The `Notes` property of a dynamic system model stores any text that you want to associate with the model. This property now accepts either character-vector or `string` values, and stores whichever type you provide. For instance, if `sys1` and `sys2` are dynamic system models, you can set their `Notes` properties as follows:

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans =

    "sys1 has a string."


ans =

  1×1 cell array

    {'sys2 has a character vector.'}
```

When you create a new model, the default value of `Notes` is now `[0×1 string]`. Previously, you could only specify the `Notes` property as a character vector or cell array of character vectors, and the default value was `{}`.

Some other dynamic system model properties accept strings as inputs, but store the values as character vectors or a cell array of character vectors.

## Functionality being removed or changed

| Functionality | Result | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `balred` with `StateElimMethod = 'MatchDC'` and restricted time or frequency range that excludes DC | Warns | `StateElimMethod = 'Truncate'` | See “Model Reduction: balred no longer ignores MatchDC option when specified frequency or time intervals exclude DC” on page 7-5. |
| `viewSpec`, `evalSpec` | Still works | `viewGoal`, `evalGoal` | If you have functions or scripts that use `viewSpec` or `evalSpec`, consider updating them to use `viewGoal` and `evalGoal` instead. |

# R2017a

## Extended and Unscented Kalman Filter Simulink Blocks: Estimate states of nonlinear systems

You can now use the Extended Kalman Filter and Unscented Kalman Filter blocks to estimate the states of a discrete-time nonlinear system in Simulink. The blocks use first-order extended and unscented Kalman filter algorithms to estimate states as new data becomes available during the operation of the system. Previously, nonlinear state estimation using these algorithms was available at the command line only. You can use the state estimates for state-feedback controllers and for applications such as condition monitoring and fault detection. You can also generate C/C++ code for these blocks using Simulink Coder software.

For information about how to use these blocks, see the Extended Kalman Filter and Unscented Kalman Filter block reference pages. For examples, see Estimate States of Nonlinear System with Multiple, Multirate Sensors and Nonlinear State Estimation of a Degrading Battery System.

## New properties of generalized state-space models and matrices

The generalized state-space model object, `genss`, has the following new properties:

- `StateName` and `StateUnit` — Track the state names and state units of the fixed LTI components and control design blocks that make up the model. When you build a `genss` model from fixed and tunable LTI components, it inherits the `StateName` and `StateUnit` values from these components
- `A`, `B`, `C`, `D`, and `E` — Access the state-space matrices of a `genss` model. These properties model the dependency of the state-space matrices on static control design blocks, such as `realp`, `ureal`, `ucomplex`, or `ucomplexm`.

  For more information and examples, see the `genss` reference page.

Additionally, the generalized matrix object, `genmat`, now has a `Name` property. Use the property to assign a name to the generalized matrix. When you convert a static control design block such as `tunableSurface` to a generalized matrix using `genmat(blk)`, the `Name` property of the block is preserved.

## Discrete-time frequency-dependent specifications for tuning goals

You can now use discrete-time LTI models to specify frequency-dependent gain profiles for tuning in discrete time. Tuning goals that you can now specify in discrete time include:

- Frequency-dependent minimum gains, maximum gains, and loop gains.
- Frequency-dependent rejection, sensitivity, or error profiles.
- Frequency-dependent weighting functions.

If you specify the gain profile in continuous time for tuning in discrete time, the tuning software discretizes the profile. Specifying the gain profile in discrete time gives you more control over the gain profile near the Nyquist frequency. For more information, see the documentation for the individual tuning goals.

## Regularization of tuning goals for improved numeric stability

When you use a tuning goal with a frequency-dependent specification, the tuning algorithm uses a frequency-weighting function to compute the normalized value of the tuning goal. This weighting

function is derived from the gain profile that you specify. For numeric stability and tractability, the software now adjusts the specified gain profile when necessary to eliminate undesirable low-frequency or high-frequency dynamics or asymptotes. This adjustment process is called regularization.

The regularized gain profile is displayed on tuning-goal plots generated with `viewSpec` or in Control System Tuner. For affected tuning goals, the `getWeight` or `getWeights` command extracts the regularized frequency-weighting functions. For more information about regularization, see Visualize Tuning Goals and the documentation for the individual tuning goals.

Also, the conic sector goal has a new `Regularization` property that allows you to specify a nonzero regularization parameter. This property is useful when other tuning goals tend to make the sector bound ill-conditioned at some frequencies. For more information, see `TuningGoal.ConicSector`.

## Maximum Natural Frequency Option in Control System Tuner: Prevent poles and zeros from going to infinity

Most tuning goals in the **Control System Tuner** app include implicit stability or minimum-phase constraints. The new **Maximum natural frequency** tuning option constrains the maximum natural frequency of the corresponding stabilized poles and zeros. This option is useful to prevent poles and zeros from going to infinity as a result of algebraic loops becoming singular or control effort growing unbounded. To access the option in the app, on the **Tuning** tab, click **Tuning Options**.

For more information about stabilized poles and zeros, see the documentation for each tuning goal, listed on the Tuning Goals page.

The new option is equivalent to the `MaxRadius` option of `systuneOptions`, introduced in R2016b for command-line tuning.

## Scaling information passed automatically to viewSpec and evalSpec

When you use `systune` to tune a MIMO feedback loop, some tuning goals are sensitive to the relative scaling of each SISO loop. `systune` tries to balance the overall loop-transfer matrix while enforcing such goals. The optimal loop scaling is now stored in the tuned closed-loop model returned by `systune`. When you pass the tuned model to `viewSpec` or `evalSpec` to examine tuning results, these functions take this scaling into account. Previously, you had to pass the `info` output of `systune` to these functions to ensure consistent scaling.

For more information, see `viewSpec` or `evalSpec`.

## Compatibility Considerations

You no longer need to use the syntaxes `viewSpec(Req,CL,info)` and `evalSpec(Req,CL,info)` to ensure consistent scaling. The syntaxes `viewSpec(Req,CL)` and `evalSpec(Req,CL)` obtain any necessary scaling information from the closed-loop model `CL` returned by `systune`. To force the functions to disregard scaling information, use `viewSpec(Req,CL,[])` or `evalSpec(Req,CL,[])`.

## Functionality being removed or changed

| Functionality | Result | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `Q` property of `TuningGoal.ConicSector` | Still works | `SectorMatrix` | If you have scripts or functions that refer to the `Q` property of a `TuningGoal.ConicSector` object, update them to use the new property name `SectorMatrix`. |
| • `viewSpec(Req,CL,info)`<br>• `evalSpec(Req,CL,info)` | Still works | • `viewSpec(Req,CL)`<br>• `evalSpec(Req,CL)` | If you have scripts or functions that use the `info` argument, consider updating them to remove the argument. See “Scaling information passed automatically to viewSpec and evalSpec” on page 8-3. |

# R2016b

## Conic Sector Tuning Goal: Tune control systems to enforce fixed or frequency-dependent sector bounds

A conic system is a system whose trajectories lie in a given conic sector of I/O space. In some control applications, it is useful to restrict system trajectories to a particular sector. Such bounds arise, for example, in robust control of feedback loops with static nonlinearities. New goals for control system tuning let you impose conic sector bounds on responses of the tuned system.

- Use `TuningGoal.ConicSector` for tuning at the command line with `systune`.
- Use Conic Sector Goal for tuning in Control System Tuner.

For more information about conic systems, see About Sector Bounds and Sector Indices.

## Improved Passivity Tuning Goal: Set output passivity index to a negative value

You can now specify a negative output passivity index when tuning a control system using `systune` or Control System Tuner. Specifying a negative output passivity index for a particular response lets the tuned response have a shortage of passivity at the output. Negative output passivity index values are permitted in:

- `TuningGoal.Passivity` and `TuningGoal.WeightedPassivity`, for tuning at the command line with `systune`.
- Passivity Goal and Weighted Passivity Goal, for tuning with Control System Tuner.

Previously, you could specify a negative or positive input passivity index with these tuning goals, but only a positive output passivity index.

For more information about passivity indices, see About Passivity and Passivity Indices.

## MaxRadius Option for Tuning: Prevent poles and zeros from going to infinity

Most tuning goals used for control system tuning with `systune` include implicit stability or minimum-phase constraints. The new `MaxRadius` option of `systuneOptions` constrains the maximum natural frequency of the corresponding stabilized poles and zeros. This option is useful to prevent poles and zeros from going to infinity as a result of algebraic loops becoming singular or control effort growing unbounded. For more information, see the `systuneOptions` reference page and the reference pages for each tuning goal.

## Improved getSectorIndex and sectorplot Functions: Compute and plot sector index for unstable systems

You can now use `getSectorIndex` and `sectorplot` to analyze the sector index for both stable and unstable systems. Previously, providing an unstable system as input to either of these functions generated an error.

## Extended and Unscented Kalman Filters: Estimate states of nonlinear systems

You can now estimate the states of discrete-time nonlinear systems at the command line using first-order extended Kalman filter algorithms and unscented Kalman filter algorithms. The new state estimation commands `extendedKalmanFilter` and `unscentedKalmanFilter` are useful for estimation of states when new data is available during the operation of the system. You can use the state estimates for state-feedback controllers and for applications such as condition monitoring and fault detection. You can use MATLAB Compiler or MATLAB Coder software to deploy the estimators in your application.

For an example of online state estimation, see Nonlinear State Estimation Using Unscented Kalman Filter.

## Phase-Wrap Branch Option: Specify cutoff point for wrapping phase in response plots

By default, response plots that show phase response, such as Bode and Nichols plots, display the exact phase. You can make these plots wrap the phase into the interval [–180º,180º) by checking **Wrap Phase** in the plot Property Editor, the Linear System Analyzer Preferences Editor, or the Toolbox Preferences Editor.

In R2016b, checking **Wrap Phase** enables a new **Branch** field that lets you specify the value at which accumulated phase wraps in the response plot. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º).

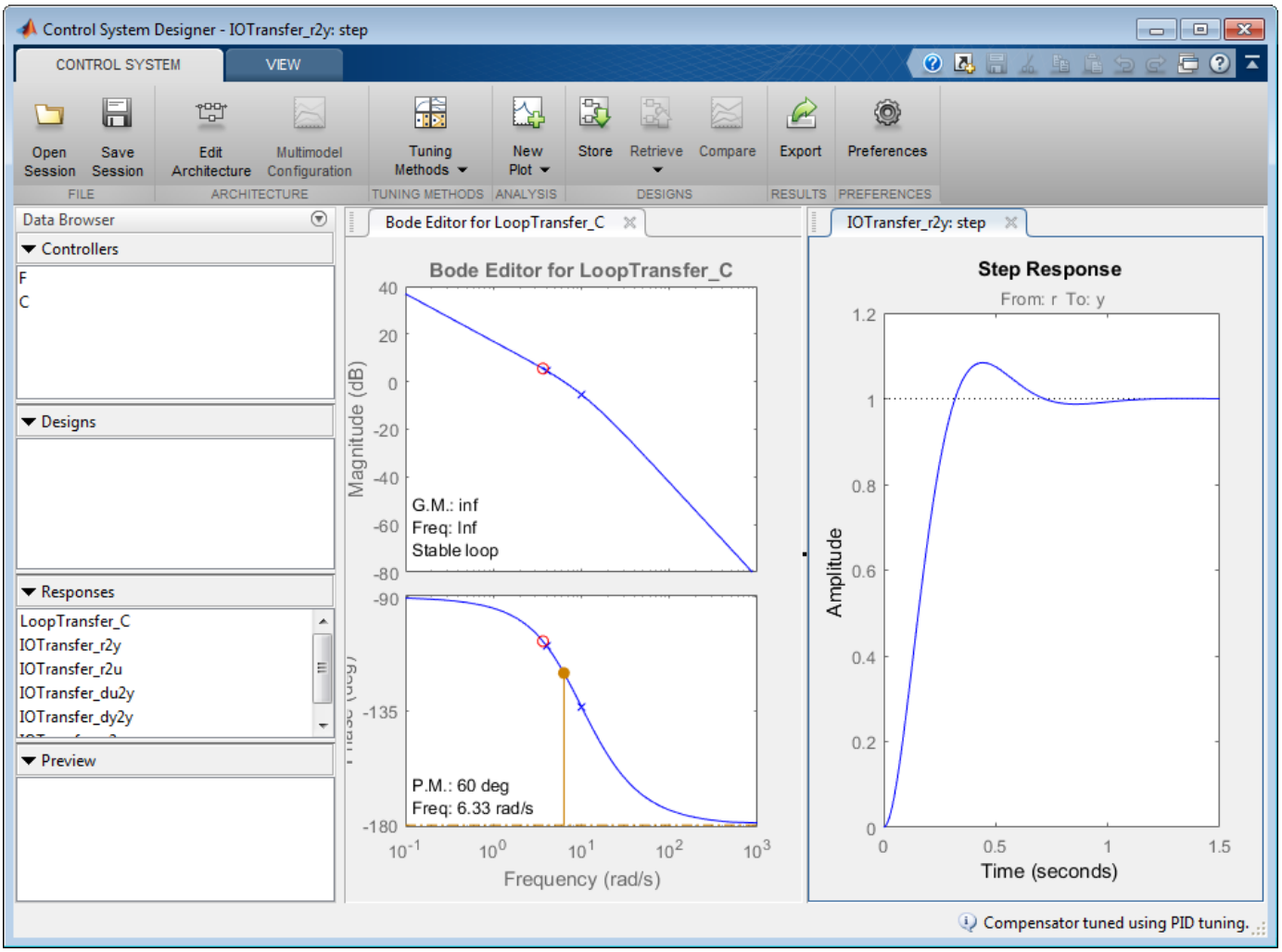
At the command line, turn on phase wrapping by setting the `PhaseWrapping` option of `bodeoptions` or `nicholsoptions` to `'on'`. Specify the phase-wrap cutoff point using the new `PhaseWrappingBranch` option.

In R2015b and R2016a, phase-wrapped plots used the interval [0º,360º). Before R2015b, phase-wrapped plots used the interval [–180º,180º).

# R2016a

## Redesigned Control System Designer App: Design SISO controllers for feedback systems using improved interactive workflows

The redesigned **Control System Designer** app streamlines workflows for designing SISO controllers for feedback control systems using graphical and automated tuning methods.




For more information on using **Control System Designer**, see:

- Control System Designer
- Control System Designer Tuning Methods
- Bode Diagram Design
- Design Compensator Using Automated Tuning Methods
- Analyze Designs Using Response Plots

## Control System Tuner App and systune Command: Automatically tune single-loop and multiloop control systems to meet design requirements

Control System Toolbox now includes automatic tuning tools that previously required a Robust Control Toolbox license. Control System Tuner and the `systune` command automatically tune control systems from high-level design goals you specify, such as reference tracking, disturbance rejection, and stability margins.

To tune a control system, you specify the tunable elements of your control system. You then capture your design requirements using the library of tuning goals. The software jointly tunes all the free parameters of your control system regardless of architecture, number of feedback loops, or whether it is modeled in MATLAB or Simulink. (Tuning control systems modeled in Simulink requires a Simulink Control Design license.)

For information about using these tools, see:

- Tuning with Control System Tuner
- Programmatic Tuning

You can also use the `systune` command to tune gain-scheduled controllers for control systems in which plant dynamics change with operating conditions or time. For more information, see Gain Scheduling.

## Model Reducer App: Compute and compare reduced-order models using interactive workflows

The new **Model Reducer** app is an interactive tool for computing reduced-order approximations of high-order models. Working with lower-order models can simplify analysis and control design. Simpler models are also easier to understand and manipulate. You can reduce a plant model to focus on relevant dynamics before designing a controller for the plant. Or, you can use model reduction to simplify a full-order controller. Using any of the following model-reduction methods, **Model Reducer** helps you reduce model order while preserving model characteristics that are important to your application:

- Balanced Truncation — Remove states with low energy contributions.
- Pole/Zero Simplification — Eliminate canceling or near-canceling pole-zero pairs.
- Mode Selection — Select modes by specifying a region of interest in the complex plane.

**Model Reducer** also provides response plots and error plots to help ensure that the reduced-order model preserves important dynamics.

For an example showing how to use **Model Reducer**, see Reduce Model Order Using the Model Reducer App. For more information about model reduction generally, see Model Reduction Basics.

## Passivity and Conic Sectors: Analyze and tune control systems for passivity and other sector bounds

A linear system is passive when it cannot produce energy on its own and can only dissipate the energy initially stored in it. More generally, an I/O map is passive if, on average, increasing the output

requires increasing the input. Passive control is often part of the safety requirements in process control, tele-operation, human-machine interfaces, and system networks.

Use the following new commands to analyze the passivity of linear systems:

- `isPassive` — Check passivity of linear system.
- `getPassiveIndex` — Compute various measures of the excess or shortage of passivity of a linear system.
- `passiveplot` — Calculate and plot passivity index as a function of frequency.

Mathematically, a system is passive when all its I/O trajectories are restricted to a particular sector of the I/O space. More generally, a conic system is a system whose trajectories lie in a given conic sector. Conic sector bounds arise, for example, in robust control of feedback loops with static nonlinearities. The following new commands let you analyze how well a linear system lies within any conic sector.

- `getSectorIndex` — Check whether the output trajectories of a linear system lie in a particular conic sector. Compute the relative sector index, a measure of how tightly the trajectories fit within the sector.
- `getSectorCrossover` — Compute the frequencies at which the range of trajectories crosses the sector boundary.
- `sectorplot` — Calculate and plot sector index as a function of frequency.

New tuning goals let you enforce passivity when tuning a control system using Control System Tuner or the `systune` command.

| Constraint | Command Line | Control System Tuner |
|---|---|---|
| Enforce passivity on specified I/Os in the control system | `TuningGoal.Passivity` | Passivity Goal |
| Enforce passivity on specified I/Os with frequency-dependent weighting | `TuningGoal.WeightedPassivity` | Weighted Passivity Goal |

For more background and details about the notions of passivity and sector bounds, see:

- About Passivity and Passivity Indices
- About Sector Bounds and Sector Indices

## Limited Balanced Truncation: Reduce model order according to energies within time-domain and frequency-domain intervals

You can now perform balanced-truncation model reduction and compute Hankel singular values based on state energies calculated within specified time and frequency intervals.

- To perform frequency-limited or time-limited balanced truncation, use `balredOptions` to set the `TimeIntervals` or `FreqIntervals` options for the `balred` command. When you use these options, `balred` determines which states to truncate based on their energy contributions within the specified interval only. For more details, see the reference pages for `balredOptions` and `balred`.

- To compute or plot Hankel singular values for specific time or frequency, use `hsvdOptions` or `hsvoptions` to set the `TimeIntervals` and `FreqIntervals` options for the `hsvd` and `hsvplot` commands, respectively. For more details, see the reference page for `hsvdOptions`.

These operations use the new functionality in the `gram` command that computes time-limited and frequency-limited controllability and observability Gramians. For details, see the reference pages for `gram` and the new `gramOptions` command.

## sampleBlock and rsampleBlock commands for sampling generalized models

The new `sampleBlock` and `rsampleBlock` commands sample Control Design blocks in a generalized model such as a `genss` or `uss` model. You can sample tunable blocks, uncertain blocks, or both. The output is the model array obtained by replacing the sampled blocks by values you specify (`sampleBlock`) or randomized values (`rsampleBlock`). Some uses of these functions include:

- Perform sensitivity analysis by varying tunable parameters randomly or across a grid.
- Study robustness by varying parameters over an uncertainty range.

For more information, see the `sampleBlock` and `rsampleBlock` reference pages.

## Spectral factorization of LTI models

The new `spectralfact` command computes the spectral factorization of LTI models. The spectral factorization of a model `H` is:

`H = G'*S*G`,

where `S` is a symmetric matrix and `G` is a square, stable, and minimum-phase system with unit (identity) feedthrough. `H` must satisfy `H = H'`. For more information, see the `spectralfact` reference page.

## Renamed tunable control design blocks

The tunable control design blocks have been renamed. Starting in R2016a, use the following block names:

| Control Design Block | New Name |
|---|---|
| Tunable gain block | `tunableGain` |
| Fixed-order state-space model with tunable coefficients | `tunableSS` |
| SISO fixed-order transfer function with tunable coefficients | `tunableTF` |
| One-degree-of-freedom PID controller with tunable coefficients | `tunablePID` |
| Two-degree-of-freedom PID controller with tunable coefficients | `tunablePID2` |

Also, several properties of tunable state-space and tunable transfer function blocks have changed. For more information, see “Functionality being removed or changed” on page 10-6.

The remaining block functionality and properties remain unchanged.

## Compatibility Considerations

If your code uses tunable control design blocks, modify your code to use the new block names. For more information, see “Renamed tunable control design blocks” on page 10-5.

## Functionality being removed or changed

| Functionality | Result | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `ltiblock.gain`, `ltiblock.ss`, `ltiblock.tf`, `ltiblock.pid`, and `ltiblock.pid2` | Still works | `tunableGain`, `tunableSS`, `tunableTF`, `tunablePID`, and `tunablePID2` respectively. | For more information, see “Renamed tunable control design blocks” on page 10-5. |
| `a`, `b`, `c`, `d`, and `e` properties of `ss` and `tunableSS` models. | Still works | `A`, `B`, `C`, `D`, and `E` respectively. | If your code uses any of these properties, consider modifying your code to use the new property names. For more information, see `ss`. |
| `num`, `den`, and `ioDelay` properties of `tf` and `tunableTF` models. | Still works | `Numerator`, `Denominator`, and `IODelay` respectively. | If your code uses any of these properties, consider modifying your code to use the new property names. For more information, see `tf`. |
| `z`, `p`, `k`, and `ioDelay` properties of zero-pole-gain models | Still works | `Z`, `P`, `K`, and `IODelay` respectively. | If your code uses any of these properties, consider modifying your code to use the new property names. For more information, see `zpk`. |

# R2015b

## pid2 and pidstd2 Model Objects: Represent, analyze, and use 2-DOF PID controllers for control design

The new numeric LTI models `pid2` and `pidstd2` are specialized for modeling two-degree-of-freedom (2-DOF) PID controllers. These are analogous to the 1-DOF PID models `pid` and `pidstd`.

Use `pid2` and `pidstd2` to represent a 2-DOF PID controller directly with the PID parameters, expressed in parallel (`pid2`) or standard (`pidstd2`) form. For example, `C2 = pid2(Kp,Ki,Kd,Tf,b,c)` creates a 2-DOF PID controller in parallel form with proportional, integral, and derivative gains `Kp`, `Ki`, and `Kd`, derivative filter time constant `Tf`, and setpoint weights `b` and `c`. In previous releases, to model a 2-DOF PID controller, you had to derive the controller's equivalent transfer function (or other model), and could not directly store the 2-DOF PID parameters.

The `pid2` and `pidstd2` commands can also convert to PID form any type of LTI object that represents a 2-DOF PID controller.

This release also includes new functions to help you work with 2-DOF PID controllers. These functions include:

- `getComponents` — Extract two SISO control components from a 2-DOF `pid2` or `pidstd2` controller.
- `make1DOF` and `make2DOF` — Convert 1-DOF `pid` and `pidstd` controllers to 2-DOF `pid2` and `pidstd2` controllers, and vice versa.
- `piddata2` and `pidstddata2` — Access parameters of 2-DOF PID controllers.

For more information about working with 2-DOF PID controller objects, see:

- Two-Degree-of-Freedom PID Controllers
- `pid2` and `pidstd2` function reference pages

## 2-DOF PID Controller Tuning: Automatically tune the gains of 2-DOF PID controllers with PID Tuner app and pidtune command

You can now use `pidtune` and PID Tuner to tune all parameters of a two-degree-of-freedom (2–DOF) PID controller, including the setpoint weights *b* and *c*. When you call `pidtune` or open the PID Tuner app with a plant, the software automatically tunes all parameters of the block to achieve a balance between performance and robustness. When you use the Response Time and Transient Behavior sliders to adjust that balance, PID Tuner adjusts all parameters, including *b* and *c* if necessary.

PID Tuner and `pidtune` also include options for tuning 2-DOF PID controllers with fixed setpoint weights, such as I-PD ($b = 0$, $c = 0$) and P-ID ($b = 1$, $c = 0$).

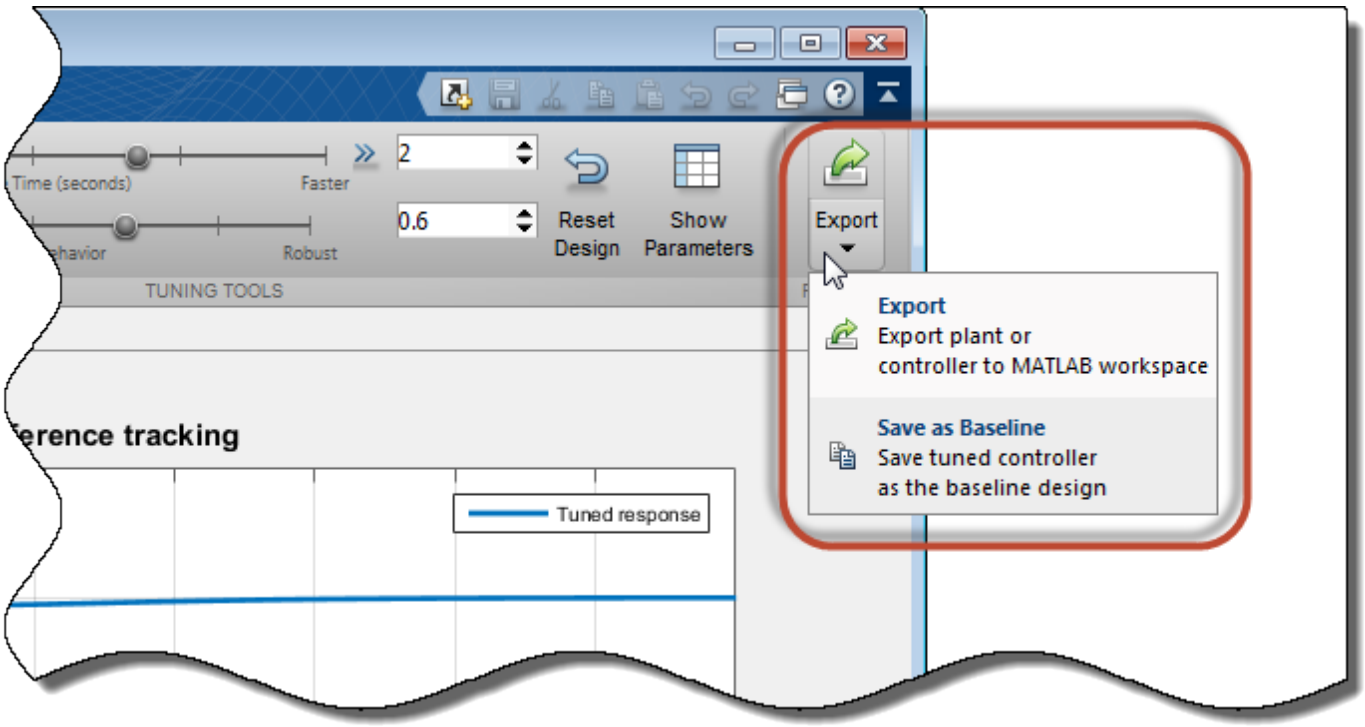
For more information, see:

- Tune 2-DOF PID Controller (PID Tuner)
- Tune 2-DOF PID Controller (Command Line)
- PID Controller Types for Tuning

## Save Current Controller Design as Baseline in PID Tuner

In PID Tuner, you can now make the current controller design the baseline controller at any time. This feature allows to you compare the performance of a PID Tuner controller design to the performance of controllers obtained by further adjustment of the design.

To make the current PID Tuner design the baseline controller, by click the **Export** arrow and select `Save as Baseline`.




When you do so, the current Tuned response becomes the Baseline response. Further adjustment of the current design creates a new Tuned response line.

Previously, you could only designate a baseline controller when you opened the PID Tuner using the syntax `pidTuner(sys,C0)`.

For more information about analyzing controller performance in PID Tuner, see Analyze Design in PID Tuner.

## Change in LPV System block default values for model delays

The default value is now `false` for **Input delay**, **Output delay**, and **Internal delay**, in the **Fixed Entries** tab of LPV System Block Parameters dialog box. A `false` value means that model delays are treated as free during simulation.

For information about changing the default values, see the LPV System block reference page.

## Analysis Plots Wrap Phase in Interval [0º,360º)

By default, response plots that show phase response, such as Bode and Nichols plots, display the exact phase. You can make these plots wrap the phase into the interval [0º,360º) by unchecking **Unwrap Phase** in the plot Property Editor. Previously, unchecking this option caused the plots to wrap the phase into [–180º,180º). The change makes it easier to visualize 180º phase crossings on analysis plots.

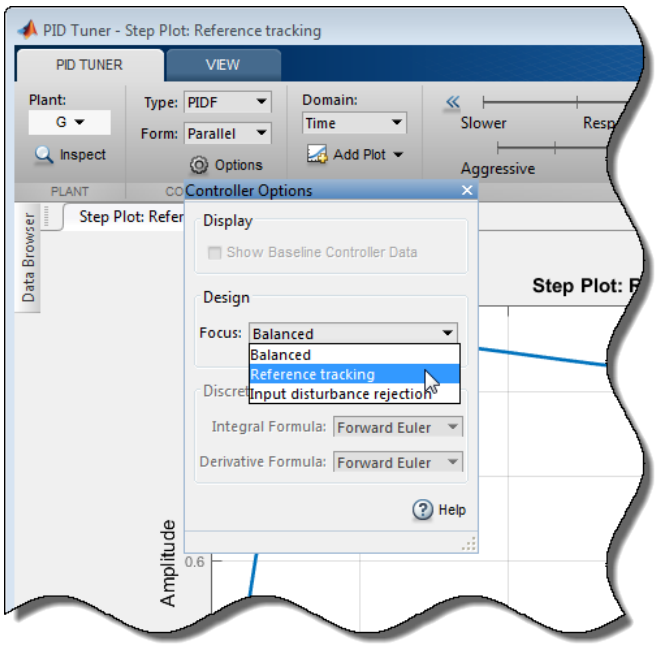
The default plot behavior (exact or unwrapped phase) is unchanged. The behavior of commands that return numerical phase response data, such as `bode`, is also unchanged. These commands always return the unwrapped phase.

# R2015a

## Improved input disturbance rejection with the PID tuning algorithm

Controller tuning with the PID Tuner app or the `pidtune` command now yields better disturbance rejection by default. For a given target phase margin, the tuning algorithm selects PID coefficients that achieve a balance between reference tracking and input disturbance rejection. If you require more disturbance rejection or better reference tracking than the default algorithm provides, PID Tuner and `pidtune` have a new Design Focus option. Use this option to alter the balance that the tuning algorithm sets between reference tracking and input disturbance rejection. For instance, setting the design focus to reference tracking improves the reference tracking performance of the tuned controller, with some cost to disturbance rejection. Similarly, setting the design focus to input disturbance rejection improves rejection with some cost to reference tracking. Changing design focus is most effective when tuning PID and PIDF controllers, rather than controllers with fewer free parameters, such as PI.

To use the Design Focus option in PID Tuner, click **Options** and select a design focus from the **Focus** menu.




You can still use the **Response Time** and **Transient Behavior** sliders to further adjust the balance between reference tracking and input disturbance rejection.

To specify a design focus for command-line tuning with `pidtune`, use `pidtuneOptions` to set the `DesignFocus` option. For example, the following commands design a PIDF controller for the plant `G` with a crossover frequency of 10 rad/s, specifying reference tracking as the design focus.

```
opt = pidtuneOptions('DesignFocus','reference-tracking');
C = pidtune(G,'pidf',10,opt);
```

For more information about using the design focus option, see:

- Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)
- Tune PID Controller to Favor Reference Tracking or Disturbance Rejection at Command Line

For more information about using PID Tuner, see Designing PID Controllers with the PID Tuner. For more information about command-line PID tuning, see PID Controller Design at the Command Line.

## Option to specify code generation settings in LPV System block

You can now specify code generation settings in the LPV System block. You specify these settings in the **Code Generation** tab of the block parameters dialog box.

For more information on Linear Parameter-Varying models, see Linear Parameter-Varying Models.

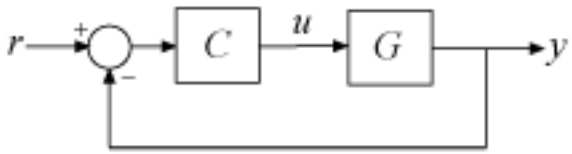
## connect command syntax for specifying analysis point locations

When you interconnect dynamic system models using the `connect` command, you can now specify analysis point locations as an input argument to the command. The following syntax creates a dynamic system model with analysis points by interconnecting multiple models `sys1,sys2,...,sysN`:

```
sys = connect(sys1,sys2,...,sysN,inputs,outputs,APs);
```

`inputs` and `outputs` are string vectors that specify the names of the inputs and outputs of the interconnected model. `APs` is a string vector that lists the signal locations at which to insert analysis points. The software automatically inserts an `AnalysisPoint` block with channels corresponding to these locations. Previously, you had to create `AnalysisPoint` blocks separately and include them in the list of models to connect.

For example, consider the following control system.




Use `connect` to build this system with an analysis point at the plant input, *u*.

```
C.InputName = 'e';  C.OutputName = 'u';
G.InputName = 'u';  G.OutputName = 'y';
Sum = sumblk('e = r-y');
T = connect(G,C,Sum,'r','y','u');
```

These commands return a generalized state-space (`genss`) model with one `AnalysisPoint` block. You can use the analysis point, for example, to extract the open-loop response of the system at *u*:

```
L = getLoopTransfer(T,'u',-1);
```

For a more detailed example, see Mark Analysis Points in Closed-Loop Models. For more information about using analysis points in dynamic system models, see the `AnalysisPoint` reference page.

## LTI Viewer renamed to Linear System Analyzer

The LTI Viewer app is now called Linear System Analyzer. The functionality of the app is unchanged.

You can access Linear System Analyzer in two ways:

- From the MATLAB desktop, in the **Apps** tab, in the **Control System Design and Analysis** section, click **Linear System Analyzer**.
- From the MATLAB command line, use the `linearSystemAnalyzer` function. Previously, this function was called `ltiview`. Using `ltiview` to open Linear System Analyzer does not generate an error in this release, but the function might be removed in a future release.

## Compatibility Considerations

If you have scripts or functions that use `ltiview`, consider replacing those calls with `linearSystemAnalyzer`.

## sisotool function renamed to controlSystemDesigner

The `sisotool` function is now called `controlSystemDesigner`. The `controlSystemDesigner` opens the SISO Design Tool. You can also access SISO Design Tool from MATLAB desktop. In the **Apps** tab, in the **Control System Design and Analysis** section, click **Control System Designer**.

Using `sisotool` to open SISO Design Tool does not generate an error in this release, but the function might be removed in a future release.

## Compatibility Considerations

If you have scripts or functions that use `sisotool`, consider replacing those calls with `controlSystemDesigner`.

## getBlockValue returns all block values in structure

A new syntax of the `getBlockValue` command now returns the current values of all Control Design Blocks of a generalized model (`genss`) in a structure. The following syntax returns a structure, `S`, whose field names are the names of the blocks in the `genss` model `M`. The values of the fields are numerical LTI models or numerical values equal to the current values of the corresponding Control Design Blocks.

```
S = getBlockValue(M)
```

This syntax lets you transfer the block values from one generalized model to another model that uses the same Control Design Blocks, as follows:

```
S = getBlockValue(M1);
setBlockValue(M2,S);
```

For more information, see the `getBlockValue` reference page.

## Compatibility Considerations

Previously, the syntax `getBlockValue(M)` returned the current values of the blocks of `M` as a vector list:

```
[Val1,Val2,...,ValN] = getBlockValue(M)
```

Now, using this syntax causes an error. You can still obtain block values in a list by specifying the block names as input arguments, as follows.

```
[Val1,Val2,...,ValN] = getBlockValue(M,Blkname1,Blkname2,...,BlknameN)
```

## Functionality being removed or changed

| Functionality | What Happens When You Use This Functionality ? | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `[Val1,Val2,...] = getBlockValue(M)` | Error | `S = getBlockValue(M)` | `getBlockValue(M)` now returns a structure containing current values of all blocks. Update scripts and functions that use `getBlockValue(M)` to use output structure. |
| `ltiview` function | Still works | `linearSystemAnalyzer` | Consider replacing `ltiview` with `linearSystemAnalyzer` in scripts and functions. |
| `sisotool` function | Still works | `controlSystemDesigner` | Consider replacing `sisotool` with `controlSystemDesigner` |

# R2014b

## LPV System block for modeling and simulating linear parameter-varying systems

This release introduces the LPV System block. You use this block to represent Linear Parameter Varying (LPV) systems in Simulink.

An LPV system is a linear state-space system whose dynamics vary as a function of certain time-varying parameters called the scheduling parameters. Mathematically, an LPV system is represented as:

$$dx(t) = A(p)x(t) + B(p)u(t)$$
$$y(t) = C(p)x(t) + D(p)u(t)$$
$$x(0) = x_0$$

where

- `u(t)` are the inputs
- `y(t)` the outputs
- `x(t)` are the model states with initial value `x0`
- $dx(t)$ is the state derivative vector $\dot{x}$ for continuous-time systems and the state update vector $x(t + \Delta T)$ for discrete-time systems. $\Delta T$ is the sample time.
- `A(p)`, `B(p)`, `C(p)` and `D(p)` are the state-space matrices parameterized by the scheduling parameter vector `p`.
- The parameters `p = p(t)` are measurable functions of the inputs and the states of the model. They can be a scalar quantity or a vector of several parameters. The set of scheduling parameters define the *scheduling space* over which the LPV model is defined.

The linear system can be extended to contain offsets in the system's states, input, and output signals. Mathematically, the LPV system is represented by the following equations:

$$dx(t) = A(p)x(t) + B(p)u(t) + \left(\overline{dx}(p) - A(p)\bar{x}(p) - B(p)\bar{u}(p)\right)$$
$$y(t) = C(p)x(t) + D(p)u(t) + (\bar{y}(p) - C(p)\bar{x}(p) - D(p)\bar{u}(p))$$
$$x(0) = x_0$$

$\overline{dx}(p)$, $\bar{x}(p)$, $\bar{u}(p)$, $\bar{y}(p)$ are the offsets in the values of `dx(t)`, `x(t)`, `u(t)` and `y(t)` at a given parameter value `p = p(t)`.

LPV system can be thought of as a first-order approximation of a nonlinear system over a grid of scheduling parameter values. For example, you can linearize a Simulink model between a given input and output ports over a grid of equilibrium operating conditions. The values of the model inputs, outputs and state values at each operating point define the offsets, while the linear state-space models obtained by linearization define the state-space data. The LPV system thus generated can work as a proxy for the original model for facilitating faster simulations and control system design. For more information, see Linear Parameter-Varying Models.

The LPV System block accepts the state-space matrices and offsets over a grid of scheduling parameter values. The state-space matrices must be specified as an array of model objects. The `SamplingGrid` property of the array defines the scheduling parameters for the LPV system. For examples of using this block, see:

- Using LTI Arrays for Simulating Multi-Mode Dynamics
- Approximating Nonlinear Behavior using an Array of LTI Systems
- LPV Approximation of a Boost Converter Model

## Kalman Filter block for estimating states of linear time-invariant and linear time-varying systems

Use the Kalman Filter block to estimate the states of linear time-invariant and linear time-varying systems online. The states are estimated as new data becomes available during the operation of the system. The system can be continuous-time or discrete-time. You can generate code for this block using code generation products such as Simulink Coder.

You can access this block from the Control System Toolbox library. For an example of using this block, see State Estimation Using Time-Varying Kalman Filter.

## AnalysisPoint Control Design Block for Marking Points of Interest for Linear Analysis

The new `AnalysisPoint` block is a unit-gain Control Design Block that you can insert anywhere in a control system model to mark points of interest for linear analysis and tuning. Incorporate `AnalysisPoint` blocks into generalized state-space (`genss`) control system models by interconnecting them with numeric LTI models and other Control Design Blocks. When you mark a location in a control system model with an `AnalysisPoint` block, you can use that location for linear analysis tasks, such as extracting responses using `getIOTransfer` or `getLoopTransfer`. You can also use such locations to specify design requirements for control system tuning using `systune` or Control System Tuner (requires Robust Control Toolbox software).

For more information about using `AnalysisPoint` blocks, see:

- `AnalysisPoint` reference page
- Control System with Multi-Channel Analysis Points
- Managing Signals in Control System Analysis and Design

## Compatibility Considerations

`AnalysisPoint` replaces the `loopswitch` Control Design Block.

Models that contain `loopswitch` blocks continue to work, for backward compatibility. However, it is recommended that you use `AnalysisPoint` blocks in new models. If you have scripts or functions that use `loopswitch` blocks, consider updating them to use `AnalysisPoint` instead.

For documentation of `loopswitch`, see `loopswitch` in the R2014a documentation.

## pidtool function renamed to pidTuner

The `pidtool` function is now called `pidTuner`. To open PID Tuner, use the `pidTuner` command or, in the MATLAB desktop **Apps** tab, click **PID Tuner**.

Using `pidtool` does not generate an error in this release, but the function may be removed in a future release.

## Compatibility Considerations

If you have scripts that use `pidtool`, consider replacing those calls with `pidTuner`.

## getSwitches function renamed to getPoints

The `getSwitches` function is now called `getPoints` to match the renaming of `loopswitch` to `AnalysisPoint`. Using `getSwitches` does not generate an error in this release, but the function may be removed in a future release.

## Compatibility Considerations

If you have scripts or functions that use `getSwitches`, consider replacing those calls with `getPoints`.

## Functionality being removed or changed

| Functionality | What Happens When You Use This Functionality? | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `loopswitch` Control Design Block | Still works | `AnalysisPoint` | Consider replacing `loopswitch` with `AnalysisPoint` in scripts and functions. |
| `getSwitches` function | Returns `loopswitch` and `AnalysisPoint` blocks in model | `getPoints` | Consider replacing `getSwitches` with `getPoints` in scripts and functions. |
| `pidtool` function | Still works | `pidTuner` | Consider replacing `pidtool` with `pidTuner` in scripts. |

# R2014a

## Redesigned PID Tuner app for improved PID tuning workflow

The redesigned PID Tuner streamlines workflows for interactively tuning PID controllers for reference tracking and disturbance rejection.

To access the PID Tuner, use the `pidtool` command. For example, to tune a PI controller for an LTI model, `G`:

```
pidtool(G,'PI')
```

For more information about the PID Tuner, see Designing PID Controllers with the PID Tuner.

## PID controller tuning using system identification to model the plant from measured input-output data in the PID Tuner app (with System Identification Toolbox)

If you have System Identification Toolbox™ software, you can use PID Tuner to fit a linear model to the measured SISO response data from your system and tune a PID controller for the resulting model. For example, if you want to design a PID controller for a manufacturing process, you can start with response data from a bump test on your system.

PID Tuner uses system identification to estimate an LTI model from the response data. You can interactively adjust the identified parameters to obtain an LTI model with a response that fits your response data. PID Tuner automatically tunes a PID controller for the estimated model. You can then interactively adjust the performance of the tuned control system, and save the estimated plant and tuned controller.

For an example, see Interactively Estimate Plant Parameters from Response Data.

## freqsep function for decomposing a linear system into fast dynamics and slow dynamics

Use the new `freqsep` command for separating numeric LTI models into fast and slow components. `freqsep` allows you to specify the cutoff frequency about which the model is decomposed. The slow component contains poles with natural frequency below the cutoff frequency. The fast component contains poles at or above the cutoff.

For more information, see the `freqsep` reference page.

## damp command display includes time constant information

When you call the `damp` command with no output arguments, the display now includes the time constant for each pole. The time constant is calculated as follows:

$$\tau = \frac{1}{\omega_n \zeta}.$$

$\omega_n$ is the natural frequency of the pole, and $\zeta$ is its damping ratio.

## Compatibility Considerations

For a discrete-time system with unspecified sample time (`Ts = -1`), `damp` now calculates the natural frequency and damping ratio by assuming `Ts = 1`. Previously, the software returned `[]` for the natural frequency and damping ratio of such systems.

`damp` returns outputs in order of increasing natural frequency. Therefore, this change can result in reordered poles for systems with unspecified sample times.

For more information on the outputs, see the `damp` reference page.

# R2013b

## SamplingGrid property for tracking dependence of array of sampled models on variable values

In Control System Toolbox, you can derive arrays of numeric or generalized LTI models by sampling one or more independent variables. The new `SamplingGrid` property of LTI models tracks the variable values associated with each model in such an array.

Set this property to a structure whose fields are the names of the sampling variables and contain the sampled variable values associated with each model. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sys.SamplingGrid = struct('time',0:10)
```

For an additional examples, see:

- Array With Variations in Two Parameters
- Sample a Tunable (Parametric) Model for Parameter Studies

## Option to retain unconnected states when interconnecting models using connect command

By default, the `connect` command discards states that do not contribute to the dynamics in the path between the inputs and outputs of the interconnected system. You can now optionally retain such unconnected states. This option can be useful, for example, when you want to compute the interconnected system response from known initial state values of the components.

To instruct `connect` to retain unconnected states, use the new `connectOptions` command with the existing `connect` command.

For more information, see the `connectOptions` reference page.

## connect command always returns state-space or frequency response data model

The `connect` command now always returns a state-space model, such as an `ss`, `genss`, or `uss` model, unless one or more of the input models is a frequency response data model. In that case, `connect` returns a frequency response data model, such as an `frd` or `genfrd` model.

For more information, see the `connect` reference page.

## Compatibility Considerations

In previous releases, `connect` returned a `tf` or `zpk` model when all input models were `tf` or `zpk` models. Therefore, `connect` might now return state-space models in cases where it previously returned `tf` or `zpk` models.

## updateSystem command for updating dynamic system data in a response plot

The new `updateSystem` command replaces the system data used to compute a response plot with data derived from a different dynamic system, and updates the plot. `updateSystem` is useful, for example, to cause a plot in a GUI to update in response to interactive input.

For more information, see:

- `updateSystem` reference page
- Build GUI With Interactive Plot Updates

## getLoopID renamed to getSwitches

The `getLoopID` function is now called `getSwitches` to more clearly reflect the purpose of the function. Using `getLoopID` does not generate an error in this release, but the function may be removed in a future release.

## Compatibility Considerations

If you have scripts or functions that use `getLoopID`, consider replacing those calls with `getSwitches`.

## LoopID property of loopswitch renamed to Location

The `LoopID` property of the `loopswitch` model component is now called `Location` to more clearly reflect the purpose of the property. Using `LoopID` does not generate an error in this release, but the name may be removed in a future release.

## Compatibility Considerations

If you have scripts or functions that use the `LoopID` property, consider updating your code to use `Location` instead.

# R2013a

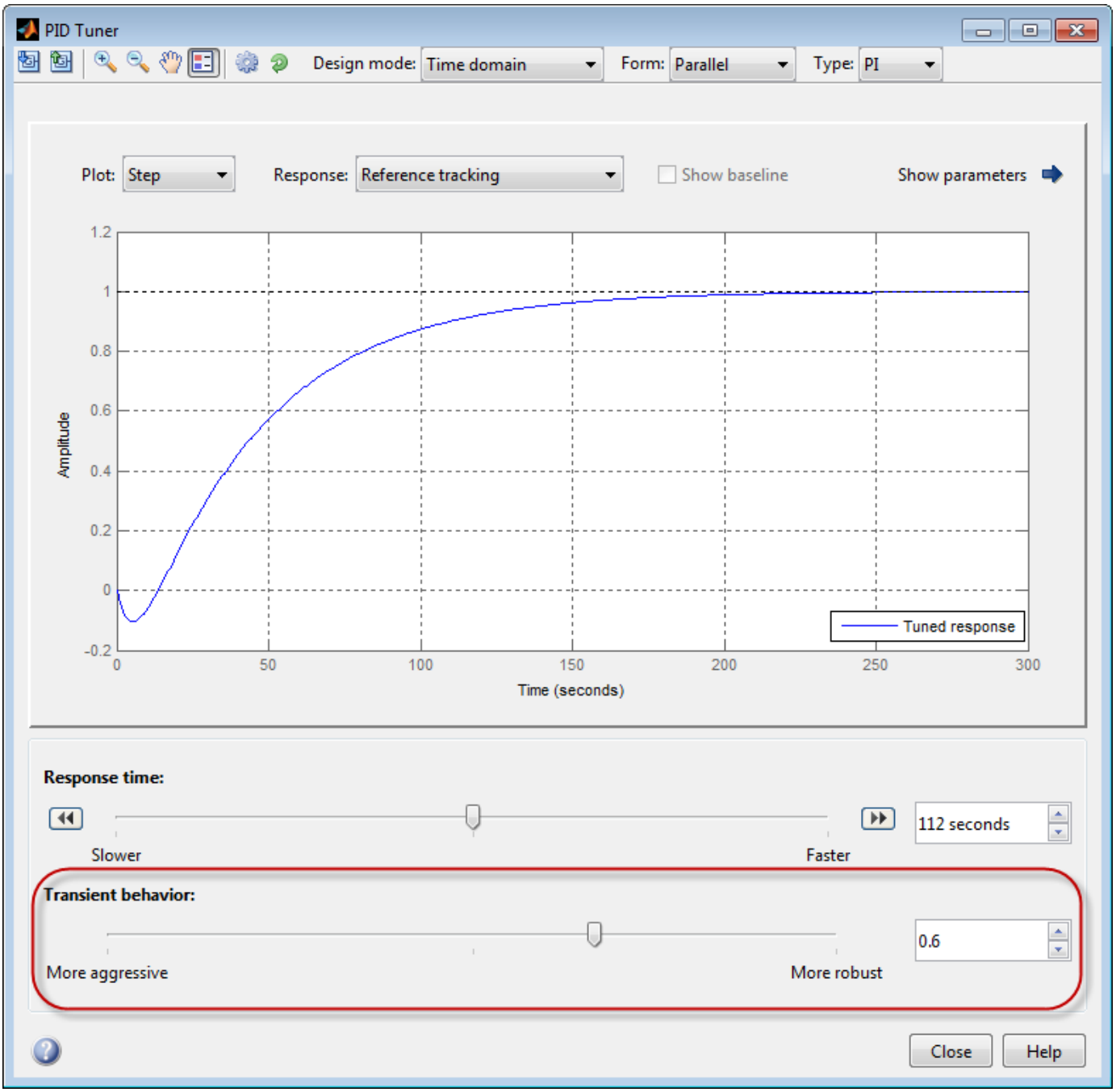
**Version: 9.5**

**New Features**

**Bug Fixes**

## Transient behavior slider for PID Tuner, increasing control over reference tracking and disturbance rejection performance

The PID Tuner now has a **Transient behavior** slider for emphasizing either reference tracking or disturbance rejection. When you open the PID Tuner, the tool starts in the `Time domain` design mode, displaying a step plot of the reference tracking response. The new **Transient behavior** slider is beneath the **Response time** slider.




You can use the **Transient behavior** slider when:

- The tuned system’s disturbance rejection response is too sluggish for your requirements. In this case, try moving the **Transient behavior** slider to the left to make the controller more aggressive at disturbance rejection.
- The tuned system’s reference tracking response has too much overshoot for your requirements. In this case, try moving the **Transient behavior** slider to the right to increase controller robustness and reduce overshoot.

In `Frequency domain` design mode, the PID Tuner has **Bandwidth** and **Phase margin** sliders. These sliders are the frequency-domain equivalents of the **Response time** and **Transient behavior** sliders, respectively.

# R2012b

## ltiblock.pid2 and loopswitch objects for tuning two-degree-of-freedom PID controllers and marking loop opening sites for open-loop requirements

New Control Design Blocks allow you to specify more control structures and more types of constraints for fixed-structure control system tuning in MATLAB:

- `ltiblock.pid2` — Tunable two-degree-of-freedom PID controller
- `loopswitch` — Control Design Block for specifying feedback loop opening locations in a tunable `genss` model of a control system

You can use these Control Design Blocks to build control systems for tuning with Robust Control Toolbox tuning commands such as `systune` and `looptune`. For more information, see the `ltiblock.pid2` and `loopswitch` reference pages.

## Commands for obtaining open-loop responses, closed-loop responses, and current values of tunable components from control system models

New commands allow you to compute open-loop and closed-loop responses from a Generalized LTI model representing a control system.

- `getLoopTransfer` — Compute point-to-point open-loop response of a Generalized LTI model of a control system, at a loop-opening site defined by a `loopswitch` block. The new command `getLoopID` returns a list of such loop-opening sites.
- `getIOTransfer` — Extract the closed-loop response from a specified input to a specified output of a control system.

These commands are particularly useful for validating the response functions of control systems tuned using Robust Control Toolbox tuning commands such as `systune`.

Additionally, the new `showTunable` command displays the current value of tunable components in a generalized LTI model of a control system. This command is useful for querying tuned parameter values of control systems tuned using Robust Control Toolbox tuning commands such as `systune`.

For more information, see the reference pages for these new commands and the following topics:

- Generalized Models
- Models with Tunable Coefficients

## Option for elementwise operation of model query commands on model arrays

The new `'elem'` flag causes elementwise operation on model arrays of the model query commands:

- `hasInternalDelay`
- `hasdelay`
- `isstatic`
- `isreal`

- `isfinite`
- `isproper`
- `isstable`

For example, for an array, `sysarray`, of dynamic system models,

```
B = hasdelay(sysarray,'elem');
```

returns a logical array. `B` of the same size as `sysarray` indicating whether the corresponding model in `sysarray` contains a time delay. Without the `'elem'` flag,

```
B = hasdelay(sysarray);
```

returns a scalar logical value that is equal to `1` if any entry in `sysarray` contains a time delay.

## Compatibility Considerations

`isfinite` and `isstable` now return a scalar logical value when invoked without the `'elem'` flag. Previously, `isfinite` and `isstable` returned a logical array by default.

If you have scripts or functions that use `isfinite(sysarray)` or `isstable(sysarray)`, replace those calls with `isfinite(sysarray,'elem')` or `isstable(sysarray,'elem')` to perform an elementwise query and obtain a logical array.

# R2012a

**Version: 9.3**

**New Features**

**Compatibility Considerations**

## Frequency Analysis Commands for Calculating Peak Gain and Finding Gain-Crossover Frequencies

Control System Toolbox software includes two new frequency analysis commands:

- `getPeakGain` — Peak gain of frequency response of a dynamic system model
- `getGainCrossover` — Frequencies at which system gain crosses a specified gain level

For more information, see the `getPeakGain` and `getGainCrossover` reference pages.

These functions use the SLICOT library of numerical algorithms. For more information about the SLICOT library, see http://slicot.org.

## Specify Target Crossover Frequency as Input to pidtune

A new syntax for `pidtune` lets you specify a target crossover frequency directly as an input argument. For example, the following command designs a PI controller, `C`, for a plant model `sys`. The command also specifies a target value `wc` for the 0 dB gain crossover frequency of the open-loop response `L = sys*C`.

```
C = pidtune(sys,'pi',wc);
```

Previously, you had to use `pidtuneOptions` to specify a target crossover frequency.

For more information, see the `pidtune` reference page.

## Rescaled Impulse Response and Impulse-Invariant Time Domain Conversion

For discrete-time dynamic system models, the input signal applied by `impulse` is now a unit area pulse of length `Ts` and height `1/Ts`. `Ts` is the sampling time of the discrete-time system. Previously, `impulse` applied a pulse of length `Ts` and unit height.

## Compatibility Considerations

Results of this change include:

- The amplitude of the impulse response calculated by `impulse` and `impulseplot` is scaled by `1/Ts` relative to previous versions.
- Discretization using the impulse-invariant (`'impulse'`) method of `c2d` returns a model that is scaled by `Ts` compared to previous releases. This scaling ensures a close match between the frequency responses of the continuous-time model and the impulse-invariant discretization as `Ts` approaches zero (for strictly proper models). In previous releases, the frequency responses differed by a factor of `Ts`.

## First-Order Hold Method for d2c

The `d2c` command now supports the first-order hold (FOH) method for converting a discrete-time dynamic system model to continuous time. The FOH method converts by performing linear interpolation of the inputs, assuming the control inputs are piecewise linear over the sampling period.

For more information about using this method, see the `d2c` reference page and Continuous-Discrete Conversion Methods.

## tzero Computes Invariant Zeros and Transmission Zeros

The `tzero` command computes the invariant zeros of SISO and MIMO dynamic system models. For minimal realizations, `tzero` computes transmission zeros. `tzero` also returns the normal rank of the transfer function of the system. For more information, see the `tzero` reference page.

## Models Created With System Identification Toolbox Can Be Used Directly With Control System Toolbox Functions

Identified linear models that you create using System Identification Toolbox software can now be used directly with Control System Toolbox analysis and compensator design commands. In prior releases, doing so required conversion to Control System Toolbox LTI model types.

Identified linear models include `idfrd`, `idss`, `idproc`, `idtf`, `idgrey` and `idpoly` models.

Identified linear models can be used directly with:

- Any Control System Toolbox or Robust Control Toolbox functions that operate on dynamic systems, including:
  - Response plots — `nichols`, `margin`, and `rlocus`
  - Model simplification — `pade`, `balred` and `minreal`
  - System interconnections — `series`, `parallel`, `feedback` and `connect`

  For a complete list of these functions, enter:

  ```
  methods('DynamicSystem')
  ```
- Analysis and design tools such as `ltiview`, `sisotool` and `pidtool`.
- The LTI System block in Simulink models.

## Functionality Being Removed or Changed

| Functionality | What Happens When You Use This Functionality? | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `impulse(sys)` and `impulseplot(sys)`, for discrete-time `sys` | Still works. | N/A | Amplitude of response is scaled by `1/Ts` compared to previous versions. `Ts` is sampling time of `sys`. |
| `c2d(sys,Ts,'impulse')` | Still works. | N/A | Resulting discretized model is scaled by `Ts` compared to previous releases. |

| Functionality | What Happens When You Use This Functionality? | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `[y,t] = impulse(sys,Tfinal)`<br>`[y,t] = step(sys,Tfinal)`<br>`[y,t,x] = initial(sys,Tfinal)` | For discrete-time `sys` with undefined sample time (`Ts=-1`), `Tfinal` is interpreted as the number of sampling periods to simulate. | N/A | Expect the number of simulation data points to be `Tfinal + 1` instead of `Tfinal`. |

# R2011b

**Version: 9.2**

**New Features**

**Compatibility Considerations**

## Formula-Based Specification of Summing Junctions and Vector Signal Naming for sumblk and connect

You can now use formula strings to specify the behavior of summing junctions with `sumblk`. For example, to create a summing junction, `S`, that takes the difference between signals `r` and `y` to produce signal `e`, enter the following command:

```
S = sumblk('e = r-y');
```

Additionally, both `sumblk` and `connect` now support vector-based signal naming for interconnecting multi-input, multi-output (MIMO) models. For more information, see the `sumblk` and `connect` reference pages.

## Commands for Interacting with Control Design Blocks in Generalized LTI Models

The following new commands allow you to examine and set the values of Control Design Blocks in Generalized LTI Models:

- `getValue` — Get nominal value of Generalized Model (replaces `getNominal`)
- `setValue` — Modify value of Control Design Block
- `getBlockValue` — Get nominal value of Control Design Block in Generalized Model
- `setBlockValue` — Set value of Control Design Block in Generalized Model
- `showBlockValue` — Display nominal values of Control Design Blocks in Generalized Model

For more information about these commands, see the reference pages for each command.

## Functionality Being Removed or Changed

| Functionality | What Happens When You Use This Functionality? | Use This Instead | Compatibility Considerations |
|---|---|---|---|
| `delay2z` | Errors | `absorbDelay` | Replace `delay2z` with `absorbDelay`. |
| `getNominal` | Errors | `getValue` | Replace `getNominal` with `getValue`. |
| `Scale` and `Info` properties of `realp` parameter | Errors | None | None |
| `sumblk('a','b','c','+-')` | Still works | `sumblk('a=b-c')` | Use new formula-based syntax for `sumblk`. |

# R2011a

**Version: 9.1**

**New Features**

**Compatibility Considerations**

## New Model Objects for Representing Tunable Parameters and Systems with Tunable Components

Control System Toolbox includes new model objects that you can use to represent systems with tunable components. You can use these models for parameter studies or controller synthesis using `hinfstruct`. The new model types include:

- Control Design Blocks—Parametric components that are the building blocks for constructing tunable models of control systems. Control Design Blocks include:
  - `realp`—Tunable real parameter
  - `ltiblock.gain`—Tunable static gain block
  - `ltiblock.tf`—Fixed-order SISO transfer function with tunable coefficients
  - `ltiblock.ss`—Fixed-order state-space model with tunable coefficients
  - `ltiblock.pid`—One-degree-of-freedom PID controller with tunable coefficients
- Generalized Matrices—Matrices that include parametric (tunable) values. Generalized matrices are `genmat` models.
- Generalized and Uncertain LTI Models—Models representing systems that have both fixed and tunable coefficients. Generalized LTI models include:
  - `genss`—Generalized state-space model
  - `genfrd`—Generalized frequency response data model

  These models arise from interconnections between numeric LTI models (such as `tf` , `ss`, or `frd`) and Control Design Blocks. You can also create `genss` models by using the `tf` or `ss` commands with one or more `realp` or `genmat` inputs.

This release also adds new functions for working with generalized models:

- `getNominal`—Nominal value of generalized model
- `replaceBlock`—Replace Control Design Blocks in generalized model
- `nblocks`—Number of blocks in generalized model
- `isParametric` — Determine if model has tunable blocks
- `getLFTModel`—Decompose generalized model

For more information about the new model types and about modeling systems that contain tunable coefficients, see the following in the *Control System Toolbox User's Guide*:

- Types of Model Objects
- Models with Tunable Coefficients

## New Time and Frequency Units for Models and Response Plots

All linear model objects now have a `TimeUnit` property for specifying unit of the time variable, time delays in continuous-time models, and sampling time in discrete-time models. The default time units is seconds. You can specify the time units, for example, as hours. See Specify Model Time Units for examples.

Frequency-response data ( `frd` and `genfrd`) models also have a new `FrequencyUnit` property for specifying units of the frequency vector. The default frequency units is `rad/TimeUnit`, where

`TimeUnit` is the system time units. You can specify the units, for example as kHz, independently of the system time units. See Specify Frequency Units of Frequency-Response Data Model for examples. If your code uses the `Units` property of frequency-response data models, it continues to work as before.

See the model reference pages for available time and frequency units options.

Changing the `TimeUnit` and `FrequencyUnit` properties changes the overall system behavior. If you want to simply change the time and frequency units without modifying system behavior, use `chgTimeUnit` and `chgFreqUnit`, respectively.

The time and frequency units of the model appear on the response plots by default. For multiple systems, the units of the first system are used. You can change the units of the time and frequency axes:

- Graphically, using the following editors:
  - Toolbox Preferences Editor
  - LTI Viewer Preferences Editor
  - Graphical Tuning Window Preferences Editor
  - Property Editor of individual plots
- Programmatically, by setting the following properties of plots:
  - `TimeUnits` for time-domain plots using `timeoptions`
  - `FreqUnits` for frequency-domain plots using, for example, `bodeoptions`

## Discrete-Time PID Controller Objects Have Stable Derivative Filter Pole

New requirements for creating `pid` and `pidstd` controller objects ensure that the derivative filter pole is always stable.

- For a discrete-time `pid` controller with a derivative filter (`Tf`≠ 0) and `Dformula` set to `'ForwardEuler'`, the sampling time `Ts` must be less than `2*Tf`.
- For a discrete-time `pidstd` controller with a derivative filter (`N`≠ `Inf`) and `Dformula` set to `'ForwardEuler'`, the sampling time `Ts` must be less than `2*Td/N`.
- The `Trapezoidal` value for `DFormula` is not available for a discrete-time `pid` or `pidstd` controller with no derivative filter (`Tf = 0` or `N = Inf`).

## Compatibility Considerations

On loading `pid` or `pidstd` controllers saved under previous versions, the software changes certain properties of controllers that do not have stable derivative filter poles.

- For a discrete-time `pid` controller with a derivative filter (`Tf`≠ 0), `Dformula` set to `'ForwardEuler'`, and sampling time `Ts` ≥ `2*Tf`, the derivative filter time is reset to `Tf = Ts`.
- For a discrete-time `pidstd` controller with a derivative filter (`N`≠ `Inf`), `Dformula` set to `'ForwardEuler'`, the sampling time `Ts` ≥ `2*Td/N`, the derivative filter constant is reset to `N = Td/Ts`.

- For a discrete-time `pid` or `pidstd` controller with no derivative filter and `DFormula = 'Trapezoidal'`, the derivative filter integrator formula is reset to `DFormula = 'ForwardEuler'`.

The software issues a warning when it changes any of these values. If you receive such a warning, validate your controller to ensure that the new values achieve the desired performance.

## New Variable q^–1 for Expressing Discrete-Time Transfer Functions

You can now express discrete-time `tf` and `zpk` models in terms of the inverse shift operator `q^-1`. The variable `q^-1` is equivalent to `z^-1`.

**Note** This new definition is consistent with the System Identification Toolbox definition of `q^-1`.

Use the new variable by setting the `Variable` property of a `tf` or `zpk` model to `q^-1`. For example, entering:

```
H = tf([1 2 3],[5 6 7],0.1,'Variable','q^-1')
```

creates the following discrete-time transfer function:

```
Transfer function:
1 + 2 q^-1 + 3 q^-2
-------------------
5 + 6 q^-1 + 7 q^-2

Sampling time (seconds): 0.1
```

When you set `Variable` to `q^-1`, `tf` interprets the numerator and denominator vectors as ascending powers of `q^-1`.

For more information, see the `tf` and `zpk` reference pages.

# R2010b

**Version: 9.0**

**New Features**

**Compatibility Considerations**

## New Commands and GUI for Modeling and Tuning PID Controllers

This release introduces specialized tools for modeling and designing PID controllers.

### PID Controller Design with the New PID Tuner GUI

The new PID Tuner GUI lets you interactively tune a PID controller for your required response characteristics. Using the GUI, you can adjust and analyze your controller's performance with response plots, such as reference tracking, load disturbance rejection, and controller effort, in both time and frequency domains.

The PID Tuner supports all types of SISO plant models, including:

- Continuous- or discrete-time plant models
- Stable, unstable, or integrating plant models
- Plant models that include I/O time delays or internal time delay

For more information about using PID Tuner, see:

- Designing PID Controllers in the *Control System Toolbox Getting Started Guide*
- The new demo `Designing PID for Disturbance Rejection with PID Tuner`

### PID Controller Design with the New pidtune Command

The new `pidtune` command lets you tune PID controller gains at the command line.

`pidtune` automatically tunes the PID gains to balance performance (response time) and robustness (stability margins). You can specify your own response time and phase margin targets using the new `pidtuneOptions` command.

`pidtune` supports all types of SISO plant models, including:

- Continuous- or discrete-time plant models.
- Stable, unstable, or integrating plant models.
- Plant models that include I/O time delays or internal time delays.
- Arrays of plant models. If `sys` is an array, `pidtune` designs a separate controller for each plant in the array.

For additional information, see:

- The `pidtune` and `pidtuneOptions` reference pages
- The new Control System Toolbox demo `Designing Cascade Control System with PI Controllers`

### Modeling PID Controllers in Parallel Form or Standard Form

The new LTI model objects `pid` and `pidstd` are specialized for modeling PID controllers.

With `pid` and `pidstd` you can model a PID controller directly with the PID parameters, expressed in parallel (`pid`) or standard (`pidstd`) form. The `pid` and `pidstd` commands can also convert to PID form any type of LTI object that represents a PID controller.

Previously, to model a PID controller, you had to derive the controller's equivalent transfer function (or other model), and could not directly store the PID parameters.

For additional information, see the `pid` and `pidstd` reference pages

## Improved PID Tuning Options in SISO Design Tool

This release includes improvements to the PID Tuning options in the Automated Tuning pane of SISO Design Tool.

In addition to the Robust Response Time tuning algorithm, SISO Design Tool offers a collection of classical design formulas, including the following:

- Approximate *M*-Constrained Integral Gain Optimization (MIGO) Frequency Response
- Approximate MIGO Step Response
- Chien-Hrones-Reswick
- Skogestad Internal Model Control (IMC)
- Ziegler-Nichols Frequency Response
- Ziegler-Nichols Step Response

For information about using SISO Design Tool, see SISO Design Tool in the *Control System Toolbox User's Guide*. For specific information about the automatic PID Tuning options in SISO Design Tool, see PID Tuning in the *Control System Toolbox User's Guide*.

## Ability to Analyze a Controller Design for Multiple Models Simultaneously in SISO Design Tool

You can now analyze a controller design for multiple models simultaneously using the SISO Design Tool. This feature helps you analyze whether the controller satisfies design requirements on a system whose exact dynamics are not known and may vary.

System dynamics can vary because of parameter variations or different operating conditions. You represent variations in system dynamics of the plant (`G`), sensor (`H`), or both in a feedback structure using arrays of LTI models. Then, design a controller for a nominal model in the array and analyze that the controller satisfies the design requirements on the remaining models using the design and analysis plots. For more information, see:

- Control Design Analysis of Multiple Models.
- `Compensator Design for a Set of Plant Models` demo.
- `Reference Tracking of a DC Motor with Parameter Variations` demo in Simulink Control Design software.

## Change in Output of repsys Command

The output of the `repsys` command when called with a single dimension argument has changed.

In prior versions, the output of `repsys(sys,N)` was the same as that of `append(sys,...,sys)`.

Now, `repsys(sys,N)` returns the same result as `repsys(sys,[N N])`.

The results of other syntaxes for `repsys` have not changed.

See the `repsys` and `append` reference pages for more information.

## Compatibility Considerations

Code that depends upon the previous result of `repsys(sys,N)` no longer returns that result. To obtain the previous result, replace `repsys(sys,N)` with `sys*eye(N)`.

# R2010a

**Version: 8.5**

**New Features**

**Compatibility Considerations**

## Enhanced c2d Command to Approximate Fractional Time Delays in Tustin and Matched Discretization Methods

The `c2d` command can now approximate fractional time delays when discretizing linear models with the `tustin` or `matched` methods. The new `c2dOptions` command lets you specify an optional Thiran all-pass filter. The Thiran filter approximates fractional delays for improved phase matching between continuous and discretized models. Previously, `c2d` rounded fractional time delays to the nearest multiple of the sampling time when using the `tustin` or `matched` methods. For more information, see the `c2d` and `c2dOptions` reference pages and Continuous-Discrete Conversion Methods in the *Control System Toolbox User Guide*.

## New Commands for Specifying Options for Continuous-Discrete Conversions

New commands `c2dOptions`, `d2dOptions`, and `d2cOptions` make it easier to specify options for

- Discretization using `c2d`
- Resampling using `d2d`.
- Conversion from discrete to continuous time using `d2c`.

## Compatibility Considerations

This release deprecates the `prewarp` method for `c2d`, `d2d`, and `d2c`. Instead, use `c2dOptions`, `d2dOptions`, or `d2cOptions` to specify the `tustin` method and a prewarp frequency. For more information, see Continuous-Discrete Conversion Methods and the `c2d`, `d2d`, and `d2c` reference pages.

## New FDEL Command to Remove Specified Data from Frequency Response Data (FRD) Models

You can now remove selected data from `frd` models using the new `fdel` command. For example, use `fdel` to:

- Remove spurious or unneeded data from `frd` models you create from measured frequency response data.
- Remove data at intersecting frequencies from `frd` models before merging them into a single `frd` model with `fcat`, which can only merge `frd` models containing no common frequencies.

For more information, see `fdel` reference page.

# R2009b

**Version: 8.4**

**New Features**

## Ability to Design Compensators for New Types of Plants

In the SISO Design Tool, you can now design compensators for plants models that:

- Contain time delays

  Previously, you had to approximate delays before designing compensators.
- You specify as frequency-response data (FRD)

For more information on designing compensators using the SISO Design Tool, see SISO Design Tool.

## New Automated PID Tuning Method

You can now tune compensators using a new automated PID tuning algorithm called Robust Response Time, which is available in the SISO Design Tool. You specify the open-loop bandwidth and phase margin, and the software computes PID parameters to robustly stabilize your system.

For information on tuning compensators using automated tuning methods, see Automated Tuning.

# R2009a

## Variable q Now Defined as the Forward Shift Operator z

The variable $q$ is now defined in the standard way as the forward shift operator $z$. Previously, $q$ was defined as $z^{-1}$.

**Note** This new definition is consistent with the System Identification Toolbox definition of $q$.

## Compatibility Considerations

If you use the $q$ variable, you may receive different results than in previous releases when you:

- Create a transfer function
- Modify the `num` or `den` properties of an existing transfer function

The resulting transfer function differs from previous releases when both the

- `Variable` property is set to $q$
- `num` and `den` properties have different lengths

For example, the following code:

```
H = tf([1,2],[1 3 8],0.1,'Variable','q')
```

now returns the transfer function

$$\frac{q+2}{q^2+3q+8} \equiv \frac{z+2}{z^2+3z+8}$$

Previously, the code returned the transfer function

$$\frac{1+2q}{1+3q+8q^2} \equiv \frac{1+2z^{-1}}{1+3z^{-1}+8z^{-2}} \equiv \frac{z^2+2z}{z^2+3z+8}$$

The two transfer functions have different numerators.

# R2008b

## New Design Tools for Linear-Quadratic-Gaussian (LQG) Servo Controllers with Integral Action

You can now design a Linear-Quadratic-Gaussian (LQG) servo controller for set-point tracking using the new `lqi` and `lqgtrack` commands. This compensator ensures that the system output tracks the reference command and rejects process disturbances and measurement noise.

For more information on forming LQG servo controllers, see Linear-Quadratic-Gaussian (LQG) Design, the `lqi` reference page, and the `lqgtrack` reference page.

#### Current Flag Moved from lqgreg to kalman

The `'current'` flag was moved from the `lqgreg` function to the `kalman` function.

## Compatibility Considerations

The following code:

```
kest = kalman(sys,Qn,Rn)
c = lqgreg(kest,k)
```

now returns the current regulator $u[n] = -K\hat{x}[n|n]$ instead of the delayed regulator $u[n] = -K\hat{x}[n|n-1]$.

To update your code to return the same results as in previous releases, use the following code with the added string `'delayed'` in the `kalman` command:

```
kest = kalman(sys,Qn,Rn,'delayed')
c = lqgreg(kest,k)
```

For information on using these functions with the current flag in the `kalman` function, see the `kalman` and `lqgreg` reference pages.

## New Upsampling Method for Rate Conversion in Discrete-Time Models

You can now upsample a discrete-time system to an integer multiple of the original sampling rate without any distortion in the time or frequency domain using the `upsample` command.

For more information on upsampling, see the `upsample` reference page and Upsample a Discrete-Time System in the *Control System Toolbox User's Guide*.

## New Scaling Tools to Enhance the Accuracy of Computations with State-Space Models

You can now scale state-space models to maximize accuracy over the frequency band of interest using the `prescale` command and associated GUI. Use this functionality when you cannot achieve good accuracy at all frequencies and some tradeoff is necessary. A warning alerts you when accuracy may be poor and using prescaling is recommended.

For more information on setting the frequency band for scaling state-space realizations, see Scaling State-Space Models and the `prescale` reference page.

## New Command to Reorder the States of State-Space Models

You can now reorder the states of state-space models according to a specified permutation using the `xperm` command.

For more information on reordering states, see the `xperm` reference page.

## Enhanced Support for Customizing Response Plots

You can now make the following changes to your Control System Toolbox response plots using the figure plotting tools:

- System name
- Line color
- Line style
- Line width
- Marker type

For more information on customizing the appearance of response plots using plot tools, see Customizing Response Plots Using Plot Tools in the *Control System Toolbox User's Guide*.

# R2008a

**Version: 8.1**

**New Features**

## Updated Error and Warning Message System

The Control System Toolbox error and warning IDs and messages have been updated. If you use error and warning IDs in your code, you must update your code to reflect the new IDs.

# R2007b

**Version: 8.0.1**

**New Features**

## Updated and Expanded Demos

The Control System Toolbox demos have been reformatted and expanded to include more examples and content. Demos in the following categories now have new and improved content:

- Getting Started with LTI Models
- Discretization and Sampling Rate Conversions
- How to Get Accurate Results

To open the Control System Toolbox demos, type

```
demo toolbox control
```

at the MATLAB prompt.

# R2007a

**Version: 8.0**

**New Features**

## Analysis of Time Delay Systems Now Fully Supported

Control System Toolbox software now lets you:

- Model, simulate, and analyze any interconnection of linear systems with delays, such as systems containing feedback loops with delays.
- Exactly analyze and simulate control systems with long delays. You can evaluate control strategies, such as Smith Predictor and PID control for first-order-plus-dead-time plants.
- Use new commands for modeling state-space models with delays including: `delayss`, `getDelayModel`, and `setDelayModel`.

For more information, see the section on Models with Time Delays.

## New and Updated Automated Tuning Methods

Control System Toolbox software now provides the following new and updated automated tuning methods:

- New Singular Frequency Based Tuning lets you design PID compensators for both stable and unstable plants.
- New H-infinity Loop Shaping lets you find compensators based on a desired open-loop bandwidth or loop shape. This feature requires Robust Control Toolbox software.
- Updated Internal Model Control (IMC) Tuning now supports unstable plants.

For more information, see the section on automated tuning in the Control System Toolbox documentation.

## New Tustin and Prewarp Options for d2d Function

The `d2d` function now includes the following new options for the resampling method:

- `'tustin'`—Performs Bilinear (Tustin) approximation
- `'prewarp'`—Performs Tustin approximation with frequency prewarping

For more information, see the `d2d` reference pages.

# R2006b

**Version: 7.1**

**New Features**

## New Loop Configurations in the SISO Design Tool

Two new loop configurations are available from the SISO Design Tool. See Modifying Block Diagram Structure for more information.

## New Design Requirements

The LTI Viewer now supports step response and upper/lower time bound design requirements. See Adding Design Requirements to the LTI Viewer for more information.

# R2006a

**Version: 7.0**

**New Features**

## SISO Design Tool

The SISO Design Tool now provides one-click automated tuning using systematic algorithms such as Ziegler-Nichols PID tuning, IMC design, and LQG design. In addition, you can calculate low-order approximations of the IMC/LQG compensators to keep the control system complexity low.

### Compensator Optimization Is Now Supported

If you have installed Simulink Response Optimization™ software, you can now optimize the compensator parameters inside the SISO Design Tool GUI. You can specify time- and frequency-domain requirements on SISO Design Tool plots such as `bode` and `step`, and use numerical optimization algorithms to automatically tune your compensator to meet your requirements. See the Simulink Response Optimization documentation for more details.

### Improved Compensator Editor

The Compensator Editor used to edit the numerical values of poles and zeros has been upgraded to better handle common control components such as lead/lag and notch filters.

### Multi-Loop Compensator Design Support

Many control systems involve multiple feedback loops, some of which are coupled and need joint tuning. The SISO Design Tool now lets you analyze and tune multi-loop configurations. You can focus on a specific loop by opening signals to remove the effects of other loops, gain insight into loop interactions, and jointly tune several SISO loops.

### SISO Design Tool Fully Integrated with the Controls & Estimation Tools Manager

To improve workflow and better leverage other tools, such as Simulink Control Design software and Simulink Response Optimization software, the SISO Design Tool is now fully integrated with the Controls & Estimation Tools Manager (CETM). This provides a signal environment for the design and tuning of compensators.

When you open the SISO Design Tool, the CETM also opens with a SISO Design Task. Many SISO Design Tool features, such as importing models, changing loop configurations, etc., have been moved to the SISO Design Task in CETM. In addition, related tasks such as Simulink based Tuning and Compensator Optimization are seamlessly integrated with the SISO Design Task. See the *Control System Toolbox Getting Started Guide* for details on the new work flow.

## LTI Viewer Enhancements

The LTI Viewer now lets you plot the response of a system to user-defined input signals (`lsim`) and initial conditions (`initial`). A new GUI lets you select input signals from a signal generator library, or import signal data from a variety of file formats.

## LTI Objects

### Descriptor and Improper State-Space Models Fully Supported

There is now full support for descriptor state-space models with a singular `E` matrix. This now lets you build state-space representations, such as PID, and manipulate improper models with the superior accuracy of state-space computations. In previous versions, only descriptor models with a nonsingular `E` matrix were supported.

### New Commands to Calculate Time Response Metrics

The new `stepinfo` and `lsiminfo` commands compute time-domain performance metrics, such as rise time, settling time, and overshoot. You can use these commands to write scripts that automatically verify or optimize such performance requirements. Previously, these metrics were available only from response plots.

### Simplified System Interconnections Using I/O Channel Names

The commands `connect`, `feedback`, `series`, `parallel`, and `lft` now let you connect systems by matching names of I/O channels. A helper function, `sumblk`, has also been added to simplify the specification of summing junctions. Altogether this considerably simplifies the task of deriving models for complicated block diagrams. In previous releases, only index-based system connection was supported.

### Changes in the Representation of I/O Delays in State-Space Models

The `ioDelay` property is deprecated from state-space models. Instead, these models have a new property called `InternalDelay` for logging all delays that cannot be pushed to the inputs or outputs. Driving this change is the switch to a representation of delays in terms of delayed differential equations rather than frequency response. See Models with Time Delays for more details on internal delays, and `ss/getdelaymodel` for details on the new internal representation of state-space models with delays.

### New Name Property for LTI Objects

This new property lets you attach a name (string) to a given LTI model. The specified name is reflected in response plots.

### New Commands and Operations for LTI Objects

The new `exp` command simplifies the creations of continuous-time transfer functions with delays. For more information, type help `lti/exp` at the MATLAB prompt.

The `frd` object has the following new methods:

- `fcat` — Concatenates one or more FRD models along the frequency dimension (data merge).
- `fselect` — Selects frequency points or range in `frd` model.
- `fnorm` — Calculates pointwise peak gain of `frd` model.

The `.*` operation is supported for transfer functions and zero-pole-gain objects. This allows you to perform element-by-element multiplication of MIMO models.

## Numerical Algorithms

There have been several major improvements in the Control System Toolbox numerical algorithms, many of which benefit the upgraded SISO Design Tool:

- New scaling algorithm that maximizes accuracy for badly scaled state-space models
- Performance improvement in time and frequency response computations through MEX-files
- More accurate computations of the zero-pole-gain and transfer function representations of a state-space model

- More accurate state-space representations of zero-pole-gain models
- Better handling of nonminimal modes in model reduction commands (`balred`, `balreal`)
- `canon` now computes a block modal form for `A` matrices that are not diagonizable or are nearly defective
- Exact phase computation for zero-pole-gain models in `bode` and `nichols`
- Accurate handling of improper models using the descriptor state-space representation

# R14SP3

**Version: 6.2.1**

**No New Features or Changes**

# R14SP2

**Version: 6.2**

**New Features**

## Command-Line API for Customizing Plots

The Control System Toolbox software now provides a command-line API for customizing units, labels, limits, and other plot options. You can now change default plot options before generating a plot, or modify plot properties after creation.

For a detailed description of the commands, see the Control System Toolbox documentation.

## Constraint Types for SISO Design

You can now create

- Single piecewise linear constraints for root-locus and Bode plots
- Gain/phase exclusion regions for Nichols plots

Design constraints are displayed as shaded regions.

## Bode and Nichols Plots Have Additional Options

When editing Bode and Nichols plots, you can now

- Set the lower limit of the magnitude manually.
- Adjust the phase offsets by multiples of 360 degrees to facilitate comparing multiple responses.

## Model-Approximation and Order-Reduction Commands

New commands have been added for model approximation and order reduction:

- `hsvd` computes and plots the Hankel singular values.
- `balred` computes low-order approximations using a numerically stable, balancing-free algorithm. You can perform multiple order reductions with a single command.