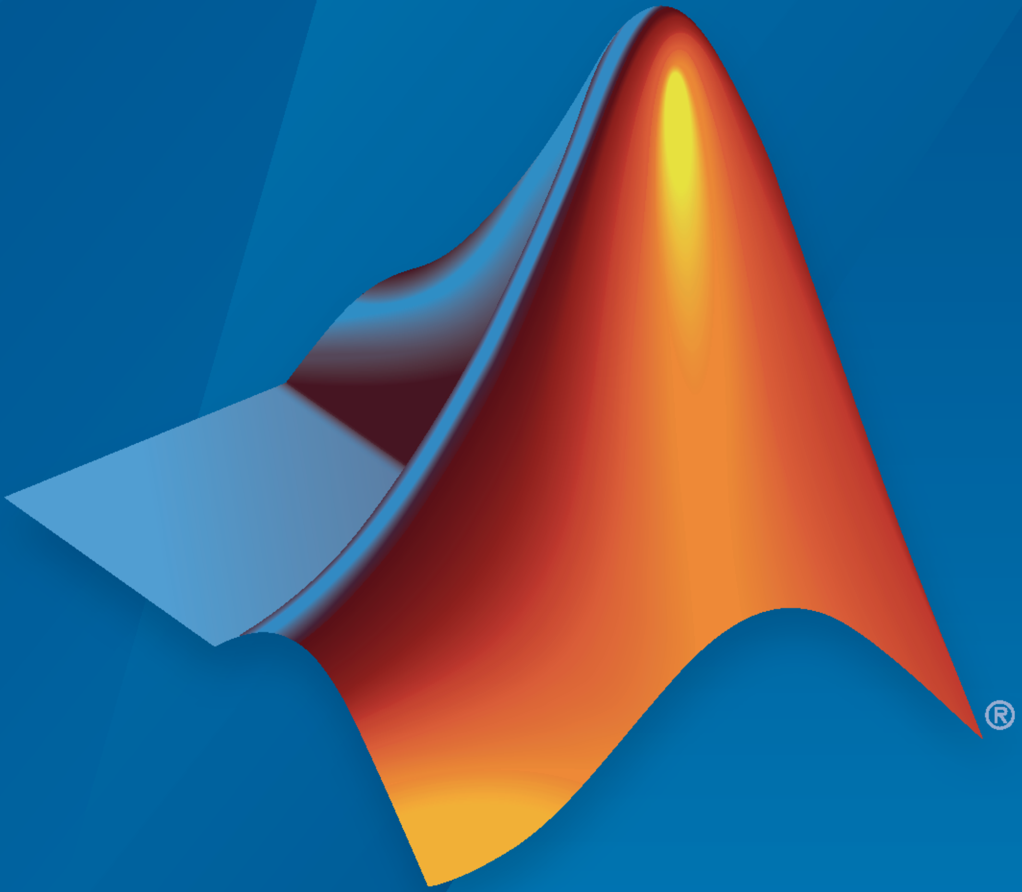
Robust Control Toolbox™ Release Notes



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

Robust Control Toolbox™ Release Notes

© COPYRIGHT 2005–2021 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Contents

R2021b

R2021a

R2020b

R2020a

R2019b

R2019a

R2018b

R2018a

Bug Fixes

R2017b

R2017a

R2016b

R2016a

R2015b

R2015a

R2014b

R2014a

R2013b

R2013a

R2012b

R2012a

R2011b

R2011a

R2010b

R2010a

Bug Fixes

R2009b

R2009a

Bug Fixes

R2008b

Bug Fixes

R2008a

R2007b

No New Features or Changes

R2007a

R2006b

R2006a

No New Features or Changes

R14SP3

No New Features or Changes

R14SP2

R2021b

Version: 6.11

New Features

Bug Fixes

Compatibility Considerations

loopsyn Command: Balance performance and robustness when designing controllers by loop shaping

The `loopsyn` command now blends two loop-shaping methods:

- Mixed-sensitivity design (`mixsyn`), which tends to optimize performance and decoupling at the expense of robustness
- The Glover-McFarlane method (`ncfsyn`), which maximizes robustness to plant uncertainty

You can adjust the tradeoff between performance and robustness to obtain satisfactory time responses while avoiding fragile designs with plant inversion or flexible-mode cancellation. You can also optionally specify the controller order. Previously, `loopsyn` computed an optimal H_∞ controller for the target loop shape without regard to robustness.

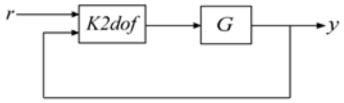
For an example illustrating the new capabilities, see “Loop-Shaping Controller Design”. For further information, see `loopsyn`.

Compatibility Considerations

Due to the change in the default behavior of `loopsyn`, you might obtain different results from previous releases. Additionally, you can no longer specify a frequency range for loop shaping using the syntax `loopsyn(G,Gd,[wmin,wmax])`. The `loopsyn` command ignores any `[wmin,wmax]` input. The `info` output structure no longer contains a `Range` field.

loopsyn Command: Synthesize controllers for two-degree-of-freedom architecture

In addition to the standard controller, `loopsyn` now returns a controller for a two-degree-of-freedom control structure, as shown in the following diagram.



This architecture can be useful for mitigating the derivative kick that can occur when the reference signal changes. The controller is returned in the `K2dof` field of the `info` output argument. For more information, see `loopsyn`.

ncfsyn Command: Adjust tolerance to help eliminate fast controller dynamics

When `ncfsyn` returns a controller with undesirable fast dynamics, you can now try increasing the tolerance to eliminate them. The tolerance specifies how tightly the performance achieved by the synthesized controller approximates the optimal performance. To adjust the tolerance, use the new syntax `[K,CL,gamma,info] = ncfsyn(G,W1,W2,tol)`. For more information, see `ncfsyn`.

ncfmr Command: Reduce model order fully programmatically

The `ncfmr` command performs coprime-factor-based model reduction. This type of model reduction is useful for simplifying controllers in feedback loops, because the reduced-order controller preserves stability provided that the approximation error is smaller than the robustness margin computed by `ncfmargin`.

`nfcmr` now allows you to extract Hankel singular values and error bounds without the interactive step of entering a model order. You can use this information to select the order of the reduced model programmatically. Previously, you had to use an interactive workflow to obtain the Hankel singular values and select a reduced model order, making programmatic selection difficult. In addition, the function previously did not return a bound on the approximation error.

These changes bring the syntaxes of `ncfmr` in line with those of the balanced-truncation model-reduction command `balred`. See `ncfmr` for information and examples about the recommended ways of using this command.

Compatibility Considerations

Because of these changes to `ncfmr`, some syntaxes and output arguments have changed, and some options are not recommended. See “ncfmr syntax and output change” on page 1-3.

Functionality being removed or changed

loopsyn ignores frequency range

Behavior change

`loopsyn` now balances performance and robustness. As a result of this change, `loopsyn` might return different results from previous releases. Additionally, you can no longer specify a frequency range for loop shaping using the syntax `loopsyn(G,Gd,[wmin,wmax])`. The `loopsyn` command ignores any `[wmin,wmax]` input. Further, the `info` output structure no longer contains a `Range` field. For information about the improvements to `loopsyn`, see “loopsyn Command: Balance performance and robustness when designing controllers by loop shaping” on page 1-2.

ncfmr syntax and output change

Behavior change

The behavior of `ncfmr` changed in R2021b to bring it in line with `balred` and to allow fully programmatic selection of the reduced-model order. (See “ncfmr Command: Reduce model order fully programmatically” on page 1-2.) As a result, some syntaxes and output arguments have changed, and some options are no longer recommended.

Prior to R2021b, calling `ncfmr(G)` without a specified reduction order invoked an interactive workflow that prompted you to enter the desired order. As of R2021b, this syntax displays a plot but no longer prompts for the desired order. Instead, call `ncfmr` again using the `ord` input argument to specify the desired order. The following table further describes this change in the interactive workflow.

Interactive Workflow Before R2021b	Interactive Workflow in R2021b
1 Enter `Gred = ncfmr(G)`. The software produces a Hankel singular-value plot and prompts you to enter the desired reduction order. **2** Examine the plot to determine the desired order based on the Hankel singular values. **3** Enter the desired order. The software computes the reduced-order model `Gred`.	**1** Enter `ncfmr(G)` without an output argument. The software produces a Hankel singular-value plot that also shows the approximation error of truncation at each mode. **2** Examine the plot to determine the desired order based on the Hankel singular values or the approximation error. **3** Call `ncfmr` again by entering `Gred = ncfmr(G,ord)`. The software computes the reduced-order model `Gred`.

The fields of the `info` output argument of `ncfmr` have also changed in R2021b. The following table summarizes these changes.

`info` Fields Before R2021b	`info` Fields in R2021b
• `info.GL` — Left normalized coprime factorization • `info.GR` — Right normalized coprime factorization • `info.hsv` — Hankel singular values	• `info.GL` — Left normalized coprime factorization • `info.HSV` — Hankel singular values of `info.GL` • `info.ErrorBound` — Bound on approximation error

Additionally, the `'MaxError'` and `'Display'` options are not recommended. Instead:

- To display a plot of the Hankel singular values of a dynamic system model `G`, call `nfcmr(G)` without an output argument. This command generates a singular-value plot that also displays the approximation error of truncation at each mode.
- To select a model order based on the approximation error, examine the plot. Or, call `[~,info] = ncfmr(G)` and use the information in `info.ErrorBound`.

For additional information about `ncfmr` syntaxes and arguments, see `ncfmr`.

R2021a

diskmarginplot Command: Visualize symmetric gain margins using simpler syntax

The syntax `diskmarginplot(DGM)` now accepts a scalar `DGM`. Use this syntax to visualize the range of simultaneous gain and phase variations corresponding to a disk-based gain margin of `[1/DGM,DGM]`. This symmetric gain margin represents gain that can increase or decrease by a factor of `DGM`.

Previously, to visualize variations corresponding to a symmetric margin, you had to explicitly provide the two-element vector `[1/DGM,DGM]`.

For more information about visualizing disk-based stability margins, see `diskmarginplot`.

R2020b

ncfsyn Command: Compute loop-shaping controllers in a more numerically stable way

Improvements to `ncfsyn` allow direct computation of Glover-McFarlane H_∞ loop-shaping controllers for:

- Discrete-time plants
- Plants with complex-valued state-space matrices

Previously, `ncfsyn` did not support plants with complex-valued matrices. Also, `ncfsyn` previously converted discrete-time plants to continuous time before performing H_∞ synthesis. With direct computation instead of conversion, the new algorithm is more stable numerically.

Additionally, the `info` output argument of `ncfsyn` now includes the optimal H_∞ performance in the field `info.gopt`. This value is the optimal performance achievable by H_∞ synthesis for the shaped plant. For numerical reasons, `ncfsyn` generally returns a controller with slightly larger H_∞ performance.

Compatibility Considerations

The syntax `ncfsyn(G,W1,W2,'ref')` is not recommended.

Functionality being removed or changed

umargin property Eccentricity renamed to Skew

Still runs

The `Eccenticity` property of the `umargin` control design block has been renamed to `Skew`. This property controls the bias of the modeled uncertainty disk toward gain increase or decrease. For more information, see the Skew property on the `umargin` reference page.

Compatibility Considerations

If your code uses this property, consider modifying it to use the new property name.

ncfsyn reference-command syntax is not recommended

Still runs

The syntax `ncfsyn(G,W1,W2,'ref')` is not recommended.

R2020a

Version: 6.8

New Features

Bug Fixes

Compatibility Considerations

umargin uncertain block: Model gain and phase variations and enforce stability margins for robust controller design

The new `umargin` control design block lets you model gain and phase variations in feedback loops. Modeling gain and phase variations in your system lets you verify stability margins during robustness analysis or enforce them during robust controller design.

To add gain and phase uncertainty to a feedback loop, you create a `umargin` block by specifying the limits for the gain and phase uncertainty of the loop. Then, incorporate the block into an uncertain state-space (`uss`) model of the closed-loop system.

For more information, see:

- `umargin`
- Uncertain Gain and Phase
- Model Gain and Phase Uncertainty in Feedback Loops

diskmarginplot and wcdiskmarginplot: Visualize disk-based stability margins

Use the new `diskmarginplot` and `wcdiskmarginplot` commands to visualized disk-based stability margins.

`diskmarginplot` provides several visualizations, including:

- Disk-based gain and phase margins of a dynamic system as a function of frequency
- Ranges of simultaneous gain and phase variations corresponding to specific disk margins
- The disk of values taken by the uncertainty model used to represent gain and phase variations

`wcdiskmarginplot` lets you visualize the variation of disk-based gain and phase margins for uncertain systems. The function plots the nominal and worst-case margins across frequency. The plot also includes margins of random samples of the system to represent roughly the envelope of margins.

For details about interpreting disk margins, see Stability Analysis Using Disk Margins.

Compatibility Considerations

`diskmarginplot` supersedes `dmplot`, which is not recommended.

diskmargin and wcdiskmargin: Compute smallest destabilizing gain or phase perturbations

The `diskmargin` and `wcdiskmargin` commands now return the smallest gain and phase variations that drive the feedback loop unstable. This variation is expressed as a diagonal perturbation `F = diag(f1,...,fN)`, where `fj` is a dynamic system representing the destabilizing gain and phase variation applied to each channel of the feedback loop.

You can access the destabilizing gain and phase variation in the new `WorstPerturbation` field of the output structures returned by `diskmargin` and `wcdiskmargin`. For details, see the `diskmargin` and `wcdiskmargin` reference pages.

uscale Command: Scale the uncertainty of an uncertain block or system

Use the new `uscale` command to scale the amount of uncertainty of an uncertain block or uncertain system by a numeric factor. This scaling can be useful in interpreting the results of robustness analysis with functions like `robstab`, `robgain`, or `musynperf`. For instance, suppose that you analyze an uncertain system with `robstab` and learn that the system can tolerate only 80% of the modeled uncertainty before going unstable. Use `uscale` to scale the uncertainty in the system by that amount and examine the actual safe ranges of system uncertainty.

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

musyn Command: Synthesize discrete-time robust controllers

The `musyn` command, introduced in R2019b, can now design robust controllers for discrete-time and continuous-time uncertain systems.

Compatibility Considerations

`musyn`, `musynOptions`, and `musynperf` supersede `dksyn`, `dksynOptions`, and `dksynperf`, which are not recommended. See “dksyn is not recommended” on page 4-4.

Functionality being removed or changed

Sampling of ureal elements now uniform in actual values

Behavior change

`usample` now uniformly samples the actual uncertainty range of `ureal` objects. Previously, `usample` first normalized the uncertain element, and then sampled uniformly in the normalized range. As a result of this change, you might obtain different results when you use `usample` to sample `ureal` elements or `uss` models that contain them, even if you use the same random seed.

The new implementation yields more uniform sampling for `ureal` parameters with skewed ranges (nominal value closer to one end of the range than the other). However, highly skewed ranges can lead to poor numeric conditioning and poor results. Therefore, for meaningful results, avoid highly skewed ranges where the nominal value is orders of magnitude closer to one end of the range than to the other.

Default value of ultidyn property SampleStateDimension changed

Behavior change

The default value of the `ultidyn` property `SampleStateDimension` is now 3. Previously, the default value was 1.

`SampleStateDimension` sets the number of states in random samples of uncertain dynamics taken with analysis commands such as `usample` and `bode`. With `SampleStateDimension` = 1, all Nyquist plots of sampled dynamics touch the gain bound at either (–1,0) (frequency = 0) or (1,0) (frequency = `Inf`). Higher `SampleStateDimension` yields points of contact at other frequencies, meaning better coverage of worst-case possibilities. (The odds of hitting a worst-case value by random sampling is still very low. You can use sampling to get a rough idea of the effects of uncertainty, but for rigorous worst-case analysis, use commands such as `wcgain` and `wcdiskmargin`.) For an example of the effect of `SampleStateDimension`, see Generate Samples of Uncertain Systems.

If you have code that relies on the default value of `SampleStateDimension` being 1, update your code to explicitly set the property.

diskmargin and wcdiskmargin: Disk-based gain-margin range can include negative gains

Behavior change

The `diskmargin` and `wcdiskmargin` commands return disk-based gain margins in the `GainMargin` field of their output structures. These margins take the form `[gmin,gmax]`, meaning that the open-loop gain can be multiplied by any factor in that range without loss of closed-loop stability. Beginning in R2020a, the lower end of the range `gmin` can be negative for some negative values of the eccentricity `E`, if the closed-loop system remains stable even if the sign of the open-loop gain changes. The eccentricity controls the bias in the disk-based gain margin toward gain decrease or increase (see Stability Analysis Using Disk Margins). Previously, the gain-margin range was always positive.

dksyn is not recommended

Warns

The `musyn` command, introduced in R2019b, performs µ-synthesis with better numeric stability than `dksyn` and better results for real uncertain parameters and for repeated parameters. `musyn` can also design fixed-structure controllers. Therefore, it is recommended that you use `musyn` instead of `dksyn`. Similarly, use `musynOptions` and `musynperf` instead of `dksynOptions` and `dksynperf`.

dmplot is not recommended

Still runs

`dmplot` is not recommended. Use `diskmarginplot` instead. `diskmarginplot` includes several visualizations, including:

- Disk-based gain and phase margins of a dynamic system as a function of frequency
- Ranges of simultaneous gain and phase variations corresponding to specific disk margins
- The disk of values taken by the uncertainty model used to represent gain and phase variations

wcsigma renamed to wcsigmaplot

Still runs

The function `wcsigma` has been renamed to `wcsigmaplot`. The syntaxes and functionality are unchanged, and `wcsigma` still runs.

R2019b

Version: 6.7

New Features

Bug Fixes

Compatibility Considerations

musyn Command: Design unstructured and fixed-structure robust controllers using mu synthesis, including for systems with real uncertainty

The new `musyn` command designs a robust controller for an uncertain plant using μ synthesis, which extends the methods of H_∞ synthesis. `musyn` can perform μ synthesis on plants with parameter uncertainty, dynamic uncertainty, or both. You can use `musyn` to:

- Synthesize "black box" unstructured robust controllers.
- Robustly tune a fixed-order or fixed-structure controller made up of tunable components such as PID controllers, state-space models, and static gains.

`musyn` uses a process called D-K iteration to find a controller that minimizes the robust H_∞ performance of the closed-loop system. The robust H_∞ performance, also called μ, quantifies how the performance of a feedback loop is affected by modeled uncertainty.

`musyn` replaces `dksyn`. In addition to supporting fixed-structure controllers, the new command has better numeric stability and yields better results for real uncertain parameters and for repeated parameters.

For information about performing μ synthesis and interpreting results, see:

- `musyn`
- Robust Controller Design Using Mu Synthesis

mixsyn Command: Perform mixed-sensitivity H-infinity synthesis in a simpler way for all plants, including discrete-time and complex-valued plants

Improvements to `mixsyn` allow direct mixed-sensitivity H_∞ synthesis of controllers for:

- Discrete-time plants
- Plants with complex-valued state-space matrices

Previously, `mixsyn` did not support plants with complex-valued matrices. Also, `mixsyn` previously converted discrete-time plants to continuous time before performing H_∞ synthesis. With direct computation instead of conversion, the new algorithm is more stable numerically.

Additionally, new syntaxes for `mixsyn` allow you to more easily specify target performance levels for mixed-sensitivity H_∞ synthesis. For more information on the new syntaxes, see `mixsyn`. For more information on mixed-sensitivity H_∞ synthesis generally, see Mixed-Sensitivity Loop Shaping.

Compatibility Considerations

The changes to `mixsyn` include new recommended syntaxes and changes to the information returned by the `info` output argument. For details, see:

- “mixsyn Name,Value options are not recommended” on page 5-3
- “mixsyn output argument info changed” on page 5-3

makeweight Command: Create frequency-weighting functions with more specific gain profiles

`makeweight` now lets you specify more characteristics of frequency-weighting functions that you create for robust controller synthesis. For instance, you can now:

- Specify a target gain at any frequency. Previously, you could specify only the unity-gain crossover frequency.
- Create a gain profile with a steeper transition between low gain and high gain by specifying the transfer-function order. Previously, `makeweight` created only first-order gain profiles.

Weighting functions are useful for capturing frequency-dependent requirements for controller design using functions such as `hinfsyn`, `mixsyn`, and `h2syn`. For more information about specifying weighting functions, see `makeweight`.

Functionality being removed or changed

mixsyn Name,Value options are not recommended

Still runs

Using `Name,Value` syntax to specify options for `mixsyn` is not recommended. Instead, to set a target performance range, use the `gamRange` input argument. For other options, create an options set with `hinfsynOptions`.

The following table shows how to update your calls to `mixsyn` to use the recommended ways of specifying options.

Not Recommended	Recommended
`[K,CL,GAM] = mixsyn(___,'GMIN',gmin,'GMAX',gmax)`	`gamRange = [gmin gmax];` `[K,CL,GAM] = mixsyn(___,gamRange)`
`[K,CL,GAM] = mixsyn(___,'TOLGAM',tol)`	`opts = hinfsynOptions('RelTol',tol);` `[K,CL,GAM] = mixsyn(___,opts);`
`[K,CL,GAM] = mixsyn(___,'METHOD',meth)`	`opts = hinfsynOptions('Method',meth);` `[K,CL,GAM] = mixsyn(___,opts);`
`[K,CL,GAM] = mixsyn(___,'DISPLAY','on')`	`opts = hinfsynOptions('Display','on');` `[K,CL,GAM] = mixsyn(___,opts);`

For more information about these and additional options available for `mixsyn` computations, see `hinfsynOptions`. For information on all syntaxes of `mixsyn`, see the `mixsyn` reference page.

mixsyn output argument info changed

Behavior change

The optional output argument `info` of the `mixsyn` command has changed. The new fields of `info` are the same as those of `hinfsyn`. For more information about the change, see `info` output argument changed on the `hinfsyn` reference page, which describes the same change for `hinfsyn` in R2018b.

R2019a

Version: 6.6

New Features

Bug Fixes

Compatibility Considerations

wcdiskmargin Command: Compute loop-at-a-time and multiloop worst-case stability margins

`wcdiskmargin` calculates the minimum guaranteed disk-based stability margins for an uncertain feedback loop. The disk margin of a feedback loop measures how much the loop gain (or phase) can vary without the system going unstable. For MIMO systems, you can compute the loop-at-a-time stability margins, which are the margins for each feedback channel with all other loops closed. You can also compute a multiloop margin, which measures the maximum tolerable independent and concurrent gain variation (or phase variation) over all feedback channels. The function also returns the worst-case perturbation, the uncertain-element values that yield the weakest margins. `wcdiskmargin` is the counterpart of `diskmargin` for uncertain feedback loops.

`wcdiskmargin` replaces `wcmargin`, which could compute only loop-at-a-time margins. Also, `wcdiskmargin` includes an optional eccentricity parameter, E, that lets you vary the shape of the uncertainty region used to compute the disk margin. Varying the eccentricity can improve the gain and phase margin estimates.

For more information, see `wcdiskmargin`.

Compatibility Considerations

`wcdiskmargin` replaces `wcmargin` for computing worst-case disk-based stability margins. `wcmargin` is not recommended. For more information, see “wcmargin is not recommended” on page 6-3.

gapmetric Improvements: Compute gap metrics with an algorithm that is stabler, numerically safer, and more reliable for discrete-time plants

Improvements to the `gapmetric` command let you more reliably compute the gap metric and the Vinnicombe (ν-gap) metric for evaluating the difference between two dynamic systems. The new algorithm is numerically safer than the previous algorithm. Also, `gapmetric` now works directly with discrete-time and descriptor systems, improving reliability for such systems. Previously, `gapmetric` converted discrete-time systems to continuous time before computing the gap metric.

For more information about computing these metrics, see `gapmetric`.

lncf and rncf commands: Compute normalized coprime factorizations

Use the new `lncf` and `rncf` commands to compute left and right normalized coprime factorizations of SISO or MIMO linear dynamic systems. These factorizations are used in other normalized coprime factor computations such as model reduction (`ncfmr`) and controller synthesis (`ncfsyn`).

For more information, see `lncf` and `rncf`.

h2syn Improvements: Compute H2 controller in a more numerically reliable and gain-scheduling friendly way

Improvements to the `h2syn` algorithm yield more numerically reliable results for all plants, with improved regularization and scaling of the plant.

In addition, `h2syn` now optionally returns gain matrices you can use to express the synthesized H_2 controller in observer state-space form. This form is useful for designing gain-scheduled H_2 controllers. For more information, see `h2syn`.

The new `h2synOptions` options command lets you turn off automatic scaling and regularization for H_2 synthesis with `h2syn`.

diskmargin command: Obtain lower and upper bounds on disk margin

The algorithm used by `diskmargin` involves a μ structured singular-value computation that produces lower and upper estimates on the true disk margin. The output structures of the `diskmargin` command now include fields containing these bounds on the true disk margin. The value in the `LowerBound` field is the same as the guaranteed disk margin returned in the `DiskMargin` field. The value in the `UpperBound` field represents an upper limit on the actual disk margin of the system. In other words, the disk margin is guaranteed to be no worse than `LowerBound` and no better than `UpperBound`.

For more information about computing disk margins, see `diskmargin`.

Functionality being removed or changed

wcmargin is not recommended

Still runs

`wcmargin` is not recommended. Use the new `wcdiskmargin` command instead. `wcdiskmargin` can compute both loop-at-a-time and multiloop margins, while `wcmargin` could only compute loop-at-a-time margins. `wcdiskmargin` can also compute stability margins with respect to independent, concurrent variations at both the plant inputs and plant outputs. Further, `wcdiskmargin` includes an optional eccentricity parameter, *E*, that lets you vary the shape of the uncertainty region used to compute the disk margin. Varying the eccentricity can improve the gain and phase margin estimates.

The following table shows some typical uses of `wcmargin` and how to update your code to use `wcdiskmargin` instead.

Not Recommended	Recommended
`wcmarg = wcmargin(L)`	`[wcDM,wcu] = wcdiskmargin(L,'siso')`
`[wcmargI,wcmargO] = wcmargin(P,C)`	`[wcmargI,wcuI] = wcdiskmargin(C*P,'siso')`, for margins at plant input `[wcmargO,wcuO] = wcdiskmargin(P*C,'siso')`, for margins at plant output

There are no plans to remove `wcmargin` at this time.

wcsens and wcgainOptions are not recommended

Still runs

Use of `wcsens` is not recommended. Instead, form the transfer function you want to analyze, and use `wcgain` to obtain the worst-case sensitivity. This approach has improved numeric stability and more reliable results relative to `wcsens`. For more information and an example, see Worst-Case Sensitivity Functions of Feedback Loops.

`wcgainOptions`, which generates option sets for `wcsens`, is not recommended. Instead, use `wcOptions` to create option sets for `wcgain` and other worst-case computation functions.

There are no plans to remove `wcsens` or `wcgainOptions` at this time.

R2018b

Version: 6.5

New Features

Bug Fixes

Compatibility Considerations

hinfsyn Improvements: Compute H-infinity controller in a simpler, numerically safer, and gain-scheduling friendly way, including controllers for discrete-time and complex-valued plants

Improvements to the `hinfsyn` algorithm yield more numerically reliable results for all plants. The improvements to `hinfsyn` also allow direct computation of H_∞ controllers for:

- Discrete-time plants
- Plants with complex-valued state-space matrices

Previously, `hinfsyn` did not support plants with complex-valued matrices. Also, `hinfsyn` previously converted discrete-time plants to continuous time before performing H_∞ synthesis. With direct computation instead of conversion, the new algorithm is more stable numerically.

In addition, `hinfsyn` now optionally returns gain matrices you can use to express the synthesized H_∞ controller in observer state-space form. This form is useful for designing gain-scheduled H_∞ controllers. For more information, see `hinfsyn`.

Two new commands let you compute H_∞ controllers for full-information and full-control problems:

- `hinffi` — Full information, which assumes controller has access to state values and disturbance signals
- `hinffc` — Full control, which assumes controller can directly affect state values and error signals

Compatibility Considerations

The changes to `hinfsyn` include new recommended syntaxes and changes to the information returned by the `info` output argument. For details, see:

- “hinfsyn Name,Value options are not recommended” on page 7-3
- “hinfsyn output argument info changed” on page 7-4

diskmargin Command: Compute disk-based stability margins for SISO or MIMO feedback loops, and vary disk shape for better margin estimates

The new `diskmargin` command computes disk-based stability margins for SISO or MIMO feedback loops. The disk margin of a feedback loop measures how much the loop gain (or phase) can vary without the system going unstable. For MIMO feedback loops, you can compute the disk margin for each feedback channel independently (with all other loops closed), or compute a multiloop disk margin. The multiloop margin measures the maximum tolerable independent and concurrent gain and phase variation over all feedback channels.

`diskmargin` includes an optional eccentricity parameter, E, that lets you vary the shape of the uncertainty region used to compute the disk margin. Varying the eccentricity can improve the gain and phase margin estimates. Computing margins based on the sensitivity, complementary sensitivity, or balanced sensitivity of the loop correspond to E = 1, -1, or 0, respectively.

For more information, see `diskmargin` and Stability Analysis Using Disk Margins.

Compatibility Considerations

The new `diskmargin` command replaces `loopmargin` for computing disk-based stability margins. `loopmargin` is not recommended. For more information, see "loopmargin is not recommended" on page 7-3.

Functionality being removed or changed

loopmargin is not recommended

Still runs

The `loopmargin` command is not recommended. To compute disk-based stability margins of SISO and MIMO systems, use `diskmargin` instead. For loop-at-a-time classical gain margins, use `allmargin`.

For stability margin analysis of feedback loops modeled in Simulink®, first linearize the model, and then use `diskmargin`. For an example, see Stability Margins of a Simulink Model.

The new `diskmargin` command has improved numeric stability and more reliable results relative to `loopmargin`. The new command also includes an option for varying the eccentricity of the disk for better margin estimates. For more information, see `diskmargin`.

Update Code

Not Recommended	Recommended
`[CM,DM,MM] = loopmargin(L)`	`[DM,MM] = diskmargin(L)` returns the disk margins of each feedback channel with all other loops closed in the structure `DM`, and the multiloop disk margin in the structure `MM`. For more information, see `diskmargin`. `CM = allmargin(L)` returns the classical loop-at-a-time gain and phase margins returned by `loopmargin` as `CM`. For more information, see `allmargin`.
`[CMI,DMI,MMI,CMO,DMO,MMO,MMIO] = loopmargin(P,C)`	`MMIO = diskmargin(P,C)` returns the multiloop disk margins returned by `loopmargin`. For more information, see `diskmargin`. `CM = allmargin(P*C)` and `CM = allmargin(C*P)` return the classical gain and phase margins at the plant output and plant input, respectively. For more information, see `allmargin`.
`[cm,dm,mm] = loopmargin(Model,Blocks,Ports)`	First linearize the Simulink model, and then use `diskmargin` or `allmargin`. For an example, see Stability Margins of a Simulink Model.

hinfsyn Name,Value options are not recommended

Still runs

Using `Name,Value` syntax to specify options for `hinfsyn` is not recommended. Instead, to set a target performance range, use the `gamRange` input argument. For other options, create an options set with `hinfsynOptions`.

Update Code

The following table shows how to update your calls to `hinfsyn` to use the recommended ways of specifying options.

Not Recommended	Recommended
`[K,CL,GAM] = hinfsyn(___,'GMIN',gmin,'GMAX',gmax)`	`[K,CL,GAM] = hinfsyn(___,gamRange)`
`[K,CL,GAM] = hinfsyn(___,'TOLGAM',tol)`	`opts = hinfsynOptions('RelTol',tol);` `[K,CL,GAM] = hinfsyn(___,opts);`
`[K,CL,GAM] = hinfsyn(___,'METHOD',meth)`	`opts = hinfsynOptions('Method',meth);` `[K,CL,GAM] = hinfsyn(___,opts);`
`[K,CL,GAM] = hinfsyn(___,'DISPLAY','on')`	`opts = hinfsynOptions('Display','on');` `[K,CL,GAM] = hinfsyn(___,opts);`

For more information about these and additional options available for `hinfsyn` computations, see `hinfsynOptions`.

hinfsyn output argument info changed

Behavior change

The optional output argument `info` of the `hinfsyn` command has changed. The new fields of `info` are described on the `hinfsyn` reference page. Formerly, `info` was a structure with the following fields.

AS	All solutions controller, scaled so that $\|Q\|_\infty < 1$
KFI	Full information gain matrix (constant feedback) $u_2(t) = K_{FI}\begin{bmatrix} x(t) \\ u_1(t) \end{bmatrix}.$
KFC	Full control gain matrix (constant output-injection; K_{FC} is the dual of K_{FI})
GAMFI	H_∞ cost for full information K_{FI}
GAMFC	H_∞ cost for full control K_{FC}

`info.AS` is still available, but its scaling has changed. See “Scaling of info.AS” on page 7-5.

For the remaining fields, the following functions are recommended instead:

- `info.KFI`, `info.GAMFI` — Use `hinffi` for full-information synthesis.
- `info.KFC`, `info.GAMFC` — Use `hinffc` for full-control synthesis.

These fields are hidden in the `info` argument returned by `hinfsyn`. However, you can still access them using dot notation. For instance:

```
[K,CL,gamma,info] = hinfsyn(P,nmeas,ncont);
gfi = info.GAMFI;
gfc = info.GAMFC;
```

Scaling of info.AS

Prior to R2018b, the all-solutions controller parameterization `info.AS` was scaled so that the free stable contraction map Q was constrained by $\|Q\|_\infty < 1$. In R2018b, the scaling of `info.AS` has changed, so that the constraint on Q is $\|Q\|_\infty < \gamma$, where γ is `info.gamma`. This new constraint ensures that the all-solutions controller K_{AS} has a finite limit as the target performance level goes to infinity.

R2018a

Version: 6.4.1

Bug Fixes

R2017b

h2syn Improvements: Handle singular problems using automatic regularization, and obtain better results when computing discrete-time controllers

`h2syn` now achieves better results when synthesizing discrete-time controllers. Additionally, for improved numeric stability and tractability, `h2syn` now automatically regularizes singular problems where the standard implicit assumptions of H_2 synthesis are not satisfied. For more information, see `h2syn`.

Previously, `h2syn` errored or returned unpredictable results for singular problems.

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.

R2017a

Version: 6.3

New Features

Bug Fixes

Fixed-structure tuning of discrete-time control systems with hinfstruct

You can now use `hinfstruct` to tune discrete-time fixed-structure control systems. To tune a discrete-time control system, use the same procedure and command syntax that you use to tune a continuous-time control system.

dksynperf command for obtaining robust H∞ performance

The robust H_∞ norm of an uncertain closed-loop system is the smallest value γ such that the I/O gain of the system stays below γ for all modeled uncertainty up to size $1/\gamma$ (in normalized units). `dksyn` synthesizes a robust controller by minimizing this quantity over all possible choices of controller. The new `dksynperf` command computes this quantity for a specified uncertain model. This quantity is useful, for instance, for simplifying a synthesized controller without sacrificing robust H_∞ performance. For an example, see `dksynperf`.

Name property of umat object

The uncertain matrix object, `umat`, now has a `Name` property. Use the property to assign a name to the uncertain matrix. When you convert a static control design block such as `ureal` to an uncertain matrix using `umat(blk)`, the `Name` property of the block is preserved.

R2016b

Improved Robustness Analysis Workflow: Calculate robustness margins using the new robstab and robgain functions

New functions improve the workflow for computing robust stability margins and robust performance margins of uncertain systems. The new functions compute the worst-over-frequency margins, and can also return the margins as a function of frequency.

- `robstab` — Calculate the robust stability margin. This margin is a measure of how far the uncertain elements of a system can deviate from their nominal values before the system becomes unstable.
- `robgain` — Calculate the robust performance margin. This margin is a measure of how far the uncertain elements of a system can deviate from their nominal values before the peak gain of the system exceeds some specified value.

For uncertain state-space (`uss` and `genss`) models, both functions use a new algorithm that always finds the smallest margin across all frequencies, as described in “Improved Robustness Guarantees: Compute the structured singular value µ without frequency gridding” on page 11-2. The new functions replace and augment the functionality previously provided by `robuststab` and `robustperf`. For more information about using the new functions, see the reference pages for `robstab` and `robgain`.

Compatibility Considerations

Using `robuststab` and `robustperf` is not recommended. Instead:

- Replace instances of `robuststab` or `robustperf` with `robstab` and `robgain`, respectively.
- Replace instances of `robuststabOptions` or `robustperfOptions` with `robOptions`.

Improved Robustness Guarantees: Compute the structured singular value µ without frequency gridding

The new `robstab` and `robgain` functions base their analysis on the structured singular value, μ. For uncertain state-space (`uss` or `genss`) models, these functions use a new algorithm that is guaranteed to detect critical peaks of μ, and always produces correct guarantees of robustness.

The new algorithm adaptively selects frequencies for computing μ. The returned upper bounds on μ are guaranteed to hold over each interval between frequencies. In previous releases, μ analysis used a grid-based computation that could miss important peaks in μ and produce over-optimistic guarantees of robustness. For `ufrd` and `genfrd` models, the computation is still performed pointwise at the frequencies specified in the model.

Improved Worst-Case Gain Computations

The `wcgain` function computes bounds on the worst-case gain of an uncertain system, uses. This command now uses the new structured-singular-value algorithm described in “Improved Robustness Guarantees: Compute the structured singular value µ without frequency gridding” on page 11-2 . Therefore this function is guaranteed to return accurate bounds on the worst-case gains for `uss` or `genss` models. Previously, `wcgain` used a grid-based approach that could miss important peaks and produce over-optimistic worst-case gains.

R2016b also includes a new function for visualizing worst-case gain as a function of frequency, `wcsigma`. Like `wcgain`, this function is guaranteed to produce correct worst-case gains for `uss` or `genss` models. `wcsigma` replaces and improves the functionality previously provided by `wcgainplot`.

Specify options for `wcgain` and `wcsigma` using the new options command `wcOptions`. You can also use `wcOptions` for `wcmargin`, `wcsens`, and `wcnorm`.

Compatibility Considerations

Using `wcgainplot`, `wcgainOptions`, and `wcmarginOptions` is not recommended. Instead:

- Replace instances of `wcgainplot` with `wcsigma`.
- Replace instances of `wcgainOptions` or `wcmarginOptions` with `wcOptions`.

Additionally, there are some changes to the default behavior and supported options for `wcgain`:

- The `MaxOverFrequency` option of `wcgainOptions` is now the `VaryFrequency` option of `wcOptions`. To compute the worst-case gain as a function of frequency, use `wcOptions('VaryFrequency','on')`.
- In the `info` output of `wcgain`, when `VaryFrequency` is `'off'`, the field `Info.Frequency` now contains the frequency at which the worst-case peak gain occurs. Previously, `Info.Frequency` contained a vector of all frequencies used for analysis, even when `MaxOverFrequency` was `'on'`.
- In `wcOptions`, the option `'VaryFrequency' = 'on'` is not available for arrays of uncertain models.
- By default, the `Sensitivity` option of `wcOptions` is `'off'`. Previously, the default value of the `Sensitivity` option of `wcgainOptions` was `'on'`. Therefore, to compute the sensitivity of the worst-case gain to each uncertain element, use `wcOptions('Sensitivity','on')`.
- The `MaxOverArray` option of `wcgainOptions` no longer exists in `wcOptions`. Instead, when you provide an array of uncertain models, `wcgain` always returns the worst case gain over the entire array. To compute the worst-case gain individually for each model in an array, use a `for` loop to step through each array entry. For example, suppose that `uarray` is an array of *N* uncertain models. The following code computes the worst-case gain for each entry in `uarray`.

  ```
  for k = 1:N
     [wcg(k),wcu(k)] = wcgain(uarray(:,:,k));
  end
  ```

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
• `robuststab` and `robuststabOptions` • `robustperf` and `robustperfOptions`	Still runs	`robstab`, `robgain`, and `robOptions`	Replace instances of `robuststab` or `robustperf` with `robstab` and `robgain`, respectively. See “Improved Robustness Analysis Workflow: Calculate robustness margins using the new robstab and robgain functions” on page 11-2.
`wcgainplot`	Still runs	`wcsigma`	Replace instances of `wcgainplot` with `wcsigma`. See “Improved Worst-Case Gain Computations” on page 11-2.
• `wcgainOptions` and `wcmarginOptions`	Still runs	`wcOptions`	Replace instances of `wcgainOptions` or `wcmarginOptions` with `wcOptions`. For more details on the differences between the old options commands and `wcOptions`, see “Improved Worst-Case Gain Computations” on page 11-2.

R2016a

Version: 6.1

New Features

Bug Fixes

Compatibility Considerations

Control system tuning tools moved to Control System Toolbox

A Robust Control Toolbox license is no longer required to use the `systune` or `looptune` commands or to use Control System Tuner. You can now:

- Tune control systems modeled in MATLAB® (tunable `genss` models) with a Control System Toolbox™ license.
- Tune control systems modeled in Simulink with a Simulink Control Design™ license.

The following still requires a Robust Control Toolbox license:

- `hinfstruct` command
- Robust tuning of control systems with parameter uncertainty using `systune`, `looptune`, or Control System Tuner

Functionality being removed or changed

Functionality	Result	Use This Instead	Compatibility Considerations
`a`, `b`, `c`, `d`, and `e` properties of `uss` 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.
`cpmargin`	Still works	`ncfmargin`	If your code uses `cpmargin`, modify it to use `ncfmargin` instead.

R2015b

Version: 6.0

New Features

Bug Fixes

Compatibility Considerations

Robust Tuning with systune Command or Control System Tuner App: Automatically tune controllers to maximize performance over a range of parameter values

Control System Tuner and the `systune` command now tune control systems for robustness against real parameter uncertainty in the plant. You represent parameter uncertainty in your control system model using uncertain real parameters `ureal` or `uss`. The software automatically finds the worst combinations of parameter values and tunes the controller to maximize performance over the parameter uncertainty range.

In MATLAB, build a generalized state-space (`genss`) model of your control system using `ureal` or `uss` blocks to represent real parameter uncertainty in the plant. You can tune the model with `systune` or in Control System Tuner exactly as you would for a tunable control system model without uncertainty. For a detailed example, see Robust Tuning of Positioning System.

In Simulink, use linearization with block substitution to replace one more blocks in the model with uncertain values represented by `ureal` or `uss` objects. (Requires Simulink Control Design software.) See Robust Tuning of Mass-Spring-Damper System.

In both cases, when you tune the model, the software automatically adjusts the tunable components to achieve the specified performance as well as possible throughout the uncertainty range. Analysis plots automatically display random samples of the uncertain system to give you a visual sense of the performance variation.

For more information about robust tuning generally, see Robust Tuning Approaches.

Compatibility Considerations

Previously, when you used `systune` to tune a model that had uncertainties, the software would set the uncertain blocks to their nominal values before tuning the system. Now, `systune` tunes the model for robustness against those uncertainties. To recover the old behavior, i.e., to tune a controller for the nominal system only, use `getNominal` to obtain the nominal value. For example:

```
[CL,fSoft,GHard,info] = systune(getNominal(CL0),SoftReqs,HardReqs);
```

In this example, `CL0` is a `genss` model containing uncertain blocks.

Gain Scheduling with systune and slTuner: Automatically tune the Lookup Table and Interpolation blocks used to model gain-scheduled controllers in Simulink

You can now use the `slTuner` interface to automatically tune control systems modeled in Simulink in which plant dynamics change with operating conditions or time. (Requires Simulink Control Design software.)

In such gain-scheduled control systems, the controller gains vary as a function of one or more scheduling variables. In the Simulink model, use the Lookup Table or Interpolation blocks to implement the variable controller gains. You then use the new `tunableSurface` command to parameterize the dependency of these gains on the scheduling variables. The software automatically tunes the coefficients of that parameterization so that the control system meets the tuning requirements you specify over the entire grid of scheduling-variable values. The software also writes the tuned coefficients back to the Lookup Table or Interpolation blocks.

In previous releases, you could not parameterize Lookup Table or Interpolation blocks in terms of the functional form of its dependence on the scheduling variable. As a result, you could not automatically tune a gain-scheduled control element and write the tuned coefficients back to the Simulink model. Using `systune` to tune and implement gain-scheduled controllers required a complex process of manually extracting coefficient values and inserting them in the blocks.

For more details, see Set Up Simulink Models for Gain Scheduling.

For examples showing how to use `tunableSurface` to tune gain-scheduled controllers implemented with Lookup Table blocks, see:

- Gain-Scheduled Control of a Chemical Reactor
- Tuning of Gain-Scheduled Three-Loop Autopilot

tunableSurface Object: Parameterize and tune gain-scheduled controllers using improved workflow

The new `tunableSurface` object lets you express gain in terms of tunable parameters for tuning gain-scheduled controllers with `systune`. In such gain-scheduled control systems, the controller gains vary as a function of one or more scheduling variables. You parameterize the dependency of controller gains on the scheduling variables. The software automatically tunes the coefficients of that parameterization so that the control system meets the tuning requirements you specify over the entire grid of scheduling-variable values. `tunableSurface` replaces the `gainsurf` command.

In previous releases, you could use the `gainsurf` command to represent tunable surfaces for control system tuning. With that command, you had to explicitly supply the values of the gain surface calculated over the grid of design points. `tunableSurface` simplifies that workflow by allowing you to specify the gain surface in terms of functions of the scheduling variables, such as the basis functions of a polynomial expansion.

For more details about creating tunable gain surfaces, see:

- Parametric Gain Surfaces
- `tunableSurface` reference page

Compatibility Considerations

`tunableSurface` replaces `gainsurf`, which was used in previous releases to parameterize controller gains as functions of scheduling variables. `gainsurf` still works, but might be removed in a future release. If you have scripts or functions that use `gainsurf`, consider updating them to use `tunableSurface` instead.

getNominal command for extracting nominal value of uncertain model

Use `getNominal` to replace the uncertain elements of a generalized model with their nominal values. All other control design blocks in the generalized model are unchanged. For example, suppose that `M` is a generalized state-space (`genss`) model that has both uncertain blocks and tunable blocks. The command `getNominal(M)` returns a `genss` model having the same tunable blocks as `M`.

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

usample samples uncertain blocks and preserves other control design blocks

The `usample` command now preserves any non-uncertain control design blocks when you use it to sample the uncertain elements of a generalized model. For example, suppose that `M` is a generalized state-space (`genss`) model that has both uncertain blocks and tunable blocks. The command `usample(M,N)` samples the uncertain blocks, and returns an array of `genss` models having the same tunable blocks as `M`.

Compatibility Considerations

Previously, when applied to models having tunable control design blocks, `usample` used the current (nominal) value of those blocks, and returned an array of numeric models. To recover the previous behavior, use `getValue`. For example, the following command randomly samples the uncertain blocks of `M`, replaces the tunable blocks of `M` with their current values, and returns an array of numeric state-space models.

```
Msamp = getValue(usample(M,N));
```

New property for limiting maximum frequency in random samples of ultidyn

Use the `SampleMaxFrequency` property of `ultidyn` to limit the natural frequency of dynamics when you take random samples of `ultidyn` blocks. For example, the following command creates SISO uncertain dynamics.

```
dH = ultidyn('dH',[1 1],'SampleMaxFrequency',1);
```

When you take random samples of `dH`, such as with `usample`, the dynamics of the samples are no faster than 1 rad/s. The default value of `SampleMaxFrequency` is `Inf` (no limit).

Also, the `SampleStateDim` property of `ultidyn` is changed to `SampleStateDimension`.

Compatibility Considerations

The property name `SampleStateDim` still works, but might be removed in a later release. If you have scripts or functions that use `SampleStateDim`, consider updating them to use `SampleStateDimension` instead.

Functionality being removed or changed

Functionality	Result	Use This Instead	Compatibility Considerations
`systune(CL0,...)` where `CL0` contains uncertain blocks	Tunes robustly against real parameter uncertainty in `CL0`	`systune(getNominal(CLO),...)`	Previously, `systune` used the nominal value of all uncertain blocks in the tuned model. Now, use `getNominal` explicitly to tune for the nominal system only. See “Robust Tuning with systune Command or Control System Tuner App: Automatically tune controllers to maximize performance over a range of parameter values” on page 13-2.
`gainsurf`	Still works	`tunableSurface`	If you have scripts or functions that use `gainsurf`, consider updating them to use `tunableSurface` instead. See “tunableSurface Object: Parameterize and tune gain-scheduled controllers using improved workflow” on page 13-3
`usample(M,N)`	Samples uncertain control design blocks of `M`, and preserves other control design blocks	`getValue(usample(M,N))`	Previously, `usample` used the current value of non-uncertain control design blocks. See “usample samples uncertain blocks and preserves other control design blocks” on page 13-4
`SampleStateDim` property of `ultidyn`	Still works	`SampleStateDimension`	Consider replacing `SampleStateDim` with `SampleStateDimension`.

R2015a

Robust tuning of controller parameters against a set of plant models specified through parameter variations in Control System Tuner app

When you use Control System Tuner to tune a Simulink model of a control system, you can now generate multiple plant models by varying model parameters. You can then tune the control system to satisfy your specified tuning goals for all the resulting models.

Tuning to multiple models is useful to help ensure that the tuned control system is robust against parameter variations or changes in operating conditions. For example, if a parameter in your Simulink model represents a process temperature, you can generate multiple models spanning the range of expected temperature variations, and tune your control system to meet your design requirements for all those models at once.

For more information about tuning control systems for multiple models in Control System Tuner, see Robust Tuning Using Multiple Plant Models in Control System Tuner. For an example showing how to specify parameter variations for tuning with Control System Tuner, see Tuning Control System with Multiple Valued Plant Parameters using Control System Tuner.

Open Control System Tuner app with saved session from command line

Use the new syntax `controlSystemTuner(sessionfile)` to open Control System Tuner and load data from a saved session. When you use Control System Tuner, you can click 💾 **Save Session** to save session data to disk such as tuning goals you have created, response I/Os you have defined, operating points, and stored designs. The string `sessionfile` is the name of a session data file saved in the current working directory or on the MATLAB path. The software also opens the Simulink model associated with the saved session.

15

R2014b

Quick Loop Tuning option in Control System Tuner app for tuning control systems to target loop bandwidth and stability margins

Quick Loop Tuning lets you use a loop-shaping approach to tune SISO or MIMO feedback loops in Control System Tuner. You can use Quick Loop Tuning to tune control systems modeled in MATLAB or Simulink. With Quick Loop Tuning you can tune your system to meet target gain crossover and margin requirements without explicitly creating tuning goals that capture these requirements. You specify feedback loops to tune by selecting the control signals and measurement signals in a block diagram of your control system. Control System Tuner adjusts the tunable parameters of your system such that the open-loop gain crossover falls within the desired frequency range with the gain and phase margins you specify.

For more information about using Quick Loop Tuning, see Quick Loop Tuning of Feedback Loops in Control System Tuner.

Tuning goals for automated tuning to meet transient response and disturbance rejection requirements

New tuning goals let you explicitly specify a target transient response or a minimum disturbance rejection in a tuned control system. These tuning goals are available both in Control System Tuner and at the command line when tuning with `systune`.

The transient response goal lets you shape how the closed-loop system responds to a specific input signal. You specify the desired transient response as a reference model. The target transient response is the response of the reference model to an impulse, step, ramp, or custom input signal. To use the transient response goal:

- In Control System Tuner, in the **Tuning** tab, in the **New Goal** menu, select `Transient Response Matching`.
- At the command line, specify the design requirement using `TuningGoal.Transient`.

The step rejection goal lets you specify a minimum standard for rejecting disturbances. You specify characteristics such as the maximum amplitude and settling time of the response at some point in your control system to a step disturbance injected at another point in the system. Alternatively, specify a reference system whose response to step input is the target response. To use the step rejection goal:

- In Control System Tuner, in the **Tuning** tab, in the **New Goal** menu, select `Rejection of Step Disturbances`.
- At the command line, specify the design requirement using `TuningGoal.StepRejection`.

MATLAB code generation from Control System Tuner app for automatically scripting control system tuning tasks

You can now generate a MATLAB script for control system tuning from Control System Tuner. Generated MATLAB scripts are useful when you want to programmatically reproduce a result you obtained interactively. You can also use generated code to perform multiple tuning operations with systematic variations in tuning configurations such as model operating point or tuning goals.

For more information, see Generate MATLAB Code from Control System Tuner for Command-Line Tuning.

Enhanced constraints on controller dynamics for control system tuning

New functionality gives you more flexibility when specifying constraints on controller dynamics for control system tuning. The following new features are available in both Control System Tuner using **Controller Poles Goal** and when tuning at the command line using `TuningGoal.ControllerPoles` (formerly `TuningGoal.StableController`).

- You can now specify a minimum damping constant for the poles of a tunable block. Previously, the damping constant of controller poles could take any value between zero and 1.
- You can now specify a negative value for the minimum decay rate of controller poles, allowing for unstable controllers. Previously, the minimum decay rate had to be positive, and therefore always enforced the stability of the constrained block.
- Fixed integrators in the constrained tunable block are no longer considered when evaluating the constraint. In other words, the tuning goal now constrains locations of all poles in the block except fixed integrators, such as the I term in a PID controller.

For more information about these features, see:

- Controller Poles Goal, for tuning in Control System Tuner.
- The `TuningGoal.ControllerPoles` reference page, for tuning at the command line.

Compatibility Considerations

`TuningGoal.StableController` has been renamed to `TuningGoal.ControllerPoles`. Scripts and functions that use `TuningGoal.StableController` do not generate errors. However, `TuningGoal.StableController` will not be maintained in future releases. You should replace instances of `TuningGoal.StableController` in your code with `TuningGoal.ControllerPoles`.

New syntax in TuningGoal.Poles for directly specifying constraints on dynamics

When you use `TuningGoal.Poles` to constrain the dynamics of a tuned control system, you can now directly specify the minimum decay rate, minimum damping, and maximum natural frequency when you create the tuning goal. To do so, use the following syntaxes:

```
R = TuningGoal.Poles(MinDecay,MinDamping,MaxFreq);
R = TuningGoal.Poles(Location,MinDecay,MinDamping,MaxFreq);
```

Previously, to specify such constraints on controller dynamics, you had to first create the tuning goal, and then modify its `MinDecay`, `MinDamping`, and `MaxFrequency` properties.

For more information, enter see the `TuningGoal.Poles` reference page.

TuningGoal.StepResp renamed to TuningGoal.StepTracking

The tuning requirement `TuningGoal.StepResp` is now called `TuningGoal.StepTracking`.

Compatibility Considerations

Scripts and functions that use `TuningGoal.StepResp` do not generate errors. However, `TuningGoal.StepResp` will not be maintained in future releases. You should replace instances of `TuningGoal.StepResp` in your code with `TuningGoal.StepTracking`.

DisturbanceInput property of TuningGoal.Rejection renamed to Location

The `DisturbanceInput` property of the tuning requirement `TuningGoal.Rejection` is now called `Location`, to unify the names of similar properties of several tuning requirements. If `Req` is a `TuningGoal.Rejection` requirement, you can access this property using `Req.Location`.

Compatibility Considerations

Scripts and functions that use the `DisturbanceInput` property do not generate errors. However, the `DisturbanceInput` property will not be maintained in future releases. You should replace instances of `DisturbanceInput` in your code with `Location`.

Functionality being removed or changed

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
`TuningGoal.StableController`	Still works	`TuningGoal.ControllerPoles`	Consider replacing `TuningGoal.StableController` with `TuningGoal.ControllerPoles`.
`TuningGoal.StepResp`	Still works	`TuningGoal.StepTracking`	Consider replacing `TuningGoal.StepResp` with `TuningGoal.StepTracking`.
`DisturbanceInput` property of `TuningGoal.Rejection`	Still works	`Location` property	Consider replacing `DisturbanceInput` with `Location`.

R2014a

Control System Tuner app for automated tuning of control systems

The new Control System Tuner lets you interactively tune SISO or MIMO control systems modeled in MATLAB or Simulink. Control System Tuner tunes the control system parameters to meet design requirements you specify, such as reference tracking, disturbance rejection, stability margins, loops shapes, and sensitivity. You can examine multiple system responses in both the time and frequency domains to evaluate performance of the tuned control system.

If you have Simulink Control Design software, you can tune a control system represented by a Simulink model. Control System Tuner can tune most blocks used to create a control system in Simulink. These blocks include `Gain`, `PID Controller`, `Transfer Fcn`, `State-Space`, `Zero-Pole`, `Discrete Filter`, and the `LTI System` block. Any controller architecture created using these blocks can be tuned. To access Control System Tuner for tuning a Simulink model, select **Analysis** > **Control Design** > **Control System Tuner**.

Control System Tuner can also tune a control system represented by a tunable `genss` model. Any control architecture constructed with Control Design Blocks such as `ltiblock.pid`, `ltiblock.tf`, or `realp` blocks can be tuned. To open Control System Tuner for tuning a control system modeled in MATLAB, use the `controlSystemTuner` command.

For more information about using Control System Tuner, see:

- Automated Tuning Basics
- Tuning with Control System Tuner

Step response and LQG requirements for control system tuning with systune and looptune commands

New `TuningGoal` requirement objects allow you to specify tuning objectives for automated tuning of control systems with `systune` and `looptune`.

- `TuningGoal.StepResp` — Requires that the step response between specified locations in the control system match the step response of a specified reference system. For details about this requirement, see the `TuningGoal.StepResp` reference page.
- `TuningGoal.LQG` — Specifies a linear-quadratic-gaussian (LQG) goal for control system tuning. This requirement lets you quantify control performance as an LQG cost. For details about this requirement, see the `TuningGoal.LQG` reference page.

Improvements to TuningGoal requirements for control system tuning

This release introduces a variety of improvements to `TuningGoal` requirement objects for automated tuning of fixed-structure control systems with `systune` and `looptune`.

Tuning Goals for constraining dynamics impose implicit stability constraints

`TuningGoal.StableController` and `TuningGoal.Poles` now impose implicit stability constraints on controller or system dynamics. This allows you to require poles of the controller or the closed-loop control system to be stable, without necessarily limiting the minimum decay or maximum frequency of those poles. Previously, you had to specify finite values for minimum decay and maximum frequency when using these tuning goals.

Compatibility Considerations

The default values of the `MinDecay` and `MaxFrequency` properties of these requirements have changed. If you have scripts that use `TuningGoal.StableController` or `TuningGoal.Poles` requirements with default values, update those scripts to explicitly set the finite values you want.

Property	Previous Default Value	New Default Value
`TuningGoal.Poles.MinDecay` `TuningGoal.StableController.MinDecay`	`1e-6`	`0`
`TuningGoal.Poles.MaxFrequency` `TuningGoal.StableController.MaxFrequency`	`1e6`	`Inf`
`TuningGoal.Poles.MinDamping`	`1e-6`	`0`

Option to limit dynamics constraint to poles in a particular feedback loop

A new syntax for creating the `TuningGoal.Poles` requirement allows you to constrain only the poles of the sensitivity function measured at a specified location. Use this syntax to narrow the scope of the requirement to a particular feedback loop.

For example, suppose you have a cascaded-loop control system in which the inner and outer loops contain loop-opening locations `'InnerLoop'` and `'OuterLoop'`, respectively. The following command uses the new syntax to constrain the poles of the inner loop sensitivity function:

```
Req = TuningGoal.Poles('InnerLoop');
Req.MinDamping = 0.5;
Req.Openings = 'OuterLoop';
```

`Req` imposes a minimum damping on the poles of the inner loop sensitivity function measured with the outer loop open. The dynamics of blocks that do not participate to the inner loop are ignored.

For more information about using this constraint, see the `TuningGoal.Poles` reference page.

TuningGoal.Tracking allows specification of peak error

A new syntax for creating the `TuningGoal.Tracking` requirement allows you to specify a maximum tracking error for a particular input-output pair in terms of a response time, a relative DC error, and a peak relative error across all frequencies. These parameters are converted to the following expression for the maximum tracking error:

$$\text{MaxError} = \frac{(\text{PeakError})s + \omega_c(\text{DCError})}{s + \omega_c}.$$

For more information about how to specify tracking error requirements, see the `TuningGoal.Tracking` reference page.

Specification of signal scaling in MIMO closed-loop Tuning Goals

New properties in several closed-loop Tuning Goals allow you to specify the relative amplitudes of multiple input and output signals in the loops constrained by the requirements. Use these properties to reduce cross-coupling in tuned systems when the choice of units results in a mix of small and large signals.

- `TuningGoal.Tracking` and `TuningGoal.Overshoot` now have an `InputScaling` property. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.
- `TuningGoal.Gain` and `TuningGoal.Variance` now have `InputScaling` and `OutputScaling` properties. The values you set for these properties are used to scale the closed-loop transfer function $T(s)$ on which you impose the tuning requirement. The requirement is evaluated for the scaled transfer function $D_o^{-1}T(s)D_i$. D_o and D_i are diagonal matrices formed from the `OutputScaling` and `InputScaling` property, respectively.

For more information on how to interpret and use these properties, see the reference pages for the Tuning Goals.

Option to remove stability constraint from loop-shape and gain-limiting Tuning Goals

The new `Stabilize` property of loop-shaping and gain-limiting Tuning Goals allows you turn off the implicit closed-loop stability constraint. If stability for the specified loop is not required or cannot be achieved, set `Stabilize` to `false` to relax the stability constraint.

This property is available for the following Tuning Goals:

- `TuningGoal.LoopShape`
- `TuningGoal.Gain`, `TuningGoal.WeightedGain`
- `TuningGoal.MinLoopGain`,`TuningGoal.MaxLoopGain`

For more information on how to use the `Stabilize` property, see the reference pages for the Tuning Goals.

ScalingOrder property added to TuningGoal.Margins

The `TuningGoal.Margins` tuning goal has a new property, `ScalingOrder`. This property controls the number of states in the diagonal scalings involved in computing MIMO stability margins. Increasing the order may improve results at the expense of increased computations.

Previously, this scaling order was set as a tuning option in `systuneOptions`.

Compatibility Considerations

If you have scripts that use the `ScalingOrder` option of `systuneOptions`, set the `ScalingOrder` property of `TuningGoal.Margins` instead.

Improved control system tuning of Simulink models with systune or looptune functions using slTuner interface (with Simulink Control Design)

Use the new `slTuner` interface for tuning control systems in Simulink models. This interface replaces `slTunable`. The `slTuner` interface allows you to:

- Tune model blocks and subsystems to meet tuning goals using the `systune` and `looptune` functions.
- Perform robust tuning of a controller against a set of plant models using `systune`. You can configure an `slTuner` interface to vary model parameter values and operating points. When you call `systune` for the interface, the software returns a control system that satisfies the tuning goals for all the specified model variations.
- Validate the controller design by examining the transfer function for relevant I/O sets using the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` functions.

`slTuner`, similar in design to `slLinearizer`, simplifies I/O management in the controller tuning and validation workflow. You specify signals of interest as analysis points. You can use these analysis points to configure design requirements and specify linearization inputs/outputs when you extract transfer functions.

For more information on command-line tuning of Simulink models with `slTuner`, see:

- Programmatic Control System Tuning

  Loop-Shaping Design

Compatibility Considerations

The `slTunable` interface will continue to work for backward compatibility. However, only the `slTuner` interface will be supported and enhanced in future releases. Therefore, adoption of the `slTuner` interface is strongly recommended.

For documentation of the `slTunable` interface, see `slTunable` in the R2013b documentation.

R2013b

Version: 5.0

New Features

Bug Fixes

Compatibility Considerations

Automatic tuning of gain-scheduled control systems with systune and looptune commands

You can now use `systune` and `looptune` to automatically tune control systems in which plant dynamics change with operating conditions or time. In such gain-scheduled control systems, the controller gains vary as a function of one or more scheduling variables. You parameterize the dependency of controller gains on the scheduling variables. The software automatically tunes the coefficients of that parameterization so that the control system meets the tuning requirements you specify over the entire range of plant operating conditions. The new `gainsurf` command helps you parameterize your controller gains as functions of scheduling variables.

Several new examples illustrating the workflow for gain-scheduled tuning, including:

- Tuning of Gain-Scheduled Three-Loop Autopilot
- Gain-Scheduled Control of a Chemical Reactor

For additional information about tuning gain-scheduled controllers, see Gain-Scheduled Controllers.

Automatic tuning of discrete-time control systems with systune and looptune commands

You can now use `systune` and `looptune` for automatic tuning of discrete-time control systems. This capability includes both:

- Control systems represented by discrete-time generalized LTI models (`genss` models with `Ts` property not equal to zero).
- Control systems represented by an `slTunable` interface to a Simulink mode. Set the `Ts` property of the `slTunable` interface to the sampling time at which you want to linearize the model.

To tune a discrete-time control system, use the same procedure and command syntax that you use to tune a continuous-time control system. For examples of discrete-time tuning, see:

- Digital Control of Power Stage Voltage
- MIMO Control of Diesel Engine

Sensitivity, overshoot, minimum and maximum loop gain requirements for control system tuning with looptune and systune

New `TuningGoal` requirement objects allow you to specify a variety of tuning objectives for automated tuning of fixed-structure control systems with `systune` and `looptune`. New tuning requirements include:

- `TuningGoal.Sensitivity` — Constraint on sensitivity to disturbance
- `TuningGoal.Overshoot` — Constraint on overshoot in step response
- `TuningGoal.MinLoopGain` — Minimum loop gain constraint
- `TuningGoal.MaxLoopGain` — Maximum loop gain constraint

Additionally, `TuningGoal.LoopShape` has two new syntaxes. These syntaxes allow you to specify a target crossover frequency or range of crossover frequencies for an open-loop response in your control system.

For more information about these `TuningGoal` requirement objects see the reference pages for each requirement object, and:

- Using Design Requirement Objects
- Specifying Design Requirements for systune
- Performance and Robustness Specifications for looptune

looptuneSetup command for switching from looptune to systune to use additional systune functionality

The new `looptuneSetup` command provides a bridge between the tuning commands `looptune` and `systune`. `looptuneSetup` takes the argument list for `looptune` and constructs an equivalent argument list for `systune`. The `looptuneSetup` command is valid for systems represented in either MATLAB or Simulink.

You can use this command to switch from `looptune` to `systune` to take advantage of the additional flexibility and functionality of `systune`. For example, `looptune` requires that you tune all channels of a MIMO feedback loop to the same target bandwidth. Converting to `systune` allows you to specify different crossover frequencies and loop shapes for each loop in your control system. Also, `looptune` treats all tuning requirements as soft requirements, optimizing them but not requiring that any constraint be exactly met. Converting to `systune` allows you to enforce some tuning requirements as hard constraints, while treating others as soft requirements.

You can also use `looptuneSetup` to probe into the tuning requirements that `looptune` implicitly imposes. When you use `looptune`, you specify a target loop bandwidth and stability margins. `looptune` expresses these as hard and soft tuning constraints, specified as `TuningGoal` objects. You can use `looptuneSetup` to examine these constraints. After examining the constraints, you can then alter them and pass them to `systune` for further tuning.

For more information, see the following reference pages:

- `looptuneSetup`
- `slTunable.looptuneSetup`

hinfnorm command for computing H∞ norm

The new `hinfnorm` command computes the H_∞ norm of SISO or MIMO systems. For SISO systems, the H_∞ norm is defined as the largest value of the frequency response magnitude. For MIMO systems, H_∞ norm is the largest singular value across frequencies.

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

Some properties of TuningGoal requirements renamed

The following properties of `TuningGoal` requirement objects are renamed to better reflect their purpose and uses:

Object	Previous Property Name	New Property Name
`TuningGoal.LoopShape`	`LoopTransfer`	`Location`

Object	Previous Property Name	New Property Name
`TuningGoal.Margins`	`LoopTransfer`	`Location`
`TuningGoal.Tracking`	`ReferenceInput`	`Input`
`TuningGoal.Tracking`	`TrackingOutput`	`Output`

Compatibility Considerations

If you have scripts or functions that use any of these properties, consider updating your code to use the new property names instead. Using the previous property names does not generate an error in this release, but the names might be removed in a future release.

Power iteration method option for structured singular value computation with mussv

A new `'p'` option to the `mussv` command allows you to specify a power iteration method for computing the lower bound on structured singular values (μ values). This method is recommended for cases of complex uncertainty. When at least one of the uncertain blocks specified in the block diagonal matrix structure is complex, `mussv` now uses the power iteration method by default.

For pure real uncertainty, `mussv` uses a gain-based lower bound algorithm by default.

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

Compatibility Considerations

Previously, `mussv` used a gain-based lower bound algorithm for both pure real and mixed uncertainty. Therefore, you might now obtain different results for the lower bounds with mixed uncertainty.

Option to specify feedback sign for stability margin calculation with ncfmargin

The `ncfmargin` command includes a new input argument that lets you specify the sign of the feedback interconnection assumed for the margin calculation. Use the syntax `[marg,freq] = ncfmargin(P,C,sign)` or `[marg,freq] = ncfmargin(P,C,sign,tol)` to specify a negative or positive feedback interconnection. For more information, see the `ncfmargin` reference page.

Compatibility Considerations

Previously, the relative accuracy `tol` was the third input argument to `ncfmargin`. If you have scripts or functions that use the syntax `[marg,freq] = ncfmargin(P,C,tol)`, update them to use `[marg,freq] = ncfmargin(P,C,-1,tol)` instead.

R2013a

Minimum damping requirement for closed-loop poles in TuningGoal.Poles object

You can now specify the minimum damping ratio of closed-loop poles for automated tuning of fixed-structure control systems with `systune` or `looptune`. To do so, create a `TuningGoal.Poles` object and set its `MinDamping` property to the minimum damping ratio you want to specify. Additionally, you can now use the `Focus` property to limit enforcement of the `TuningGoal.Poles` requirements to poles within a specified frequency range.

For more information about the `TuningGoal.Poles` requirement, see the `TuningGoal.Poles` reference page. For more information about using requirement objects to tune control systems, see Using Design Requirement Objects.

TuningGoal.Rejection object for specifying disturbance rejection requirement

You can now specify a disturbance rejection requirement for automated tuning of fixed-structure control systems with `systune` or `looptune`. The new `TuningGoal.Rejection` object allows you to specify a frequency-dependent attenuation factor for a disturbance injected at a specified location in the control system.

For more information about the `TuningGoal.Rejection` requirement, see the `TuningGoal.Rejection` reference page. For an example, see PID Tuning for Setpoint Tracking vs. Disturbance Rejection.

For more information about using requirement objects to tune control systems generally, see Using Design Requirement Objects.

looptune returns detailed results from multiple random starts

The `info` output of `looptune` now includes detailed results from each optimization run. When you use the `RandomStart` option of `looptuneOptions` to perform multiple optimization runs, the field `info.Runs` of the `info` output now contains a `struct` array. Each entry in the `struct` array includes results from the corresponding optimization run such as minimum constraint values and tuned block values. You can optionally use this information to analyze independent optimization results.

See the `looptune` reference page for more information.

Compatibility Considerations

The `Extra` field of `info` is now renamed to `Runs`. If you use `info.Extra` in a script, update your code to use `info.Runs` instead.

Additional automated tuning examples

New examples in this release include:

- Multi-Loop Control of a Helicopter
- Fault-Tolerant Control of a Passenger Jet

- Multi-Loop PID Control of a Robot Arm

R2012b

systune command for multiobjective tuning with soft and hard constraints

The new `systune` command allows automated tuning of fixed-structure control systems to high-level tuning objectives.

To use `systune`, you specify tuning objectives such as reference tracking, disturbance rejection, or stability margins. You can specify both soft requirements (objectives) and hard requirements (constraints). `systune` automatically tunes the parameters of your control system to meet the requirements.

You can use `systune` to tune control systems modeled in either MATLAB or Simulink.

For more information, see:

- Tuning Control Systems with SYSTUNE
- Tuning Control Systems in Simulink
- Automated Tuning
- The `systune` reference page

H2 performance, stability margin, pole location, and disturbance rejection requirements

New `TuningGoal` requirement objects allow you to specify a variety of tuning objectives for automated tuning of fixed-structure control systems with `systune` and `looptune`. New tuning requirements include:

- `TuningGoal.Margins` — Tune to stability margin requirements by specifying minimum gain and phase margins for any feedback loop in your control system.
- `TuningGoal.Poles` — Constrain closed-loop dynamics of your control system.
- `TuningGoal.StableController` — Constrain dynamics or ensure stability of tunable elements.
- `TuningGoal.WeightedGain` — Limit on frequency-weighted gain from specified inputs to specified outputs in your control system.
- `TuningGoal.Variance` and `TuningGoal.WeightedVariance` — Tune to H_2 performance requirements by minimizing or constraining variance amplification. `TuningGoal.Variance` specifies the maximum output variance for a unit-variance input signal from a specified input to a specified output in your control system. `TuningGoal.WeightedVariance` imposes a frequency-weighted variance amplification limit.

For more information about these `TuningGoal` requirement objects see the reference pages for each requirement object, and:

- Using Design Requirement Objects
- Specifying Design Requirements for systune
- Performance and Robustness Specifications for looptune

Robust tuning of one controller against a set of plant models

The new `systune` command can simultaneously tune the parameters of multiple models or control configurations. This feature allows you, for example, to tune a single controller against a range of plant models, to help ensure that the tuned control system is robust against parameter variations. As another example, you can tune for reliable control by simultaneously to multiple plant configurations that represent different failure modes of a system. In either case, `systune` finds values for tunable parameters that best satisfy the specified tuning objectives for all models.

For more information, see Tune Controller Against Set of Plant Models.

Option to constrain tuned parameter values and to restrict some tuning requirements to a frequency band

You can now optionally impose lower and upper bounds on tunable parameters when tuning fixed-structure control systems using `systune`, `looptune`, or `hinfstruct`. For example, you can constrain a gain to always be positive, or impose a maximum value on a filter time constant.

To impose bounds on tunable parameters, set the `Maximum` and `Minimum` properties of the parameter in the corresponding Control Design Block. For example, create a scalar gain block and constrain the gain to be positive:

```
gainblock = ltiblock.gain('gainblock',1,1);
gainblock.Gain.Minimum = 0;
```

Then, use `gainblock` as a component in a tunable `genss` model of the control system. When you tune the control system, the tuning command enforces the constraint.

Additionally, you can limit the range of frequencies in which almost any `TuningGoal` requirement is enforced for fixed-structure control system tuning with `systune` or `looptune`. The only exceptions are `TuningGoal.Variance` and `TuningGoal.WeightedVariance`.

For example, you can enforce a stability margin requirement in a frequency band extending for one decade on each side of the target gain crossover frequency.

To limit the range of frequencies in which a requirement is enforced, use the `Focus` property of the `TuningGoal` requirement object. For example, create a requirement that limits the gain from an input `du` to an output `u` to 10. Limit enforcement of the requirement to the frequency range 10–1000 rad/s.

```
Req = TuningGoal.Gain('du','u',10);
Req.Focus = [10 1000];
```

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 in Control System Toolbox 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

For more information, see the `ltiblock.pid2` and `loopswitch` reference pages.

TuningGoal.MaxGain and GainLimit property renamed

The tuning requirement `TuningGoal.MaxGain` is now called `TuningGoal.Gain`. Additionally, the `GainLimit` property of that tuning requirement is now called `MaxGain`.

For more information, see the `TuningGoal.Gain` reference page.

Compatibility Considerations

Replace instances of `TuningGoal.MaxGain` in your code with `TuningGoal.Gain`. Replace references to the `GainLimit` property with `MaxGain`.

Options in hinfstructOptions and looptuneOptions renamed or removed

The following options in `hinfstructOptions` and `looptuneOptions` are changed:

- `SpecRadius` is now called `MaxFrequency`. Additionally, `NaN` is no longer a supported value for this option. For an unconstrained `MaxFrequency` value, use `Inf`.
- `StableOffset` is now called `MinDecay`.
- `StableRadius` option has no effect.
- `StableExclude` option of `hinfstructOptions` has no effect. `hinfstruct` now automatically excludes from stability tests Control Design Blocks such as weighting functions or multipliers. These blocks do not affect the closed-loop stability of the actual control system to tune.

For more information about these options, see the `hinfstructOptions` and `looptuneOptions` reference pages.

Compatibility Considerations

If you use any of the affected options in your code, update your code to reflect the current names and supported values.

R2012a

Parallel Computing Support for looptune and hinfstruct

If you have Parallel Computing Toolbox™ software installed, you can use parallel computing to speed up tuning of fixed-structure control systems with the `looptune` or `hinfstruct` commands. When you run multiple randomized `looptune` or `hinfstruct` optimization starts, parallel computing speeds up tuning by distributing the optimization runs among MATLAB workers.

For more information about using parallel computing to speed up `looptune` or `hinfstruct` tuning, see:

- Speed Up Tuning with Parallel Computing Toolbox Software.
- The Robust Control Toolbox demo `Using Parallel Computing to Accelerate the Tuning Process`.

For more information about tuning fixed-structure control systems with `looptune` or `hinfstruct`, see Tuning Fixed Control Architectures.

Faster and More Accurate H-infinity Norm Computation Using SLICOT Algorithms

H_∞ norm calculations now use the SLICOT library of numerical algorithms. These algorithms improve the speed and accuracy of functions such as `hinfstruct` and `looptune`.

For more information about the SLICOT library, see http://slicot.org.

R2011b

looptune Tunes Fixed-Structure Control Systems

Use `looptune` to tune fixed-structure control systems to meet your requirements. To use `looptune`, specify design requirements such as loop bandwidth, stability margin, setpoint tracking, or target loop shape. `looptune` automatically tunes the parameters of your controller to meet the specified requirements.

The requirements objects `TuningGoal.MaxGain`, `TuningGoal.Tracking`, and `TuningGoal.LoopShape` let you express your design requirements directly. You do not have to first convert them to weighting functions or mathematical constraints on an optimization problem.

You can use `loopview` to validate the performance the performance of the tuned control structure against your specified design requirements.

For more information, see Tuning Fixed Control Architectures and the `looptune` and `loopview` reference pages.

Control System Tuning for Simulink Models with looptune or hinfstruct Using slTunable Interface

If you have Simulink Control Design software, you can use tuning commands, such as `slTunable.looptune` and `hinfstruct`, to tune control systems modeled in Simulink. The `slTunable` object provides an interface between your Simulink model and these commands.

Use `slTunable` to specify information about your control structure and parametrization. `slTunable` also automates tasks such as linearizing the Simulink model, parametrizing the tunable blocks of your system, and applying tuned parameter values to the model. After you create and configure an `slTunable` object for your control architecture, you can tune the control system using `slTunable.looptune` or `hinfstruct`.

For more information, see Tuning Fixed Control Architectures and the following demos:

- `Tuning of a Digital Motion Control System`
- `Decoupling Controller for a Distillation Column`
- `Tuning of a Two-Loop Autopilot`
- `Tuning of Cascaded PID Loops`
- `Loop Shaping Design with HINFSTRUCT`
- `Fixed-Structure Autopilot for a Passenger Jet`

wcgainplot for Visualizing Worst-Case Gains

`wcgainplot` plots the nominal, sampled, and worst-case gains of uncertain systems as a function of frequency. Use `wcgainplot` for visual analysis of uncertain systems.

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

Functionality Being Removed or Changed

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
`umat` object can no longer contain `ultidyn` or `udyn` uncertainty.	• Presence of `ultidyn` or `udyn` uncertain elements forces model type to `uss` or `ufrd` rather than `umat`. • Mixing `ureal` or `ucomplex` models with `udyn` or `ultidyn` objects produces `uss` instead of `umat`.	Expect a model type of `uss` or `ufrd` instead of `umat` when working with `udyn` or `ultidyn` uncertain elements.	Update code to work with `uss` or `ufrd` instead of `umat` when `udyn` or `ultidyn` elements are present.
`uss(sys_frd)`, where `sys_frd` is a `frd` model object no longer converts `sys_frd` to `ufrd`.	Errors.	`ufrd(sys_frd)`.	Replace `uss(sys_frd)` with `ufrd(sys_frd)`.
`ufrd(udat,freq,...)` no longer constructs an uncertain `frd` model from the `umat` object `udat`.	Converts `udat` to a `ufrd` object with frequencies `freq`.	Use `frd(udat,freq,...)` to construct an uncertain `frd` model from the `umat` object `udat`.	Replace `ufrd(udat,freq,...)` with `frd(udat,freq,...)`.
`frd(sys_uss,w)` where `sys_uss` is a `uss` model.	Warns; returns `frd` model containing data based on nominal response of `sys_uss`.	`ufrd(sys_uss,w)` to obtain a `ufrd` model.	Replace `frd(sys_uss,w)` with `ufrd(sys_uss,w)`.
Nominal value of `ultidyn` object.	Nominal value is `ss` model object.	None.	Update code to work with `ss` model objects when working nominal value of `ultidyn`.
`usubs`.	Applied to array of uncertain models, default substitution is `'-once'`.	Use `'-batch'` to perform batch substitution on uncertain model arrays.	Replace `usubs(...)` with `usubs(...,'-batch')`.
	`usubs(M,{a1;a2;...},{v1;v2;...})` returns error.	`usubs(M,a1,v1,a2,v2,...)`.	Replace `usubs(M,{a1;a2;...},{v1;v2;...})` with `usubs(M,a1,v1,a2,v2,...)`.
`usample(sys,'a',na,'b',nb)` where uncertain element `b` does not exist in `sys`.	Returns `na`-by-`nb` array with constant values across `nb` dimension, instead of `na`-by-1 array.	None.	Update code to reflect correct dimensionality.
`wcgopt`.	Still runs.	`wcgainOptions` or `wcmarginOptions`.	Replace `wcgopt` with `wcgainOptions` or `wcmarginOptions`.

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
`robuststab` and `robustperf`.	For `ufrd` models, `BadUncertainValues` field of `Info` output returns `Nf`-by-1 struct array, where `Nf` is the number of frequency points.	None.	Update code to work with `Nf`-by-1 struct array for `BadUncertainValues` instead of `Nf`-by-1 cell array.
	For nominally unstable models, performance margin is zero (instead of a negative value).	None.	Update code to reflect correct performance margin .
`robopt`.	Still runs.	`robuststabOptions` or `robustperfOptions`.	Replace `robopt` with `robuststabOptions` or `robustperfOptions`.
`actual2normalized`.	First output argument is normalized uncertain block value. The second output argument is normalized distance between block value and nominal value.	`[NV,ndist] = actual2normalized(BLK,AV)`.	Use second output argument `ndist` for normalized distance.
`reshape(unc_sys,S)`.	`S` does not include the I/O size of the models in the array `unc_sys`. For example, if `unc_sys` is a 6-by-1 array of 2-output, 4-input models, `reshape(unc_sys,[2 3])` converts `unc_sys` to a 2-by-3 array.	None.	Remove I/O size dimensions from `reshape` on uncertain model arrays.
`diag(uss_sys)` where `uss_sys` is a `uss` model.	Errors.	None.	Remove `diag(uss_sys)`.

R2011a

Version: 3.6

New Features

Bug Fixes

Enhanced Workflow for H-Infinity Synthesis of Fixed-Structure Control Systems

New Generalized LTI models in Control System Toolbox allow you to model control systems with tunable parameters. Using these models simplifies controller tuning with `hinfstruct`. You can model a closed-loop transfer function, including tunable parameters, as a generalized state-space (`genss`) model and directly tune the parameters to minimize the closed-loop gain. The `hinfstruct` command can tune any fixed-structure SISO or MIMO control system using H_∞ synthesis techniques.

Additionally, new `realp` and `genmat` objects let you create parametric expressions. You can use such expressions to create custom tunable components. For example, you can define a low-pass filter parametrized by its cutoff frequency, or an observer-based controller parametrized by the state-feedback and observer gains.

For more information about creating tunable Generalized LTI models, see Models with Tunable Coefficients in the *Control System Toolbox User's Guide*.

For more information about H_∞ tuning with `hinfstruct`, see Tuning Fixed Control Architectures in the *Robust Control Toolbox Getting Started Guide*.

For examples of designing controllers for several different architectures using `hinfstruct`, see the following updated and new demos:

- `Loop Shaping Design with HINFSTRUCT` (updated)
- `Tuning of a Fixed-Structure Autopilot` (updated)
- `Decoupling Controller for a Distillation Column` (updated)
- `Multi-Loop PID Control of a Robot Arm` (updated)
- `Fixed-Structure Autopilot for a Passenger Jet` (new)

R2010b

New Commands for H-Infinity Synthesis of Fixed-Structure Control Systems

New commands in this release allow you to tune fixed-structure SISO and MIMO control systems using the techniques of H_∞ synthesis.

The new `hinfstruct` command lets you use the frequency-domain methods of H_∞ synthesis to tune control systems with a broad range of architectures and controller structures. For example, you can tune:

- Fixed-order, fixed-structure controllers, such as pure gains, PID controllers, or fixed-order transfer function or state-space models
- Single feedback-loop architectures with multiple tunable elements, such as a PID controller plus a filter
- Multiple feedback-loop architectures with multiple tunable elements

Specify the tunable elements of your system using the new parametrized Control Design blocks `ltiblock.gain`, `ltiblock.pid`, `ltiblock.tf`, and `ltiblock.ss`.

For examples of designing controllers for several different architectures using `hinfstruct`, see the following new demos:

- `Loop Shaping Design with HINFSTRUCT`
- `Tuning of a Fixed-Structure Autopilot`
- `Decoupling Controller for a Distillation Column`
- `Multi-Loop PID Control of a Robot Arm`

For more information, see Tuning Fixed Control Architectures in the *Robust Control Toolbox Getting Started Guide*.

R2010a

Version: 3.4.1

Bug Fixes

R2009b

New Option to Improve Robust Performance by Accounting for Real Uncertain Parameters

You can now improve robust performance by accounting for real uncertain parameters when designing controllers using µ-synthesis. The user-defined options you use in the `dksyn` command now includes a new option `MixedMU`. Set this option to `'on'` to account for real uncertain parameters in your system. For more information, see the `dkitopt`, and `dksyn` reference pages.

New Command to Linearize Simulink Models with Uncertainty

If you have Simulink Control Design software installed, you can take model uncertainty into account when linearizing a Simulink model. You can then use the resulting uncertain linearized model (`uss` object) to perform linear analysis and robust control design.

If your model already contains Uncertain State Space blocks, use the new `ulinearize` command to obtain an `uss` model. If you want to account for uncertainty in your linear analysis without using Uncertain State Space blocks, you can specify individual Simulink blocks to linearize to an uncertain variable. For more information, see "Computing Uncertain State-Space Models from Simulink Models" in the *Robust Control Toolbox User's Guide*.

New Interface for Simulating Effects of Uncertainty in Simulink Models

This version of the product provides a new interface to simulate the effects of uncertainty in Simulink models. The interface includes the following:

- Uncertain State Space block to specify uncertain system in Simulink. You should replace USS System blocks in your existing models with the Uncertain State Space block. To do so, run the `slupdate` command on your models.
- `ufind` command to extract all uncertain variables from a Simulink model.
- `usample` command to generate random values of these uncertain variables.

For more information on simulating the effects of uncertainty using the new interface, see "Simulating Effects of Uncertainty" in the *Robust Control Toolbox User's Guide*.

New Command to Model Multiple LTI Responses as One Uncertain System

This version of the product includes a new `ucover` command that lets you model a family of LTI responses as one uncertain system. For more information, see the `ucover` reference page.

New and Updated Demos

The following new and updated demos illustrate use of the new features:

- `Control of Spring-Mass-Damper Using Mixed mu-Synthesis` shows use of the new `MixedMU` option and `dksyn` command for mixed-mu synthesis.
- `Linearization of Simulink Models with Uncertainty` shows how to compute uncertain state-space models using `ulinearize` and Simulink Control Design software.
- `Robustness Analysis in Simulink` uses the new interface for simulating effects of uncertainty in Simulink models.

- `Simultaneous Stabilization Using Robust Control` and `Modeling a Family of Responses as an Uncertain System` show use of the `ucover` command.
- `First-Cut Robust Design` shows use of the `usample`, `ucover` and `dksyn` commands.

To access the demos, type

```
demo('toolbox','robust control')
```

Functions, Properties and Blocks Being Removed

Function, Property or Block Name	What Happens When You Use Function or Property?	Use This Instead	Compatibility Considerations
`usiminfo`	Still runs	`ufind`	See “New Interface for Simulating Effects of Uncertainty in Simulink Models” on page 25-2.
`usimfill`	Still runs	`ufind`	See “New Interface for Simulating Effects of Uncertainty in Simulink Models” on page 25-2.
`usimsamp`	Still runs	`usample`	See “New Interface for Simulating Effects of Uncertainty in Simulink Models” on page 25-2.
USS System block	Still runs	Uncertain State Space block	See “New Interface for Simulating Effects of Uncertainty in Simulink Models” on page 25-2.
`ltiarray2uss`	Still runs	ucover	See “New Command to Model Multiple LTI Responses as One Uncertain System” on page 25-2.

R2009a

Version: 3.3.3

Bug Fixes

R2008b

Version: 3.3.2

Bug Fixes

R2008a

Version: 3.3.1

New Features

Ability to Use LOOPMARGIN with Simulink

This version of Robust Control Toolbox software lets you analyze the robustness of nonlinear Simulink models using the LOOPMARGIN command.

If you have the Simulink Control Design product installed, you can perform stability margin analysis of a Simulink model by passing the model name and a point within that model to the LOOPMARGIN command.

R2007b

Version: 3.3

No New Features or Changes

R2007a

Version: 3.2

New Features

New Simulink Blocks

- USS System — This Robust Control Toolbox version introduces a new Simulink block, USS System. You can use this block to import uncertain systems into Simulink models.
- Multiplot Graph — Plot multiple signals in one figure.

31

R2006b

Version: 3.1.1

New Features

New Function ltiarray2uss

This Robust Control Toolbox version introduces a new function, `ltiarray2uss`. This function constructs an uncertain state-space model from an LTI array.

R2006a

Version: 3.1

No New Features or Changes

33

R14SP3

Version: 3.0.2

No New Features or Changes

R14SP2

Version: 3.0.1

New Features

mussvunwrap Is Renamed

`mussvunwrap` has been renamed. It is now called `mussvextract`.

New Functions actual2normalized and normalized2actual

This Robust Control Toolbox version introduced two new functions:

- `actual2normalized` — Calculate normalized distance between nominal value and given value for uncertain atom.
- `normalized2actual` — Convert value for atom in normalized coordinates to corresponding actual value.