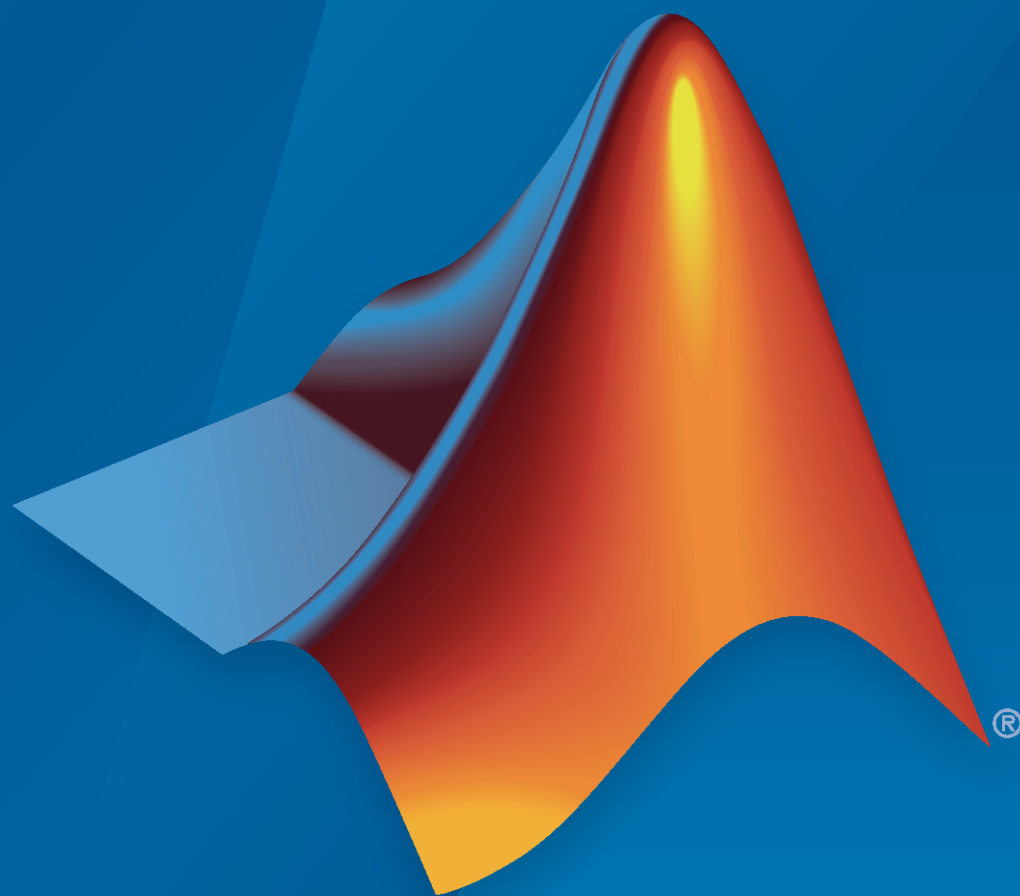


Fixed-Point Designer™ Release Notes



MATLAB®



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

Fixed-Point Designer™ Release Notes

© COPYRIGHT 2013–2022 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.

R2022b

Manage Simulink parameter diagnostics related to numeric issues	1-2
New Euler to North-East-Down Transformation HDL Optimized block . .	1-2
Improved numerical accuracy and generated code efficiency for relational operations	1-2
fixed.svd and svd Functions: Fixed-point singular value decomposition	1-3
Lookup Table Optimizer support for curve fitting objects	1-3
Reduced latency of partial-systolic QR decomposition and matrix solve blocks	1-3
New blocks for burst Q-less QR decomposition and asynchronous matrix solve	1-3
New Simulink edit-time and Model Advisor check for numeric efficiency	1-4
Expanded support for half-precision data type	1-5
fixed.realConditionNumberUpperBound and fixed.complexConditionNumberUpperBound Functions: Analytically determine an upper bound for condition number	1-5
fixed.singularValueUpperBound Function: Upper bound for largest singular value of matrix	1-5
Specify maximum word length in functions for analytically determining fixed-point data types	1-5
fixed.fimathLike: Create fimath object like input	1-6
Linear system solver and matrix factorization blocks use AMBA AXI handshake protocol	1-6
Updated Numeric Type Scope Interface	1-6
New Fixed-Point Designer Examples	1-6

Expanded support for Tikhonov regularization parameter in linear system solver and matrix factorization blocks and functions	2-2
Lookup table optimization support for curve fitting objects	2-3
Lookup table optimization improved memory reduction for 1D and flat interpolation	2-3
Estimate cost of generated code for Simulink models	2-3
Improved numerical accuracy and generated code efficiency for fixed-point division with mixed signedness and slope and bias scaling	2-3
Improved generated code fixed-point division by zero protection	2-4
Improved numerical accuracy and generated code efficiency for fixed-point operations that do not lose precision	2-5
Implicit expansion for logical operators, bitwise operations, and division, automatically expand dimensions of length 1	2-5
Improved accuracy in comparing fi objects and floating-point numbers using relational operators	2-6
GPU code generation support for half-precision data types in MATLAB Function blocks	2-7
New functions and syntax supported for half-precision inputs	2-7
fi bitset now supports scalar expansion	2-7
New Fixed-Point Designer Examples	2-8

Generate native half-precision C code for embedded hardware targets	3-2
Rapid Accelerator mode support for half-precision floating-point data types in Simulink	3-2
Generate an optimized lookup table approximation as a MATLAB function	3-2
Improved numerical accuracy and generated code efficiency for fixed-point multiplication with slope and bias scaling	3-2

Improved numerical accuracy and generated code efficiency for fixed-point division with slope and bias scaling	3-2
Implicit Expansion: For fi plus, minus, and times, automatically expand dimensions of length 1	3-3
Fixed-Point Tool: Pause and resume data type optimization search, import fxpOptimizationOptions object, and guided workflow selection	3-3
Pause and resume data type optimization search	3-3
Fixed-Point Tool: Import fxpOptimizationOptions object	3-3
Fixed-Point Tool provides guided workflow selection	3-4
Data Type Optimization: Restrict instrumentation to a subsystem, enforce known data types, maintain model parameter settings, and warn about unsupported constructs	3-4
Restrict instrumentation to a subsystem	3-4
Enforce known data types for variables in a system	3-4
Maintain original values of model parameters that are altered by fxpopt	3-5
Warn about unsupported constructs	3-5
Analytically determine fixed-point data types when solving linear systems of equations	3-5
fixed.cordicDivide and fixed.cordicReciprocal Functions: Fixed-point divide and reciprocal using CORDIC	3-6
New functions supported for half-precision inputs	3-6
fi support for dec2base, dec2bin, and dec2hex	3-6
Data Type Optimization: Specify multiple types of tolerances	3-6
New Fixed-Point Designer Examples	3-7
Functionality being removed or changed	3-7
Change in default behavior of quantizenumeric for complex input	3-7
Change in rounding behavior for quantize function	3-8

R2021a

Half-precision data type support for MATLAB Function blocks	4-2
New HDL-optimized Simulink blocks for reciprocal, divide, and modulo	4-2
New Simulink blocks and MATLAB functions for divide and modulo	4-2
Improved numerical accuracy and generated code efficiency for cast operations	4-2

Generate optimized one-dimensional lookup tables for HDL applications	4-3
New Fixed-Point Designer Examples	4-3
Reduced HDL resource utilization in fixed-point matrix library blocks	4-4
fixed.extractNumericType function: Extract numeric type of input	4-4
Generate C++ code for half-precision floating-point data types in Simulink	4-4
Control inherited block output data type for half-precision	4-4
Fixed-Point Tool: View optimization details, visualize data types, and manually stop optimization	4-4
View optimization details in the Fixed-Point Tool	4-4
Data type visualizer: Understand and analyze optimized data types by viewing histograms of the dynamic ranges of signals in your model	4-5
Stop data type optimization	4-6
Lookup table optimization support for functions with scalar inputs	4-6
Improved lookup table value optimization	4-6
Improved numerical accuracy and generated code efficiency for fi inputs to power, .^	4-6
Data type optimization workflow improvements	4-7
Override data types with scaled doubles	4-7
Log a reduced set of data points	4-7
Stop optimization in Lookup Table Optimizer app	4-7
New Fixed-Point Designer Simulink block library	4-7
Functionality being removed or changed	4-7
Inexact property names for fi, fimath, and numeric type objects not supported	4-7

R2020b

Half Precision in Simulink: Design, simulate, and generate code for half-precision systems	5-2
Expanded half-precision support for Deep Learning Toolbox and FFT functions	5-2
Explore half precision in optimized lookup tables	5-2

New API functions for half-precision data type support in user-written S-functions	5-2
New QR decomposition and matrix solve Simulink blocks	5-3
New QR decomposition and matrix solve MATLAB functions	5-3
Optimize data types based on operator counts	5-4
Export optimization workflow steps to a MATLAB script	5-5
Automatically propagate slope-bias data types during data type optimization	5-5
Data type optimization workflow improvements	5-5
Automatically isolate constructs not supported for fixed-point conversion	5-5
Override data types in range collection step of optimization	5-5
Inspect optimization solutions using Simulation Manager	5-6
Functionality being removed or changed	5-6
Change in default behavior of fi for -Inf, Inf, and NaN	5-6
Change in default data type override in the Fixed-Point Tool	5-6

R2020a

Half Precision: Design, simulate, and generate code for half-precision systems	6-2
Half precision code generation in MATLAB	6-2
Tech Preview: Half precision in Simulink	6-2
Fixed-Point Tool: Convert and optimize data types, and explore ranges	6-2
New Fixed-Point Designer Simulink block library	6-2
Math Operations	6-2
Matrix Operations	6-2
Lookup Table Optimization: Iterative redesign and batch compression of lookup tables, parallelization of lookup table optimization	6-3
Iteratively redesign lookup tables in your model	6-3
Automatically compress all lookup tables in a system	6-3
Parallelized lookup table optimization	6-3
Data Type Optimization: Specify a safety margin, enforce known data types, and other enhancements	6-3
Review all changes made during optimization	6-3
Specify a safety margin for optimization	6-4
Enforce known data types in a system	6-4
Revert optimization	6-4

Coder Type Editor: Create and edit input types interactively	6-5
normalizedReciprocal: Compute the normalized reciprocal	6-5
nextpow2: Compute the next-higher power of 2 of fixed-point values ...	6-5
Improved numerical accuracy for slope-bias scaled fixed-point operations	6-5
Generate test data as a dataset	6-6
Functionality being removed or changed	6-6

R2019b

Propose data types based on multiple simulation scenarios in the Fixed-Point Tool	7-2
Restore model to original design	7-3
Quantize and generate fixed-point C/C++ code for a trained SVM model (requires MATLAB Coder and Statistics and Machine Learning Toolbox)	7-3
Allow off-curve table values in optimized lookup tables	7-3
Generate optimized AUTOSAR-compliant lookup table	7-4
Generate simulation inputs to test full operating range of design	7-4
Features under tech preview	7-5
Tech Preview: HDL-optimized fixed-point matrix operations blocks	7-5
Tech Preview: Half-precision data types in Simulink	7-5

R2019a

Emulate hardware handling of denormal numbers	8-2
New data type propagation rules for Sum, Gain, and Product blocks	8-2
Automatically prepare Simulink systems for conversion to fixed point ..	8-2
Complex support for half-precision	8-2
Specify multiple simulation scenarios for data type optimization	8-3

Lookup table optimization options available in the app	8-3
Specify new constraints for lookup table optimization	8-3
Derived range analysis support for fixed-point optimization	8-3
Specify tolerances of signals in system for conversion	8-4
New functions supported for half-precision inputs	8-4

R2018b

Lookup Table Optimization: Automatically replace subsystems with a direct lookup table and other enhancements	9-2
Approximate a Subsystem with a lookup table	9-2
Generate a direct lookup table to approximate a function or subsystem ..	9-2
Generate a lookup table approximation from a function handle using the Lookup Table Optimizer app	9-2
Generate lookup tables with flat and nearest interpolation methods	9-2
Automatically replace blocks with an optimized lookup table block	9-2
Data Type Optimization: Using parallel simulations, automatically select and apply heterogeneous data types for your system under design ...	9-2
Parallel support for data type optimization	9-2
New method for specifying required behavior of optimized design	9-2
Single Precision Converter: Convert MATLAB Function blocks to single precision	9-3
cordicos and cordicasin Functions: Compute fixed-point CORDIC inverse sine and cosine	9-3
Simulation Analysis and Performance: Instrumentation support for Fast Restart mode	9-3
Explore and debug Fixed-Point Tool results with sorting and filtering functionalities	9-3
Design and simulate half-precision systems in MATLAB	9-4

R2018a

Lookup table optimization: Approximate functions using a lookup table and optimize existing lookup tables to minimize RAM usage	10-2
Data type optimization: Automatically select and apply heterogeneous data types for your system under design, optimizing bit width.	10-3

Redesigned code generation reports: View fiaccel and instrumentation results with improved user interface	10-3
--	-------------

R2017b

Simplified Fixed-Point Tool: Convert Simulink systems to fixed point using the updated tool that provides guidance at each step of the workflow	11-2
Data Type Visualizer: Understand and analyze data type choices by viewing histograms of the dynamic range of signals in your model	11-2
Data Type Exploration: Iteratively explore multiple floating point to fixed-point conversions to determine the optimal choice	11-3
Function Input and Output Logging: Selectively log and plot function inputs and outputs at any level of your design in the Fixed-Point Converter app	11-3
Simulink Diagnostic Management: Suppress immaterial diagnostic warnings and errors from specific blocks to efficiently discover modeling errors	11-5
Expanded Overflow Diagnostics: Comprehensive run-time diagnostics for wrapping and saturating overflows from Stateflow and MATLAB Function blocks	11-6
Autoscaling Lookup Table Objects: Propose and apply fixed-point data types for Simulink Lookup Table and Breakpoint objects	11-6
Check for expensive fixed-point data types in generated code	11-6
Propose and apply data types for model reference blocks programmatically	11-6
cordictanh function for computing fixed-point CORDIC-based hyperbolic tangent	11-6
Functionality being removed or changed	11-7

R2017a

Simulink Diagnostic Management: Control which simulation and fixed-point diagnostic warnings you receive from specific blocks, including model reference	12-2
Select blocks with certain diagnostic suppressions by default	12-2

Diagnostic suppressor functions support MSLDiagnostic as input argument	12-2
Improved workflow for suppressing diagnostics from referenced models	12-2
Derived range analysis support for System objects in Simulink	12-3
Autoscaling support for Simulink.AliasType objects	12-3
Improved data type proposals for shared data type groups across model reference	12-3
More fixed-size variable information in Convert to Fixed-Point step of the Fixed-Point Converter app	12-3
fimath property changes	12-4

R2016b

Single-Precision Conversion: Automatically convert double-precision systems to use single-precision data types in Simulink	13-2
Float to Fixed Conversion of MATLAB Function Blocks: Automatically generate fixed-point versions of floating-point MATLAB Function blocks	13-2
Histogram Instrumentation in Simulink: Generate log2 histograms of Simulink signals and blocks from simulation data	13-3
Autoscaling numerictype Objects: Propose and apply fixed-point data types for Simulink numeric type objects	13-5
Range analysis support for FIR filters, Dead Zone, and Rate Limiter blocks	13-5
Simulink Diagnostic Suppressor	13-5
Reduced number of multiplication helper functions	13-5
Improved accuracy of fixed-point sin, cos, and mod functions	13-5
Improved workflow for collecting and analyzing ranges in the Fixed-Point Converter app	13-6

Autoscaling Parameter Objects: Automatically propose and apply data types for parameter objects	14-2
View and edit fi objects in Model Explorer	14-2
Simulate system level designs that integrate referenced models targeting an assembly of heterogeneous embedded devices	14-2
Enhancements to Fixed-Point Converter app	14-3
Support for arrays of structures	14-3
Structures in generated fixed-point code	14-3
Revert changes to input type definitions	14-3
View complete error message in error table	14-3
Additional keyboard shortcuts in the code generation report	14-3
Changes to Fixed-Point Conversion Code Coverage	14-4

Bug Fixes

Simulink Fixed-Point Tool workflow simplification: Propose signedness and data types for inherited and floating-point types	16-2
System under design (SUD) specification	16-2
Signedness proposals	16-2
Proposals for objects using inherited and floating-point types	16-2
Two-way traceability between model and Fixed-Point Tool	16-3
New configurations for model settings	16-3
Double-precision to single-precision conversion: Convert double-precision MATLAB code to single-precision MATLAB code using the command line	16-4
MATLAB Fixed-Point Converter app streamlined workflow: Restore project state and minimize regeneration of MEX files	16-4
Saving and restoring fixed-point conversion workflow state in the app ..	16-4
Minimized regeneration of MEX files	16-5
Specification of additional fimath properties in app editor	16-5
Improved management of comparison plots	16-5
Variable specializations	16-6
Improvements to Readability of Generated Code	16-7
Tab completion for specifying files	16-8

Improvements for manual type definition	16-8
Compatibility between the app colors and MATLAB preferences	16-9
Range analysis for Delay blocks: Improve accuracy and speed of range analysis on models using Delay blocks	16-9
Control of signed shifts in fixed-point scaling operations: Control the use of signed shifts in generated code	16-9
MATLAB	16-9
Simulink	16-10
Access full-precision value of fi object in decimal and string format ..	16-10
Detection of multiword operations	16-10
MATLAB	16-10
Simulink	16-10
Enhanced Model Advisor check for implementing strict single-precision designs	16-11
System object instrumentation in Fixed-Point Tool	16-11

R2015a

Derived Ranges for MATLAB Function Blocks in Simulink	17-2
Fixed-Point Converter app enhancements, including detection of dead and constant folded code, support for projects with multiple entry point functions and support for global variables	17-2
Support for projects with multiple entry-point functions	17-2
Support for global variables	17-2
Code coverage based translation	17-2
Conversion from project to MATLAB scripts for command-line fixed-point conversion	17-2
Generated fixed-point code enhancements	17-2
Integration with MATLAB Coder app interface	17-3
Automated conversion of additional DSP System objects using the Fixed-Point Converter app	17-3
Fixed-Point SimState logging and root logging improvements	17-3
Flexible structure assignment of buses	17-3
eye(m,'like',a) syntax supported for fixed-point inputs	17-3
New interpolation method for generating lookup table MATLAB function replacements	17-4
Fixed-point scaling information in Code Interface Report	17-4

Fixed-Point Converter app for automated conversion of floating-point MATLAB code	18-2
Commands for scripting fixed-point conversion and accessing the collected data in Simulink	18-2
Automated fixed-point conversion for commonly used DSP System objects, including Biquad Filter, FIR Filter, and FIR Rate Converter	18-2
Simulation range collection and data type proposals for MATLAB Function blocks in Simulink	18-3
Overflow diagnostics to distinguish between wrap and saturation in Simulink	18-3
Highlighting of potential data type issues in generated HTML report	18-3
Code generation of for loops using fixed-point loop indices	18-3
Cast net slope computations using rational numbers	18-3
Lock Column View option in the Fixed-Point Tool	18-4
Fixed-Point Advisor enhancements	18-4
hdlram renamed hdl.RAM	18-4
Changes to data type strings	18-4
Signal data type display	18-4
tostring function now uses 0 and 1 to represent signedness	18-4
New featured examples	18-5

Data type override and automatic data typing for bus objects	19-2
Data type override for bus objects	19-2
Autoscaling for bus objects	19-2
Derived ranges for complex signals in Simulink	19-2
cordicsqrt function for fixed-point CORDIC-based square root functionality	19-2

Overflow detection with scaled double data types in MATLAB Coder projects	19-2
Fixed-point ARM Cortex-M code replacement support for DSP System Toolbox FIR filters	19-3
Fixed-Point Advisor support for referenced configuration sets	19-3
Enhancements to automated conversion of MATLAB code	19-3
Support for MATLAB classes	19-3
Generated fixed-point code enhancements	19-3
Fixed-point report	19-3
Automatic C compiler setup	19-3
More flexible control of dsp.LMSFilter System object fixed-point settings	19-4
Derived ranges for For Each and For Each Subsystem blocks	19-4

R2013b

C99 long long integer data type for embedded code generation	20-2
Model Advisor fixed-point checks with additional coverage and optimization awareness	20-2
fi object as an index in colon expressions and an argument to numel and bit index functions	20-2
fi object as an index in colon expressions	20-2
fi objects as bit index input argument	20-2
fi objects as shift-value input argument	20-3
numel function support for fi inputs	20-3
Improved efficiency of data type internal rules for Lookup Table blocks	20-3
Derived ranges for complex variables in MATLAB Coder projects	20-3
Simplified modeling of single-precision designs	20-3
Range analysis support on Mac platforms	20-4
Changes to showInstrumentationResults function options	20-4
New option to suppress display of MATLAB code	20-4
Removal of -browser option	20-4
Changes to Continuous state-space block family range analysis support	20-5

Enhanced fiaccel support for int64 and uint64 functions	20-5
Support for LCC compiler on Microsoft Windows (64-bit) machines ...	20-5
Warning for use of inexact fi and fimath property names	20-5
Conversion of numeric variables into Simulink.Parameter objects	20-5
Fixed-point conversion test file coverage results	20-6
Fixed-point conversion workflow supports designs that use enumerated types	20-6
Fixed-point conversion of variable-size data using simulation ranges	20-6
Error checking improvements for bitconcat, bitandreduce, bitorreduce, bitxorreduce, bitsliceget functions	20-6
Legacy data type specification functions return numeric objects	20-6
numberofelements function being removed in a future release	20-8

R2013a

Product restructuring	21-2
Histogram logging in instrumented MATLAB Code Generation report	21-2
fi object in indexing and switch-case expressions	21-2
zeros, ones, and cast code reuse for floating-point and fixed-point types	21-2
Code generation for x.^n when n is a variable and x is a fi object	21-3
Fixed-Point Advisor support for model reference	21-3
Automated conversion of floating-point to fixed-point types in MATLAB Coder projects	21-3
Improved autoscaling for models with virtual bus signals	21-4
Data Type Override for MATLAB Function block using built-in doubles and singles	21-4
Instrumentation for arrays of structs	21-4
File I/O function support	21-4

Support for nonpersistent handle objects	21-5
Load from MAT-files for code acceleration	21-5
New toolbox functions supported for code acceleration and generation	21-5
Function to be removed in a future release	21-6
Function being removed	21-6

R2022b

Version: 7.5

New Features

Bug Fixes

Compatibility Considerations

Manage Simulink parameter diagnostics related to numeric issues

Diagnostic messages for parameter overflow, underflow, and precision loss now provide more numerically rich information to help you identify the cause of the warning or error. You can choose to disable or suppress the warning or error directly from the suggested actions of the diagnostic message or explore more details in the **Parameter Quantization Advisor**.

The screenshot shows the Parameter Quantization Advisor window. On the left, there are filters for 'Overflow', 'Underflow', and 'Precision Loss'. The main table lists parameters with their quantization issues, dialog values, quantized values, bits of error, absolute errors, and relative errors. A legend on the right indicates the color coding for each issue: red for Overflow, yellow for Underflow, and blue for Precision Loss.

Parameter Name	Quantization Issue	Dialog Value	Quantized Value	Bits of Error	Abs Error	Rel Error (%)
Value[1 1]	Overflow	-2048	0	2	2048	100
Value[1 29]	Overflow	9216	7168	-2	2048	22.22222222222222
Value[1 28]	Overflow	8813.714285714286	7168	-1.6071428571428577	1645.7142857142862	18.672199170124486
Value[1 2]	Overflow	-1645.7142857142858	0	1.6071428571428572	1645.7142857142858	100
Value[1 3]	Overflow	-1243.4285714285716	0	1.2142857142857144	1243.4285714285716	100
Value[1 27]	Overflow	8411.42857142857	7168	-1.2142857142857135	1243.4285714285706	14.782608695652167
Value[1 4]	Underflow	-841.1428571428571	0	0.8214285714285714	841.1428571428571	100
Value[1 26]	Precision Loss	8009.142857142857	7168	-0.8214285714285712	841.1428571428569	10.502283105022828
Value[1 15]	Precision Loss	3584	4096	0.5	512	14.285714285714285
Value[1 10]	Precision Loss	1572.5714285714284	2048	0.4642857142857144	475.42857142857156	30.232558139534895
Value[1 20]	Precision Loss	5595.428571428572	5120	-0.4642857142857144	475.42857142857156	8.496732026143793
Value[1 25]	Precision Loss	7606.857142857143	7168	-0.4285714285714288	438.8571428571431	5.769230769230773
Value[1 5]	Underflow	-438.8571428571429	0	0.4285714285714286	438.8571428571429	100
Value[1 23]	Precision Loss	6802.285714285714	7168	0.35714285714285765	365.71428571428623	5.376344086021513
Value[1 7]	Underflow	365.7142857142858	0	-0.3571428571428572	365.7142857142858	100
Value[1 12]	Precision Loss	2377.142857142857	2048	-0.3214285714285712	329.1428571428569	13.846153846153836
Value[1 18]	Precision Loss	4790.857142857143	5120	0.3214285714285712	329.1428571428569	6.870229007633583
Value[1 13]	Precision Loss	2779.4285714285716	3072	0.2857142857142856	292.57142857142844	10.52631578947368
Value[1 17]	Precision Loss	4388.571428571428	4096	-0.2857142857142856	292.57142857142844	6.666666666666666

The details pane on the right shows information for a selected parameter: Source: param_value_diagnostics_2/Const2, Parameter Name: Value, Dimension: [1 29], Complexity: Real. It also includes a table of properties:

Property	Dialog Data Type	Run Time Data Ty
Data Type	double	numeric(0,3,...
Minimum	-1.79769313486...	0
Maximum	1.797693134862...	7168
Precision	4.940656458412...	1024

New Euler to North-East-Down Transformation HDL Optimized block

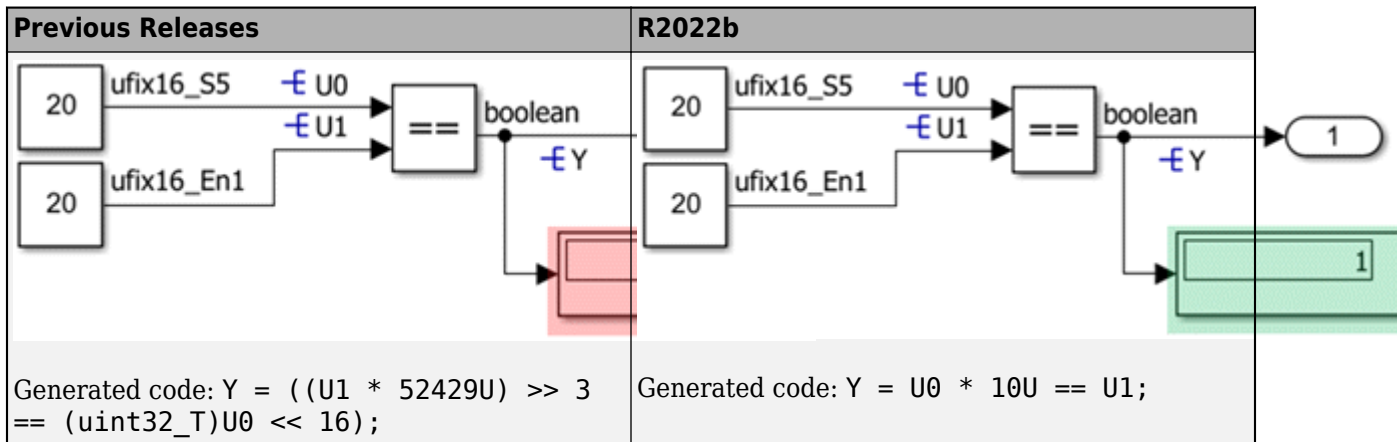
The Euler to NED Transformation HDL Optimized block computes the Euler to North-East-Down (NED) coordinate transformation using a hardware-efficient CORDIC rotation kernel. The block provides a pipelined or burst architecture and hardware-friendly control signals. This block supports HDL code generation using HDL Coder™.

Improved numerical accuracy and generated code efficiency for relational operations

Fixed-Point Designer now has improved numerical accuracy for relational operations involving certain combinations of fixed-point and integer data types. Additionally, generated code is more efficient and more readable.

Compatibility Considerations

Numerical outputs and generated code for relational operations involving fixed-point and integer data types may be more accurate than in previous releases. For example, this table illustrates the difference in generated code for this Simulink® model.



fixed.svd and svd Functions: Fixed-point singular value decomposition

Use the `fixed.svd` and `svd` functions to compute the singular value decomposition of fixed-point matrices. The `svd` function automatically adjusts the data type of fixed-point input to avoid overflow and increase precision. The `fixed.svd` function allows full control over the fixed-point types. You can use `fixed.svd` and `svd` to generate efficient, purely integer C code.

Lookup Table Optimizer support for curve fitting objects

The **Lookup Table Optimizer** now supports curve fitting `cf fit` objects as valid inputs for approximation.

Reduced latency of partial-systolic QR decomposition and matrix solve blocks

These blocks now have reduced latency as compared to previous releases.

Linear system solvers:

- Complex Partial-Systolic QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor

Matrix factorizations:

- Complex Partial-Systolic Matrix Solve Using QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

New blocks for burst Q-less QR decomposition and asynchronous matrix solve

Use these blocks to perform burst Q-less QR decomposition.

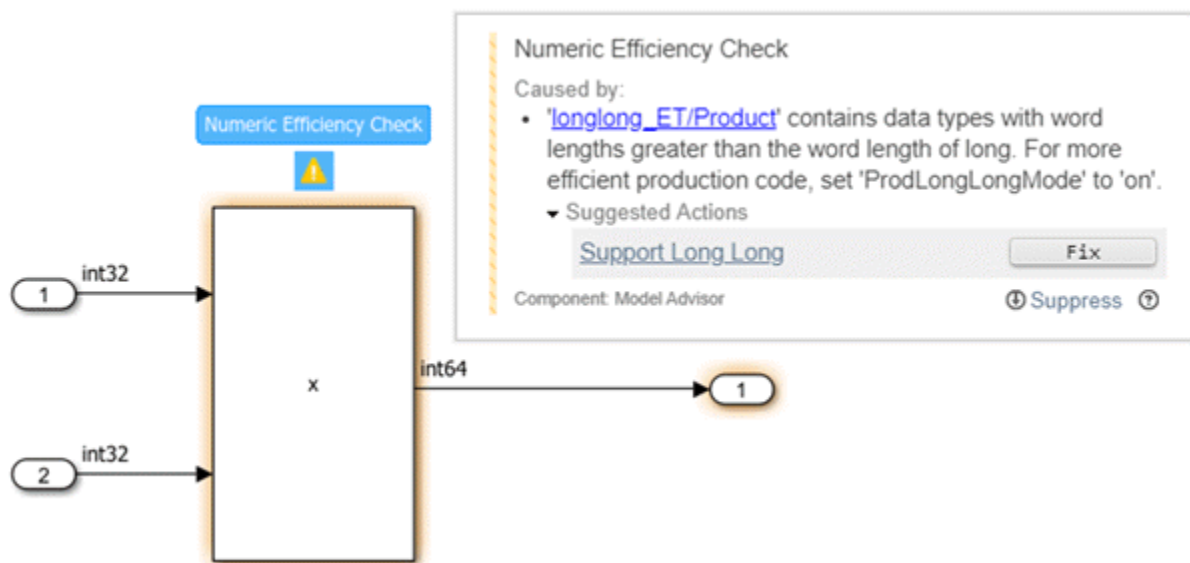
- Real Burst Q-less QR Decomposition Whole R Output
- Complex Burst Q-less QR Decomposition Whole R Output
- Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output
- Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output

Use these blocks to perform asynchronous matrix solve. The forward- and backward-substitution and Q-less QR decomposition run independently using the latest R and B matrices.

- Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition
- Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition
- Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor
- Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

New Simulink edit-time and Model Advisor check for numeric efficiency

You can now use Simulink edit-time checking to identify when code generated from a Simulink model will be more efficient if you enable the **Support long long** parameter. This numeric efficiency check alerts you when signals and ports in your model will result in expensive multi-word types in generated code because the `long long` data type is not enabled. The edit-time check includes a **Fix** button you can use to enable the **Support long long** parameter, or you can choose to suppress the warning. Edit-time checking requires a Simulink Check™ license. For more information on how to enable edit-time checking, see “Check Model Compliance Using Edit-Time Checking” (Simulink Check).



When you generate code for a model that could be made more efficient by enabling the **Support long long** parameter, a message is logged in the **Diagnostic Viewer** with an **Apply** button you can use to easily enable the parameter.

You can also access this check in the Model Advisor under **By Product > Embedded Coder > Check usage of 'long long' data type**. This Model Advisor check requires an Embedded Coder® license. For more information, see “Embedded Coder Checks” (Embedded Coder).

Only explicit usage of signals and ports having data types with word lengths greater than the `long` data type are detected. This check does not flag operation outputs that implicitly have word lengths greater than `long` data type, such as the output of a multiply operation.

Expanded support for half-precision data type

The following now support the half-precision data type:

- System object™
- MATLAB System block
- `hdl.RAM` System object

These functions now support half-precision inputs:

- `allfinite`
- `anynan`

For more information, see `half` and “Half Precision Code Generation Support”.

`fixed.realConditionNumberUpperBound` and `fixed.complexConditionNumberUpperBound` Functions: Analytically determine an upper bound for condition number

Use the `fixed.realConditionNumberUpperBound` and `fixed.complexConditionNumberUpperBound` functions to analytically determine an upper bound for the condition number of a matrix. You can use this value to choose the number of bits of precision required in the solution of a least-squares matrix equation.

`fixed.singularValueUpperBound` Function: Upper bound for largest singular value of matrix

Use the `fixed.singularValueUpperBound` function to analytically determine an upper bound for the largest singular value of a matrix.

`fixed.singularValueUpperBound`, `fixed.realSingularValueLowerBound`, and `fixed.complexSingularValueLowerBound` are used by the `fixed.realConditionNumberUpperBound` and `fixed.complexConditionNumberUpperBound` functions to analytically determine an upper bound for the condition number of a matrix.

Specify maximum word length in functions for analytically determining fixed-point data types

You can now use the `maxWordLength` parameter to specify the maximum word length of fixed-point types in these functions:

- `fixed.complexQlessQRMatrixSolveFixedpointTypes`
- `fixed.complexQRMatrixSolveFixedpointTypes`
- `fixed.qlessqrFixedpointTypes`

- `fixed.qrFixedpointTypes`
- `fixed.realQlessQRMatrixSolveFixedpointTypes`
- `fixed.realQRMatrixSolveFixedpointTypes`

This functionality allows you to specify word lengths larger than 128 bits in MATLAB® interpreted execution for `fixed.qrMatrixSolve` and related functions.

fixed.fimathLike: Create fimath object like input

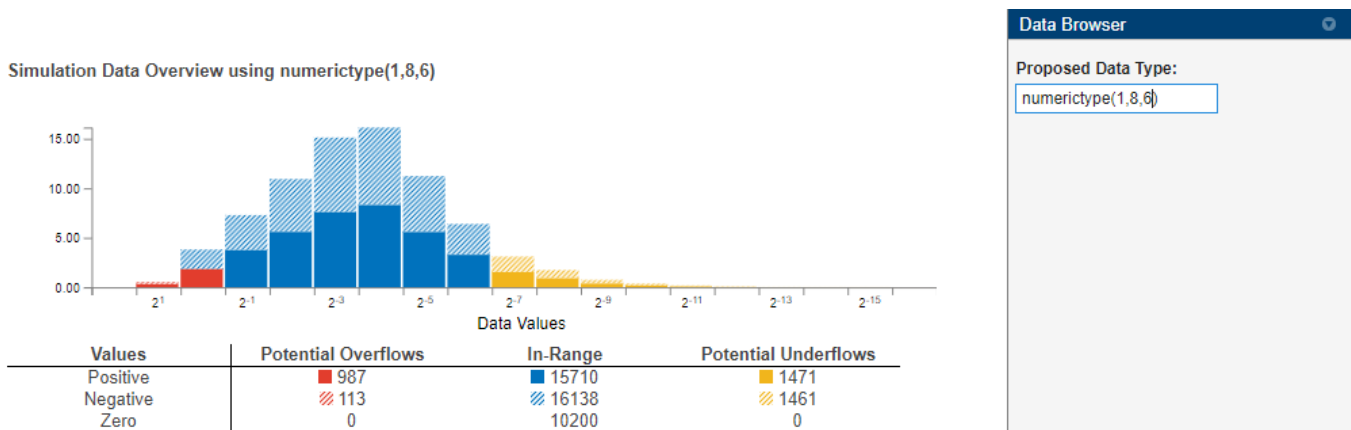
Use the `fixed.fimathLike` function to create a `fimath` object that has the same fixed-point math settings as the input. This allows you to specify that the fixed-point arithmetic be the same as the input, rather than requiring you to explicitly specify a `fimath` object and attach it to the input. `fixed.fimathLike` supports binary-point scaling and slope-bias scaling.

Linear system solver and matrix factorization blocks use AMBA AXI handshake protocol

All blocks in the **Fixed-Point Designer HDL Support > Matrices and Linear Algebra** library now use the industry-standard AMBA AXI handshake protocol. For more information, see “AMBA AXI Handshake Process”.

Updated Numeric Type Scope Interface

The **Numeric Type Scope** has a new, simplified interface when you launch the scope from the **Instrumentation Report Viewer** which shows an improved display of the histogram information. For an example, see `showInstrumentationResults`.



New Fixed-Point Designer Examples

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Matrix Factorizations** library:

- “Implement Hardware-Efficient Real Burst Q-less QR with Forgetting Factor”
- “Implement Hardware-Efficient Complex Burst Q-less QR with Forgetting Factor”

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Linear System Solvers** library:

- “Implement Hardware-Efficient Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor”
- “Implement Hardware-Efficient Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor”
- “Implement Hardware-Efficient Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition”
- “Implement Hardware-Efficient Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition”

New examples to help you get started with **Design Evolution Manager** in MATLAB Online™:

- “Use Design Evolution Manager with the Fixed-Point Tool”

R2022a

Version: 7.4

New Features

Bug Fixes

Compatibility Considerations

Expanded support for Tikhonov regularization parameter in linear system solver and matrix factorization blocks and functions

The Tikhonov regularization parameter is now supported for all linear system solver and matrix factorization blocks and functions. This parameter is also supported for all functions for analytically determining fixed-point data types for linear system solvers and matrix factorizations.

Use of the Tikhonov regularization parameter can improve the conditioning of an ill-posed problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates. Use of the Tikhonov regularization parameter can also result in more efficient fixed-point data types analytically determined for the problem.

These linear system solver and matrix factorization blocks now support the Tikhonov regularization parameter:

- Complex Burst QR Decomposition
- Real Burst QR Decomposition
- Complex Burst Matrix Solve Using QR Decomposition
- Real Burst Matrix Solve Using QR Decomposition
- Complex Burst Q-less QR Decomposition
- Real Burst Q-less QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition
- Real Partial-Systolic Q-less QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Complex Burst Matrix Solve Using Q-less QR Decomposition
- Real Burst Matrix Solve Using Q-less QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor
- Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

These linear system solver and matrix factorization functions now support the Tikhonov regularization parameter:

- `fixed.qlessQR`
- `fixed.qlessQRMatrixSolve`

These functions for analytically determining fixed-point data types for linear system solvers and matrix factorizations now support the Tikhonov regularization parameter:

- `fixed.qrFixedpointTypes`
- `fixed.qlessqrFixedpointTypes`
- `fixed.realQRMatrixSolveFixedpointTypes`
- `fixed.complexQRMatrixSolveFixedpointTypes`

-
- `fixed.realQlessQRMatrixSolveFixedpointTypes`
 - `fixed.complexQlessQRMatrixSolveFixedpointTypes`
 - `fixed.realSingularValueLowerBound`
 - `fixed.complexSingularValueLowerBound`

Lookup table optimization support for curve fitting objects

The `FunctionApproximation.Problem` object now supports curve fitting `cfit` (Curve Fitting Toolbox) objects as valid inputs for approximation.

```
f = fitype('expl');  
c = cfit(f,0.1,0.2);  
p = FunctionApproximation.Problem(c);
```

Lookup table optimization improved memory reduction for 1D and flat interpolation

The **Lookup Table Optimizer** has an improved algorithm for lookup table value and breakpoint optimization for one-dimensional functions with flat interpolation. This enhancement can enable improved memory reduction of the optimized lookup table and faster completion of the lookup table optimization process.

This improvement applies when the function to approximate is one-dimensional and all of these options are specified in `FunctionApproximation.Options`:

- `Interpolation` is set to `Flat`.
- `BreakpointSpecification` is set to `ExplicitValues`.
- `OnCurveTableValues` is set to `false`.

Estimate cost of generated code for Simulink models

Fixed-Point Designer now has design cost metrics you can use to estimate the cost of implementing your Simulink design in embedded C code. Analyze your model and report detailed cost data that can be traced back to the source in the Simulink model. Use a data segment metric to estimate the total size in bytes of all global variables and static local variables used in code generation. Use a program size indicator based on a weighted count of operators to conduct trade studies or track design growth following a change. For more information, see [Collect Design Cost Metrics](#).

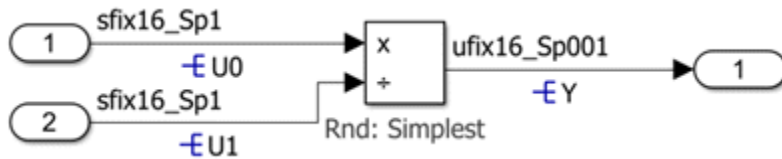
Improved numerical accuracy and generated code efficiency for fixed-point division with mixed signedness and slope and bias scaling

Fixed-Point Designer now has improved numerical accuracy for fixed-point division operations with mixed signedness in simulation and generated code. Additionally, generated code is more efficient and more readable.

This improvement applies to fixed-point division where the fixed-point slope is not a power of 2 and to non-zero bias.

Compatibility Considerations

Numerical outputs and generated code for fixed-point division operations with mixed signedness and slope and bias scaling may be more accurate than in previous releases. For example, this table illustrates the difference in generated code for this Simulink model.



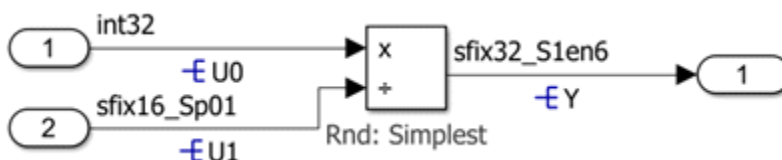
Previous Releases	R2022a
$Y = (\text{uint16_T})(((\text{uint16_T})((U0 \ll 9) / U1)$	$Y = (\text{int16_T})(U0 * 1000 / U1);$

Improved generated code fixed-point division by zero protection

Fixed-Point Designer now has improved generated code for fixed-point division operations. Fewer helper functions are generated for code that guards against division by zero, and inline code is generated instead.

This improvement applies to fixed-point division when the parameter `NoFixptDivByZeroProtection` is set to `Off`. For more information, see [Remove code that protects against division arithmetic exceptions \(Embedded Coder\)](#).

Generated code for fixed-point division operations may differ from previous releases but is numerically equivalent. For example, this table illustrates the difference in generated code for this Simulink model.



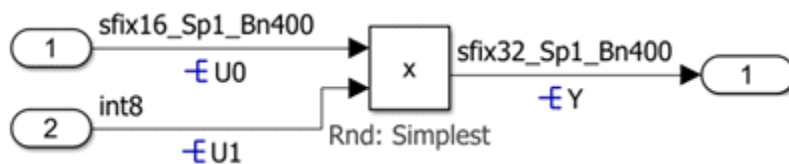
Previous Releases	R2022a
$Y = \text{div_s32s64}(U0 * 100000000LL, U1);$	<pre> if (U1 == 0LL) { if (U0 * 100000000LL >= 0LL) { Y = MAX_int32_T; } else { Y = MIN_int32_T; } } else { Y = (int32_T)(U0 * 100000000LL / U1); } </pre>

Improved numerical accuracy and generated code efficiency for fixed-point operations that do not lose precision

Fixed-Point Designer now has improved numerical accuracy for fixed-point operations that can be computed without loss of precision in simulation and generated code. Additionally, generated code is more efficient and more readable.

Compatibility Considerations

Numerical outputs and generated code for fixed-point operations that do not lose precision may be more accurate than in previous releases. For example, this table illustrates the difference in generated code for this Simulink model.



Previous Releases	R2022a
$Y = \text{mul_s32_loSR}(\text{mul_s32_loSR}(U0 * 13107 + U1, 14U), 5, 2U) + 4000;$	$Y = 2478800 * U0 + 4000 * U1 + 4000;$

Implicit expansion for logical operators, bitwise operations, and division, automatically expand dimensions of length 1

These functions now support implicit expansion with `fi` inputs.

Relational operators:

- `eq`
- `ge`
- `gt`
- `le`
- `lt`
- `ne`

Logical operators:

- `and`
- `or`
- `xor`

Division:

- `divide`
- `ldivide(.\)`

- `rdivide(./)`

Implicit expansion is a generalization of scalar expansion. With scalar expansion, a scalar expands to the same size as another array to facilitate element-wise operations. With implicit expansion, the functions listed here can implicitly expand their inputs to the same size as long as the arrays have compatible sizes. Two arrays have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same, or one of them is 1. See [Compatible Array Sizes for Basic Operations](#).

Improved accuracy in comparing `fi` objects and floating-point numbers using relational operators

Fixed-Point Designer now has improved accuracy when comparing `fi` objects to floating-point numbers using relational operators.

In previous releases, when comparing a single or double to a `fi` object, the floating-point value was cast to the same word length and signedness of the `fi` object. This process could lead to incorrect results. For example:

```
fi(0,0,8) > [-1,10]
```

```
ans =
```

```
    1x2 logical array
```

```
    0    0
```

```
fi(65534)
```

```
fi(65534.25) == 65534.25
```

```
ans =
```

```
    65534
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Signed  
    WordLength: 16  
    FractionLength: -1
```

```
ans =
```

```
    logical
```

```
    1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]
```

```
ans =
```

```
    1x2 logical array
```

```
    1    0
```

```
fi(65534.25) == 65534.25
```

```
ans =  
    logical  
    0
```

Note that the updated algorithm can produce subtle, but accurate, results. For example:

```
fi(pi) == pi  
ans =  
    logical  
    0
```

Compatibility Considerations

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest word length growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

GPU code generation support for half-precision data types in MATLAB Function blocks

GPU code generation with GPU Coder™ is now supported for half-precision data types in MATLAB Function blocks.

New functions and syntax supported for half-precision inputs

These functions and syntaxes now support half-precision inputs:

- `flintmax`
- `realmax`
- `realmin`
- `eps('half')`
- `eps('like',half(1))`

For more information, see `half` and Half Precision Code Generation Support.

`fi` `bitset` now supports scalar expansion

Prior to R2022a, `fi` `bitset` required that the second and third input arguments be the same size, or an error would occur.

```
A = fi(pi);  
disp(bin(A))  
  
bit = fi([15,3,8,2]);  
C = bitset(A,bit,1);  
disp(bin(C))
```

```
0110010010001000
```

The Second and third arguments to BITSET must be the same size.

Starting in R2022a, the input arguments *A*, *bit*, and *V* support scalar expansion. That is, if any of *A*, *bit*, or *V* are nonscalar, the other inputs can be scalar or arrays of the same size.

```
A = fi(pi);
disp(bin(A))
```

```
bit = fi([15,3,8,2]);
C = bitset(A,bit,1);
disp(bin(C))
```

```
0110010010001000
```

```
0110010010001000  0110010010001100  0110010010001000  0110010010001010
```

New Fixed-Point Designer Examples

New examples to help you get started with Tikhonov regularization parameter in linear system solver and matrix factorization blocks:

- Implement Hardware-Efficient Complex Burst Matrix Solve Using QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Complex Burst Matrix Solve Using Q-less QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Complex Partial-Systolic Matrix Solve Using QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Real Burst Matrix Solve Using QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Real Burst Matrix Solve Using Q-less QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Real Partial-Systolic Matrix Solve Using QR Decomposition with Tikhonov Regularization
- Implement Hardware-Efficient Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Tikhonov Regularization

New examples to help you get started with analytically determining data types for linear system solver blocks with the Tikhonov regularization parameter:

- Determine Fixed-Point Types for Complex Least-Squares Matrix Solve with Tikhonov Regularization
- Determine Fixed-Point Types for Complex Q-less QR Matrix Solve with Tikhonov Regularization
- Determine Fixed-Point Types for Real Least-Squares Matrix Solve with Tikhonov Regularization
- Determine Fixed-Point Types for Real Q-less QR Matrix Solve with Tikhonov Regularization

New examples to help you progress with CORDIC algorithms in Simulink:

- Hardware-Efficient Euler Rotations Using CORDIC

R2021b

Version: 7.3

New Features

Bug Fixes

Compatibility Considerations

Generate native half-precision C code for embedded hardware targets

You can now generate native half-precision C code for embedded hardware targets that natively support half precision floating-point data types, for example, `_Float16` and `_fp16` data types for ARM® compilers. For more information, see [Generate Native Half-Precision C Code from Simulink Models](#) and [Generate Native Half-Precision C Code Using MATLAB Coder](#).

Rapid Accelerator mode support for half-precision floating-point data types in Simulink

You can now use Rapid Accelerator mode in addition to Normal and Accelerator modes with the half-precision floating-point data types in Simulink, including MATLAB Function blocks that use half-precision data types. For more information about features that support half precision, see [The Half-Precision Data Type in Simulink](#).

Generate an optimized lookup table approximation as a MATLAB function

You can now use the `FunctionApproximation.Problem` object to generate an optimized lookup table approximation as a MATLAB function. To generate MATLAB function, in a `FunctionApproximation.Options` object, set the `ApproximateSolutionType` property to `MATLAB`.

The generated MATLAB function is editable and supports C/C++ code generation using MATLAB Coder™.

Improved numerical accuracy and generated code efficiency for fixed-point multiplication with slope and bias scaling

Fixed-Point Designer now has improved numerical accuracy for fixed-point multiplication operations in simulation and generated code. Additionally, generated code is more efficient and more readable.

This improvement is only applied when the configuration parameter `Use division for fixed-point net slope computation` is set to `On`. This improvement applies to fixed-point multiplication operations where the fixed-point slope is not a power of 2 and for non-zero bias when simplicity and accuracy conditions are met.

Compatibility Considerations

Numerical outputs and generated code for fixed-point multiplication operations may differ from previous releases when the **Use division for fixed-point net slope computation** parameter is set to `On`.

Improved numerical accuracy and generated code efficiency for fixed-point division with slope and bias scaling

Fixed-Point Designer now has improved numerical accuracy for fixed-point division in simulation and generated code. Additionally, generated code is more efficient and more readable.

This improvement applies to fixed-point division operations where the fixed-point slope is not a power of 2 and for non-zero bias when simplicity and accuracy conditions are met.

Compatibility Considerations

Numerical outputs and generated code may differ from previous releases for fixed-point division operations where the fixed-point slope is not a power of 2 and for non-zero bias.

Implicit Expansion: For `fi plus`, `minus`, and `times`, automatically expand dimensions of length 1

The functions `plus`, `minus`, and `times` now support implicit expansion with `fi` inputs.

Implicit expansion is a generalization of scalar expansion. With scalar expansion, a scalar expands to the same size as another array to facilitate element-wise operations. With implicit expansion, the functions listed above can implicitly expand their inputs to the same size as long as the arrays have compatible sizes. Two arrays have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1. See [Compatible Array Sizes for Basic Operations](#).

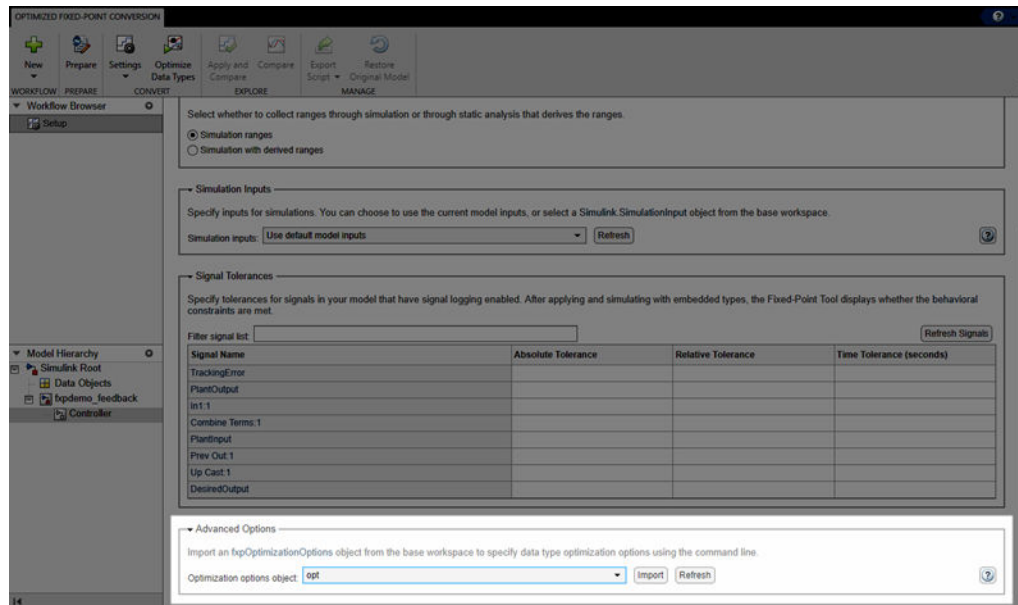
Fixed-Point Tool: Pause and resume data type optimization search, import `fxpOptimizationOptions` object, and guided workflow selection

Pause and resume data type optimization search

The Optimized Fixed-Point Conversion workflow in the **Fixed-Point Tool** now allows you to pause the optimization solver before the optimization search is complete. Any solutions found before the optimization process pauses are immediately available for use. You can resume the optimization search from where you left off, using the same or expanded search criteria.

Fixed-Point Tool: Import `fxpOptimizationOptions` object

The Optimized Fixed-Point Conversion workflow in the **Fixed-Point Tool** now has **Advanced Options** in the setup pane that allow you to import a `fxpOptimizationOptions` object from the base workspace. Importing a `fxpOptimizationOptions` object allows you access to all available optimization options and allows you to save and reuse a set of options.



Fixed-Point Tool provides guided workflow selection

On start-up, the **Fixed-Point Tool** now provides additional information to help you select the best workflow for your application.

Data Type Optimization: Restrict instrumentation to a subsystem, enforce known data types, maintain model parameter settings, and warn about unsupported constructs

Restrict instrumentation to a subsystem

During the range collection step of optimization, `fxpopt` instruments the model in order to log minimum, maximum, and overflow data during simulation. If your application does not require instrumenting of the full model, restricting instrumentation to a subsystem can reduce the time required to run the optimization solver.

To restrict instrumentation to a subsystem, use the `InstrumentationContext` property of the `fxpOptimizationOptions` object to specify the subsystem to use for instrumentation and range collection.

```
options.AdvancedOptions.InstrumentationContext = [model '/Subsystem'];
```

The subsystem must be under the top-level model and contain the system under design.

Enforce known data types for variables in a system

You can now use the `addSpecification` function to specify known data types for variables within your system in addition to being able to specify known data types for block parameters. For example, specify the data type for the parameter `myParam` using a `Simulink.Simulation.Variable` object, then add this specification to the `fxpOptimizationOptions` object, `opt`.

```
myParam = Simulink.Parameter(2); % a parameter used in the model
myParamCopy = copy(myParam); % make a copy of the parameter
```

```
myParamCopy.DataType = 'single'; % set the data type to a new value
var = Simulink.Simulation.Variable('myParam',myParamCopy); % create a variable
options = fxpOptimizationOptions();
addSpecification(options,'Variable',var);
```

Maintain original values of model parameters that are altered by fxpopt

During the optimization process, `fxpopt` changes several model configuration parameters, as described in [Model Configuration Changes Made During Data Type Optimization](#). Use the `KeepOriginalModelParameters` option of `explore` to maintain the original values of model parameters.

```
explore(result,'KeepOriginalModelParameters',true)
```

Warn about unsupported constructs

You can now choose to display a warning message when `fxpopt` encounters blocks that are not supported for data type conversion, in addition to the existing options to isolate or error. To warn for unsupported constructs, set the `HandleUnsupported` property of the `fxpOptimizationOptions` object to `'Warn'`.

```
options.AdvancedOptions.HandleUnsupported = 'Warn';
```

`fxpopt` will warn when the system contains blocks that are not supported for fixed-point conversion. Unsupported constructs are ignored and data type optimization continues. This option allows you to replace unsupported constructs with other solutions, such as lookup tables, after optimization is complete.

Analytically determine fixed-point data types when solving linear systems of equations

Fixed-Point Designer has a new set of functions to help you select fixed-point data types for linear least-squares matrix factorizations and linear system solvers.

These functions are designed to be used with blocks in the **Fixed-Point Designer HDL Support > Linear System Solvers** and **Fixed-Point Designer HDL Support > Matrix Factorizations** libraries.

To compute data types for matrix factorizations, use:

- `fixed.qrFixedpointTypes`
- `fixed.qlessqrFixedpointTypes`

To compute data types for linear system solvers, use:

- `fixed.realQRMatrixSolveFixedpointTypes`
- `fixed.complexQRMatrixSolveFixedpointTypes`
- `fixed.realQlessQRMatrixSolveFixedpointTypes`
- `fixed.complexQlessQRMatrixSolveFixedpointTypes`
- `fixed.realSingularValueLowerBound`
- `fixed.complexSingularValueLowerBound`

To compute the forgetting factor, use:

- `fixed.forgettingFactor`
- `fixed.forgettingFactorInverse`

To compute the quantization noise standard deviation, use:

- `fixed.realQuantizationNoiseStandardDeviation`
- `fixed.complexQuantizationNoiseStandardDeviation`

fixed.cordicDivide and fixed.cordicReciprocal Functions: Fixed-point divide and reciprocal using CORDIC

Use `fixed.cordicDivide` to perform division using a CORDIC algorithm. Use `fixed.cordicReciprocal` to compute the reciprocal using a CORDIC algorithm. The outputs of these functions are numerically equivalent to the output of the blocks Real Divide HDL Optimized and Complex Divide HDL Optimized, and Real Reciprocal HDL Optimized, respectively.

These functions support C/C++ code generation using MATLAB Coder.

New functions supported for half-precision inputs

The following functions now have improved support for half-precision inputs:

- `cumsum` — Supported for simulation in MATLAB
- `issorted` — Supported for simulation in MATLAB
- `log2` — Two-output syntax supported for simulation in MATLAB
- `sort` — Supported for simulation in MATLAB

For more information, see `half` and Half Precision Code Generation Support.

fi support for dec2base, dec2bin, and dec2hex

The functions `dec2base`, `dec2bin`, and `dec2hex` now support `fi` inputs. Use these functions to convert the real-world value of a `fi` object to base-*n*, binary, or hexadecimal representation. For more information, see `dec2base`, `dec2bin`, and `dec2hex`.

Data Type Optimization: Specify multiple types of tolerances

You can now specify multiple types of tolerances using the `addTolerance` function.

```
addTolerance(options, 'model/blockPath', 1, 'AbsTol', 5e-2, 'RelTol', 1e-2);
```

Compatibility Considerations

In previous releases, you specified options for logging information with a `Simulink.SimulationData.LoggingInfo` object as:

```
addTolerance(options, blockPath, portIndex, tolType, tolValue, logInfo)
```

You must now specify logging information as a name-value argument:

`addTolerance(options,blockPath,portIndex,tolType,tolValue,'LoggingInfo',logInfo)`

New Fixed-Point Designer Examples

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Linear System Solvers** library:

- Implement Hardware-Efficient Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor
- Implement Hardware-Efficient Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Matrix Factorizations** library:

- Implement Hardware-Efficient Complex Partial-Systolic Q-less QR with Forgetting Factor
- Implement Hardware-Efficient Real Partial-Systolic Q-less QR with Forgetting Factor

New examples to help you get started with analytically determining fixed-point data types when solving linear systems of equations:

- Determine Fixed-Point Types for QR Decomposition
- Determine Fixed-Point Types for Q-less QR Decomposition
- Algorithms to Determine Fixed-Point Types for Complex Q-less QR Matrix Solve $A'AX=B$
- Determine Fixed-Point Types for Complex Q-less QR Matrix Solve $A'AX=B$
- Algorithms to Determine Fixed-Point Types for Complex Least-Squares Matrix Solve $AX=B$
- Determine Fixed-Point Types for Complex Least-Squares Matrix Solve $AX=B$
- Algorithms to Determine Fixed-Point Types for Real Q-less QR Matrix Solve $A'AX=B$
- Determine Fixed-Point Types for Real Q-less QR Matrix Solve $A'AX=B$
- Algorithms to Determine Fixed-Point Types for Real Least-Squares Matrix Solve $AX=B$
- Determine Fixed-Point Types for Real Least-Squares Matrix Solve $AX=B$
- Compute Forgetting Factor Required for Streaming Input Data
- Estimate Standard Deviation of Quantization Noise of Complex-Valued Signal
- Estimate Standard Deviation of Quantization Noise of Real-Valued Signal

New examples to help you progress with lookup table optimization:

- Generate an Optimized Lookup Table as a MATLAB Function
- Generate an Optimized Lookup Table as a MATLAB Function Programmatically
- Optimize Lookup Tables for Periodic Functions

Functionality being removed or changed

Change in default behavior of `quantizenumeric` for complex input

Behavior change

In previous releases, `quantizenumeric` would remove the imaginary part of a complex input x . For example,

```
x = complex(pi, exp(1))
y = quantiznumeric(x,1,16,12, 'floor')
```

```
x =
```

```
3.1416 + 2.7183i
```

```
y =
```

```
3.1414
```

`quantiznumeric` now preserves the imaginary part in the same way that other quantize functions behave for complex inputs. For example,

```
x = complex(pi, exp(1))
y = quantiznumeric(x,1,16,12, 'floor')
```

```
x =
```

```
3.1416 + 2.7183i
```

```
y =
```

```
3.1414 + 2.7183i
```

Change in rounding behavior for quantize function

Behavior change

In previous releases, `quantize` would round to infinity for values in the range $\text{realmax} < \text{input} < \text{realmax} + 0.5 \cdot \text{eps}(\text{realmax})$ and negative infinity for values in the range $-\text{realmax} > x > -\text{realmax} - 0.5 \cdot \text{eps}$. Starting in R2021b, values in these ranges quantize as follows, depending on the rounding method used.

Rounding Method	Values in the range $\text{realmax} < \text{input} < \text{realmax} + 0.5 \cdot \text{eps}(\text{realmax})$ round to	Values in the range $-\text{realmax} > x > -\text{realmax} - 0.5 \cdot \text{eps}$ round to
floor	realmax (for $x < \text{realmax} + \text{eps}$)	$-\text{Inf}$
ceil	Inf	$-\text{realmax}$ (for $x > -\text{realmax} - \text{eps}$)
round	realmax	$-\text{realmax}$
convergent	realmax	$-\text{realmax}$
fix	realmax (for $x < \text{realmax} + \text{eps}$)	$-\text{realmax}$ (for $x > -\text{realmax} - \text{eps}$)
nearest	realmax	$-\text{realmax}$

R2021a

Version: 7.2

New Features

Bug Fixes

Compatibility Considerations

Half-precision data type support for MATLAB Function blocks

MATLAB Function blocks can now use 16-bit half-precision floating-point type data. For more information, see Floating-Point Numbers.

New HDL-optimized Simulink blocks for reciprocal, divide, and modulo

Starting in R2021a, Fixed-Point Designer has additional Simulink blocks for performing reciprocal, division and modulo operations:

- Complex Divide HDL Optimized
- Real Divide HDL Optimized
- Real Reciprocal HDL Optimized
- Divide by Constant HDL Optimized
- Modulo by Constant HDL Optimized

These blocks use hardware-friendly control signals and provide an efficient hardware implementation. These blocks support HDL code generation using HDL Coder.

New Simulink blocks and MATLAB functions for divide and modulo

Starting in R2021a, Fixed-Point Designer has additional Simulink blocks:

- Divide by Constant and Round
- Modulo by Constant

and MATLAB functions:

- `ceilDiv`
- `fixDiv`
- `floorDiv`
- `nearestDiv`
- `modByConstant`

for performing division and modulo operations.

These blocks and functions compute division via a multiplication by inverse, which generally results in better performance on embedded systems.

Improved numerical accuracy and generated code efficiency for cast operations

Fixed-Point Designer now has improved numerical accuracy for cast operations in simulation and generated code. Additionally, generated code is more efficient and more readable.

This improvement is only applied when the configuration parameter Use division for fixed-point net slope computation is set to **On**.

Generate optimized one-dimensional lookup tables for HDL applications

In R2021a, you can use lookup table optimization to generate a subsystem consisting of a prelookup step followed by interpolation that functions as a lookup table with explicit pipelining to generate efficient HDL code. To generate an HDL-optimized lookup table, use the `FunctionApproximation.Options` class:

```
problem.Options.HDLOptimized = true;
```

To generate an HDL-optimized approximate, the function to approximate must be one-dimensional and `BreakpointSpecification` must be set to `EvenSpacing` or `EvenPow2Spacing`.

New Fixed-Point Designer Examples

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Math Operations** library:

- Implement Hardware-Efficient Real Divide HDL Optimized
- Implement Hardware-Efficient Complex Divide HDL Optimized
- Implement HDL Optimized Modulo By Constant

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Linear System Solvers** library:

- Implement Hardware-Efficient Real Burst Matrix Solve Using QR Decomposition
- Implement Hardware-Efficient Real Burst Matrix Solve Using Q-less QR Decomposition
- Implement Hardware-Efficient Complex Burst Matrix Solve Using QR Decomposition
- Implement Hardware-Efficient Complex Burst Matrix Solve Using Q-less QR Decomposition
- Implement Hardware-Efficient Real Partial-Systolic Matrix Solve Using QR Decomposition
- Implement Hardware-Efficient Real Partial-Systolic Matrix Solve Using QR Decomposition with Diagonal Loading
- Implement Hardware-Efficient Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Implement Hardware-Efficient Complex Partial-Systolic Matrix Solve Using QR Decomposition
- Implement Hardware-Efficient Complex Partial-Systolic Matrix Solve Using QR Decomposition with Diagonal Loading
- Implement Hardware-Efficient Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition

New examples to help you get started with blocks in the **Fixed-Point Designer HDL Support > Matrix Factorizations** library:

- Implement Hardware-Efficient Real Burst QR Decomposition
- Implement Hardware-Efficient Real Burst Q-less QR Decomposition
- Implement Hardware-Efficient Complex Burst QR Decomposition
- Implement Hardware-Efficient Complex Burst Q-less QR Decomposition
- Implement Hardware-Efficient Real Partial-Systolic QR Decomposition

- Implement Hardware-Efficient Real Partial-Systolic Q-less QR Decomposition
- Implement Hardware-Efficient Complex Partial-Systolic QR Decomposition
- Implement Hardware-Efficient Complex Partial-Systolic Q-less QR Decomposition

New examples to help you progress with data type optimization:

- Perform Data Type Optimization with Custom Behavioral Constraints

New examples to help you progress with the **Fixed-Point Tool**:

- Use Custom Data Type Override Settings for Range Collection

Reduced HDL resource utilization in fixed-point matrix library blocks

In R2021a, blocks in the **Fixed-Point Designer HDL Optimized > Matrices and Linear Algebra** library that operate on complex inputs have improved algorithms to reduce resource utilization on hardware-constrained target platforms.

fixed.extractNumericType function: Extract numeric type of input

Use the `fixed.extractNumericType` function to extract the numeric type from an input numeric value, or from an input that specifies a numeric type. The numeric type is returned as an `embedded.numericType` object. For more information, see `fixed.extractNumericType`.

Generate C++ code for half-precision floating-point data types in Simulink

You can now generate C++ code for the half-precision floating-point data type in Simulink. For more information about features that support half-precision, see [The Half-Precision Data Type in Simulink](#).

Control inherited block output data type for half-precision

You can now control the output data type of blocks that support the half-precision data type and that support the 'Inherit via Internal Rule' block output data type. To set the data type rule, go to **Configuration Parameters > Math and Data Types > Inherit floating-point output type smaller than single precision**. For more information, see [Inherit floating-point output type smaller than single precision](#).

Fixed-Point Tool: View optimization details, visualize data types, and manually stop optimization

View optimization details in the Fixed-Point Tool

Prior to R2021a, details of the optimization process were printed to the MATLAB Command Window. You can now view this information in the **Optimized Fixed-Point Conversion** workflow of the Fixed-Point Tool. In the **Workflow Browser** pane, during data type optimization or after the optimization process has terminated, select **Optimization Details**.

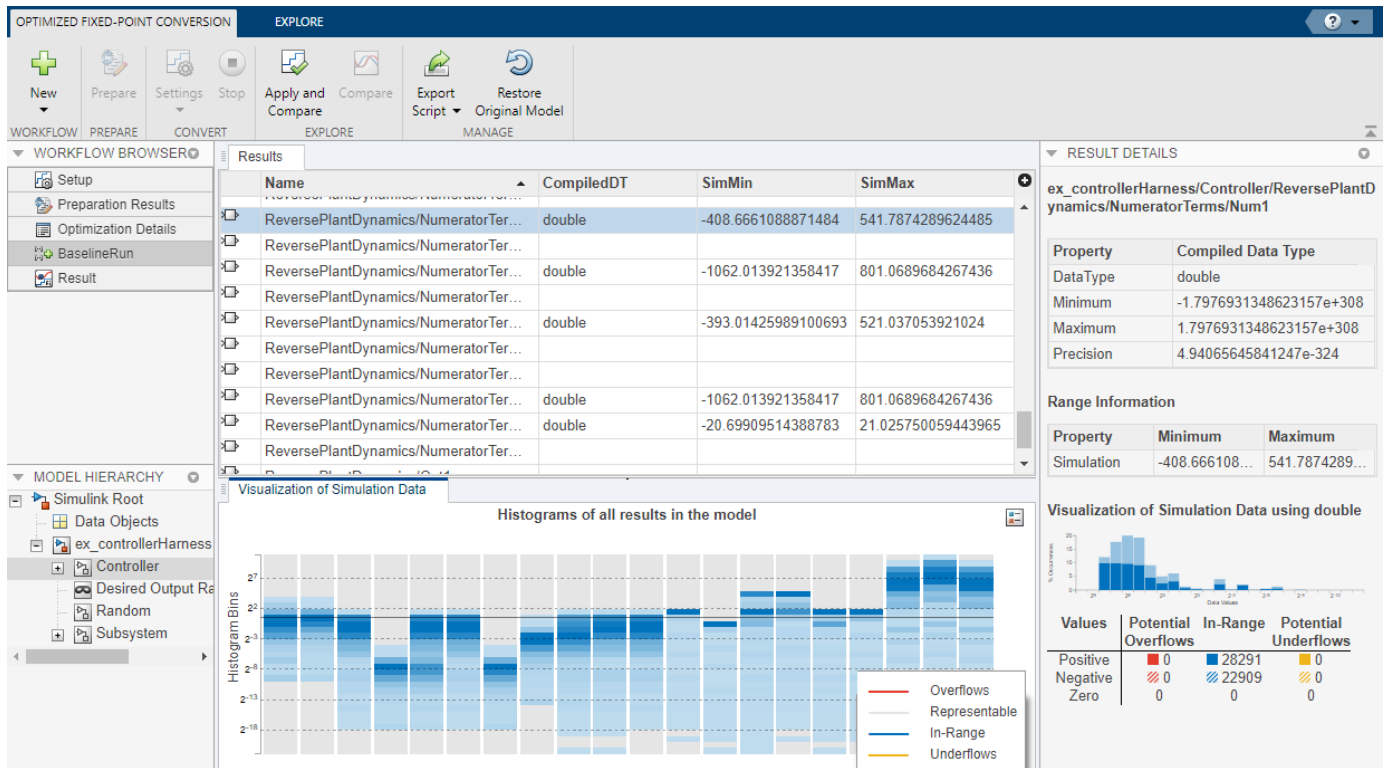
Starting the optimization process... Select Stop to end.

- + Preprocessing
- + Modeling the optimization problem
 - Constructing decision variables
- + Running the optimization solver
 - Evaluating new solution: cost 496, does not meet the behavioral constraints.
 - Evaluating new solution: cost 976, does not meet the behavioral constraints.
 - Evaluating new solution: cost 1936, meets the behavioral constraints.
 - Updated best found solution, cost: 1936
 - Evaluating new solution: cost 1920, meets the behavioral constraints.
 - Updated best found solution, cost: 1920
 - Evaluating new solution: cost 1840, meets the behavioral constraints.
 - Updated best found solution, cost: 1840
 - Evaluating new solution: cost 1808, meets the behavioral constraints.
 - Updated best found solution, cost: 1808
 - Evaluating new solution: cost 1792, does not meet the behavioral constraints.

Data type visualizer: Understand and analyze optimized data types by viewing histograms of the dynamic ranges of signals in your model

In the Optimized Fixed-Point Conversion workflow in the Fixed-Point Tool, you can now view a summary of the ranges of objects in your model and histograms of the bits used by each object. This data is collected during the range collection phase of optimization.

Use this data type visualization to see a summary of the ranges of objects in your model and to quickly spot sources of overflows and underflows and inefficient data types.



Stop data type optimization

The Optimized Fixed-Point Conversion workflow in the Fixed-Point Tool now allows you to stop the optimization solver before the optimization search is complete. Any solutions found before the optimization process terminates remain available for use.

Lookup table optimization support for functions with scalar inputs

Previously, the `FunctionApproximation.Problem` class required that functions and function handles to approximate were vectorized, meaning that for each input, there is exactly one output. Starting in R2021a, lookup table optimization fully supports approximation of Simulink blocks and subsystems that only allow scalar inputs.

Improved lookup table value optimization

In R2021a, the **Lookup Table Optimizer** has an improved algorithm for lookup table value optimization for the `Flat` and `Nearest` interpolation methods when off-curve table values are allowed. This enhancement can enable faster completion of the lookup table optimization process and improved memory reduction of the optimized lookup table.

Improved numerical accuracy and generated code efficiency for fixed-point inputs to power, \cdot^{\wedge}

Fixed-Point Designer now has improved numerical accuracy for fixed-point inputs to power, \cdot^{\wedge} in simulation and generated code. Additionally, generated code is more efficient.

Data type optimization workflow improvements

Override data types with scaled doubles

Using the `fxpOptimizationOptions` object, you can now override data types in a model with scaled doubles.

```
options.AdvancedOptions.DataTypeOverride = 'ScaledDouble';
```

Log a reduced set of data points

Using the `addTolerance` method of the `fxpOptimizationOptions` object, you can now control the amount of data logged by the Simulation Data Inspector by specifying a decimation factor.

```
logInfo = Simulink.SimulationData.LoggingInfo();  
logInfo.DecimateData = true;  
logInfo.Decimation = 10;  
addTolerance(options, 'model/blockPath', 2, 'AbsTol', 1, logInfo);
```

Stop optimization in Lookup Table Optimizer app

You can now stop the optimization solver in the **Lookup Table Optimizer** before the optimization search is complete. The app will choose the best solution found at the time the **Stop** button is selected and display it in the app.

New Fixed-Point Designer Simulink block library

The Fixed-Point Designer Simulink block library has been split. Blocks in the **Fixed-Point Designer HDL Support** library use hardware-friendly control signals and provide an efficient hardware implementation. These blocks support HDL code generation using HDL Coder. Blocks in the **Fixed-Point Designer** library are not optimized for HDL applications.

Functionality being removed or changed

Inexact property names for `fi`, `fimath`, and `numericType` objects not supported

In previous releases, inexact property names for `fi`, `fimath`, and `numericType` objects would result in a warning. In R2021a, support for inexact property names was removed. Use exact property names instead.

R2020b

Version: 7.1

New Features

Bug Fixes

Compatibility Considerations

Half Precision in Simulink: Design, simulate, and generate code for half-precision systems

Signals and block outputs can now specify a half-precision data type. The half-precision data type is supported for simulation and code generation for parameters and a subset of blocks, including Lookup Table blocks.

For more information about half-precision in Simulink, see [The Half-Precision Data Type in Simulink](#).

Expanded half-precision support for Deep Learning Toolbox and FFT functions

The following MATLAB functions now support half-precision inputs.

- `fft`
- `fft2`
- `fftn`
- `fftshift`
- `ifft`
- `ifft2`
- `ifftn`
- `ifftshift`
- `permute`

The following Deep Learning Toolbox™ functions now support half-precision inputs.

- `activations`
- `classify`
- `predict`
- `predictAndUpdateState`

For more information on the half-precision data type in MATLAB, see `half`.

Explore half precision in optimized lookup tables

The new `ExploreHalf` property of the `FunctionApproximation.Options` object allows you to specify whether the optimization process explores half-precision data types for table data and breakpoint values. The default property value of this property is `true`. If you specify `false`, the optimizer does not explore half precision.

```
options = FunctionApproximation.Options('ExploreHalf',true);
```

You can also access this property in the **Lookup Table Optimizer**.

New API functions for half-precision data type support in user-written S-functions

To create S-function blocks that work with half-precision data types, use these new functions:

-
- `ssGetDataTypeIsDoubleSingleorHalf` — Determine whether registered data type is double, single, or half-precision data type.
 - `ssGetDataTypeIsHalfPrecision` — Determine whether registered data type is half-precision data type.
 - `ssRegisterDataTypeHalfPrecision` — Register half-precision data type and return its data type ID.

New QR decomposition and matrix solve Simulink blocks

Fixed-Point Designer now has additional Simulink blocks for matrix operations. These blocks use a partial-systolic implementation and have hardware-friendly control signals that provide an efficient hardware implementation. These blocks support HDL code generation using HDL Coder.

Linear System Solvers

- Real Partial-Systolic Matrix Solve Using QR Decomposition
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor
- Complex Partial-Systolic Matrix Solve Using QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Matrix Factorization

- Real Partial-Systolic QR Decomposition
- Real Partial-Systolic Q-less QR Decomposition
- Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor
- Complex Partial-Systolic QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor

New QR decomposition and matrix solve MATLAB functions

Fixed-Point Designer now has additional functions for matrix QR factorization and solving linear systems of equations. To learn more about these functions, see the following links.

- `fixed.backwardSubstitute`
- `fixed.forwardSubstitute`
- `fixed.qlessQR`
- `fixed.qlessQRUpdate`
- `fixed.qrAB`
- `fixed.qrMatrixSolve`
- `fixed.qlessQRMatrixSolve`

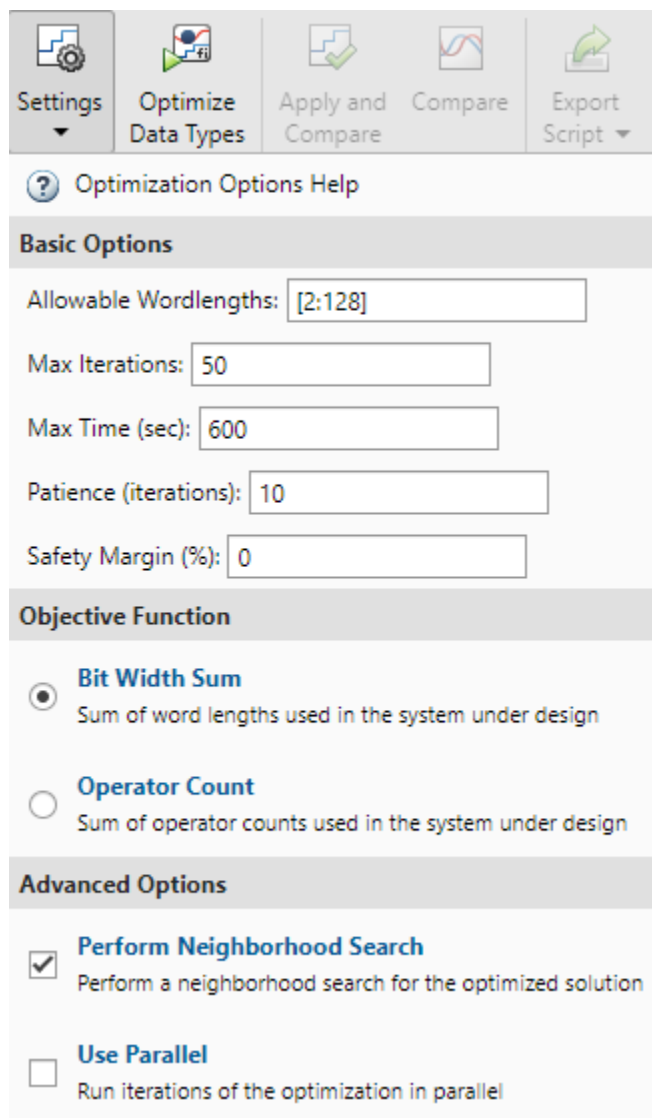
Optimize data types based on operator counts

The `fxpopt` function now supports data type optimization that minimizes an estimated count of operators in generated code. This results in a lower program memory size for C code generated from Simulink models.

By default, `fxpopt` minimizes the total bit width. To minimize operator counts during optimization, set the `'ObjectiveFunction'` property of the `fxpOptimizationOptions` object to `'OperatorCount'`.

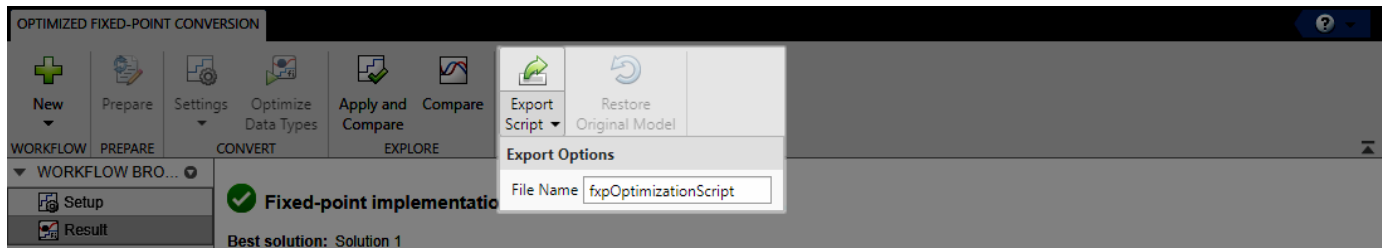
```
opt = fxpOptimizationOptions('ObjectiveFunction','OperatorCount');
```

In the Fixed-Point Tool Optimized Fixed-Point Conversion workflow, expand the **Settings** menu to choose which objective function to use for optimization.



Export optimization workflow steps to a MATLAB script

In the Optimized Fixed-Point Conversion workflow of the Fixed-Point Tool, you can now use the **Export Script** button to export optimization workflow steps, including optimization options and signal tolerances, to a MATLAB script.



Automatically propagate slope-bias data types during data type optimization

The `fxpopt` function can now propagate slope-bias data types from outside the system under design. The optimizer chooses slopes and biases that reduce the complexity of generated code.

To allow slope-bias data type propagation during optimization, set the `PerformSlopeBiasCancellation` property of the `fxpOptimizationOptions` object to `true`.

```
options.AdvancedOptions.PerformSlopeBiasCancellation = true;
```

Data type optimization workflow improvements

Automatically isolate constructs not supported for fixed-point conversion

You can now automatically isolate blocks that are not supported for fixed-point conversion. After optimization, you can choose to replace these constructs with a lookup table, CORDIC implementation, or other solution.

To surround unsupported constructs with Data Type Conversion blocks, set the `HandleUnsupported` property of the `fxpOptimizationOptions` object to `'Isolate'`.

```
options.AdvancedOptions.HandleUnsupported = 'Isolate';
```

Override data types in range collection step of optimization

Using the `fxpOptimizationOptions` object, you can now override data types in the model when collecting ranges used to optimize data types.

The optimization process collects ranges through simulation, static range analysis, and design ranges specified on a model and finds an optimal data type based on the union of all ranges. When you specify a data type override, the software overrides all data types in the model with singles or doubles during the range collection step of optimization. For example, to override all data types with doubles, set the `DataTypeOverride` property of the `fxpOptimizationOptions` object to `'Double'`.

```
options.AdvancedOptions.DataTypeOverride = 'Double';
```

Inspect optimization solutions using Simulation Manager

You can now inspect solutions found during optimization in Simulation Manager.

After using the `fxpopt` function to optimize the data types in your model, you can inspect the solutions found during optimization in Simulation Manager using the `openSimulationManager` function.

Functionality being removed or changed

Change in default behavior of `fi` for `-Inf`, `Inf`, and `NaN`

Behavior change

In previous releases, `fi` would return an error when passed the non-finite values `-Inf`, `Inf`, or `NaN`.

In R2020b, when `fi` is specified as a fixed-point numeric type,

- `NaN` maps to `0`.
- When the `'OverflowAction'` property of the `fi` object is set to `'Wrap'`, `-Inf` and `Inf` map to `0`.
- When the `'OverflowAction'` property of the `fi` object is set to `'Saturate'`, `Inf` maps to the largest representable value, and `-Inf` maps to the smallest representable value.

Best-precision scaling is not supported for input values of `-Inf`, `Inf`, or `NaN`.

Change in default data type override in the Fixed-Point Tool

Behavior change

In previous releases, when you clicked the **Collect Ranges** button in the **Iterative Fixed-Point Conversion** or **Range Collection** workflows in the Fixed-Point Tool, the tool would by default override data types in the model with doubles.

In R2020b, the default behavior for the **Collect Ranges** button is set to `Use current settings`, which maintains the current data type override set on the model.

R2020a

Version: 7.0

New Features

Bug Fixes

Compatibility Considerations

Half Precision: Design, simulate, and generate code for half-precision systems

Half precision code generation in MATLAB

You can now generate C, C++, and GPU code for half-precision floating-point data types in MATLAB. The half-precision data type occupies only 16 bits of memory, but its floating-point representation enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size.

For more information about features that support half-precision, see `half`.

Tech Preview: Half precision in Simulink

Beginning in R2020a, signals and block outputs can now specify a half-precision data type. The half-precision data type is supported for simulation and code generation for parameters and a subset of blocks, including Lookup Table blocks.

This feature is under tech preview and should not be used for production code generation. For more information on supported features, see Half-Precision in Simulink.

Fixed-Point Tool: Convert and optimize data types, and explore ranges

You can now select between three workflows in the Fixed-Point Tool. To get started, open the Fixed-Point Tool from the **Apps** menu in Simulink. In the Fixed-Point Tool, click the **New** button and select one of the following workflows:

- **Optimized Fixed-Point Conversion** - Optimize the data types of a system to use the minimum cost while meeting system behavior requirements.
- **Iterative Fixed-Point Conversion** - Convert a system to fixed point based on simulation, derived, or design ranges. Adjust data type proposal settings and visualize their effects until system meets your requirements.
- **Range Collection** - Debug the behavior of a model and pinpoint numerical issues by visualizing ranges.

New Fixed-Point Designer Simulink block library

Fixed-Point Designer now has a Simulink block library for math operations and matrix operations. These blocks use hardware-friendly control signals and provide an efficient hardware implementation. These blocks support HDL code generation using HDL Coder.

Math Operations

Use the Hyperbolic Tangent HDL Optimized block to compute the CORDIC-based hyperbolic tangent.

Use the Normalized Reciprocal HDL Optimized block to compute the normalized reciprocal of an input value.

Matrix Operations

The Real Burst QR Decomposition and Complex Burst QR Decomposition blocks use Givens rotations to compute the QR decomposition of an input matrix.

Use the Real Burst Matrix Solve Using QR Decomposition or Complex Burst Matrix Solve Using QR Decomposition blocks to compute the value of x in the equation $Ax = b$.

The Real Burst Q-less QR Decomposition and Complex Burst Q-less QR Decomposition blocks use Givens rotations to compute the R factor of the QR decomposition without computing Q .

Use the Real Burst Matrix Solve Using Q-less QR Decomposition or Complex Burst Matrix Solve Using Q-less QR Decomposition blocks to compute the value of x in the equation $A'Ax = b$.

Lookup Table Optimization: Iterative redesign and batch compression of lookup tables, parallelization of lookup table optimization

Iteratively redesign lookup tables in your model

The **Lookup Table Optimizer** now replaces blocks being approximated by a lookup table with a variant subsystem containing the function approximation. The variant subsystem enables you to return to the original function and perform the optimization again using different optimization settings and constraints.

Automatically compress all lookup tables in a system

Use the `FunctionApproximation.compressLookupTables` function to locate and optimize all lookup tables in a specified system. `FunctionApproximation.compressLookupTables` performs a lossless compression of all supported lookup table blocks in a specified subsystem and reports the memory savings.

Parallelized lookup table optimization

The new `UseParallel` property of the `FunctionApproximation.Options` object allows you to specify whether to run iterations of the optimization in parallel. The default property value of this property is `false`. Running iterations in parallel requires a Parallel Computing Toolbox™ license. If you do not have a Parallel Computing Toolbox license, or if you specify `false`, the iterations run in serial.

Data Type Optimization: Specify a safety margin, enforce known data types, and other enhancements

Review all changes made during optimization

Use the `contents` function to get a summary of all changes made to your model during optimization.

For example, after optimizing the model used in the Optimize Data Types Using Multiple Simulation Scenarios example, you can see a summary of model parameters and data types that are changed.

```
result = fxpopt(model,sud,opt);  
contents(result.Solutions(3))
```

```
ModelName: 'ex_auto_gain_controller'
```

ModelParameters:		
Index	Name	Value
1	SignalLogging	'on'

2	ReturnWorkspaceOutputs	'on'
3	SaveFormat	'Dataset'
4	ShowPortDataTypes	'on'
5	SignalRangeChecking	'error'
6	ParameterDowncastMsg	'none'
7	ParameterUnderflowMsg	'none'
8	ParameterPrecisionLossMsg	'none'
9	ParameterOverflowMsg	'none'
10	FixptConstPrecisionLossMsg	'none'
11	FixptConstOverflowMsg	'none'
12	FixptConstUnderflowMsg	'none'
13	IntegerOverflowMsg	'none'
14	IntegerSaturationMsg	'none'
15	SaveTime	'off'
16	SaveOutput	'off'
17	SimulationMode	'accelerator'

BlockParameters:

Index	Name	BlockPath	Value
1	OutDataTypeStr	ex_auto_gain_controller/input_signal	'fixdt(1,17,15)'
2	OutDataTypeStr	ex_auto_gain_controller/sud/x	'fixdt(1,17,15)'
3	OutDataTypeStr	ex_auto_gain_controller/sud/Product	'fixdt(1,11,6)'
4	OutDataTypeStr	ex_auto_gain_controller/sud/y	'fixdt(1,11,6)'
5	OutDataTypeStr	ex_auto_gain_controller/sud/Loop gain	'fixdt(0,17,29)'
6	OutDataTypeStr	ex_auto_gain_controller/sud/Product1	'fixdt(1,17,22)'
...			

Specify a safety margin for optimization

Using the `fxpOptimizationOptions` object, you can now specify a safety margin for the ranges used to optimize data types.

The optimization process collects ranges through simulation, static range analysis, and design ranges specified on a model and finds an optimal data type based on the union of all ranges. When you specify a safety margin, the software augments the collected ranges with the specified margin. For example, to add ten percent to all collected ranges, set the `SafetyMargin` property of the `fxpOptimizationOptions` object, to 10.

```
options.AdvancedOptions.SafetyMargin = 10;
```

Enforce known data types in a system

Use the `addSpecification` function to specify known data types for blocks and parameters within your system. For example, if you know that the input to your system will always be an `int8`, you can do the following to enforce this during optimization.

```
bp = Simulink.Simulation.BlockParameter(...
'ex_auto_gain_controller/input_signal','OutDataTypeStr','int8');
addSpecification(opt,'BlockParameter',bp)
```

Revert optimization

After using the `fxpopt` function to optimize the data types in your model, you can revert your model back to its original state using the `revert` function.

Coder Type Editor: Create and edit input types interactively

While using the `fiaccel`, `buildInstrumentedMex`, or `convertToSingle` command, you can specify the type, size, and complexity of the input arguments of your MATLAB entry-point functions by using `coder.Type` objects. In R2020a, you can create and edit `coder.Type` objects interactively by using the Coder Type Editor. To launch the Coder Type Editor, run this command at the MATLAB command line:

```
coderTypeEditor
```

See [Create and Edit Input Types by Using the Coder Type Editor and coderTypeEditor](#).

normalizedReciprocal: Compute the normalized reciprocal

The `normalizedReciprocal` function computes the normalized reciprocal of an input, which can then be used in fixed-point applications. For syntax and examples, see [normalizedReciprocal](#).

nextpow2: Compute the next-higher power of 2 of fixed-point values

The `nextpow2` function returns the exponents of the next-higher powers of 2 that satisfy $2.^p \geq \text{abs}(a)$ for fixed-point values.

```
a = fi([1 -2 3 -4 5 9 519]);  
p = nextpow2(a)
```

```
p =
```

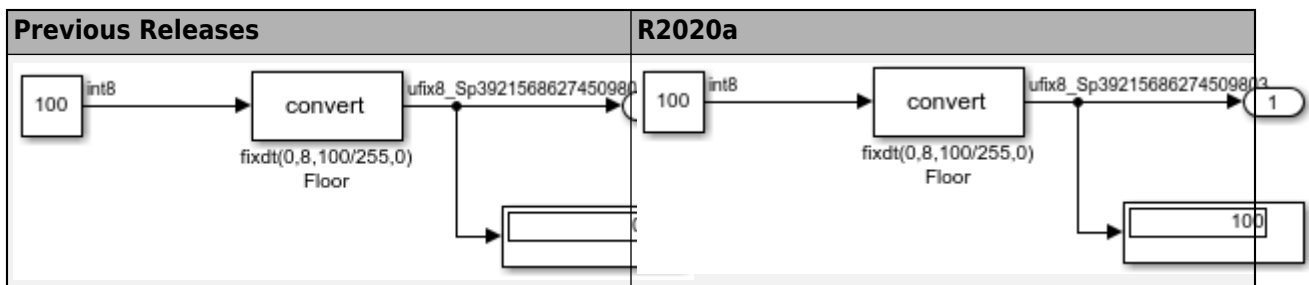
```
     0     1     2     2     3     4    10
```

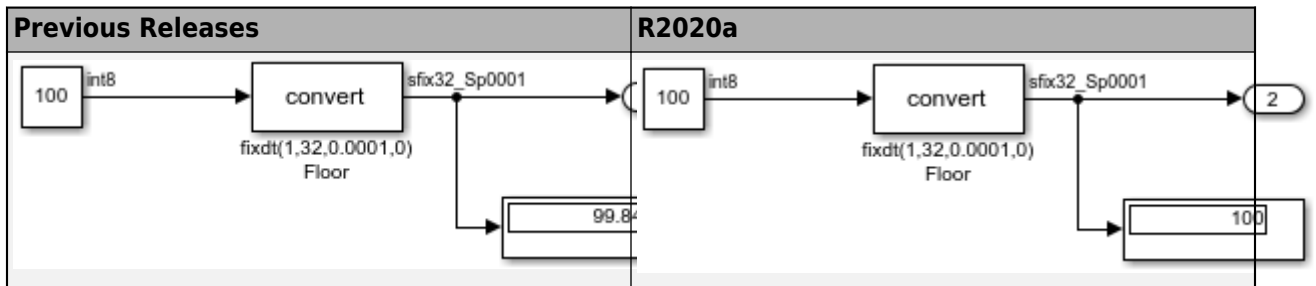
```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 6  
      FractionLength: 0
```

For more information, see [nextpow2](#).

Improved numerical accuracy for slope-bias scaled fixed-point operations

Fixed-Point Designer now has improved numerical accuracy for slope-bias scaled fixed-point operations in simulation and code generation. For example, this table illustrates the difference in numerical accuracy in simulation for these Simulink models.





Generate test data as a dataset

Use the `outputAllData` function to generate test data as a dataset. In previous releases, you could output data only as a `timeseries` object or an array.

Functionality being removed or changed

The `applyOnRootInport` function will be removed in a future release. In R2020a, this function emits a warning.

R2019b

Version: 6.4

New Features

Bug Fixes

Propose data types based on multiple simulation scenarios in the Fixed-Point Tool

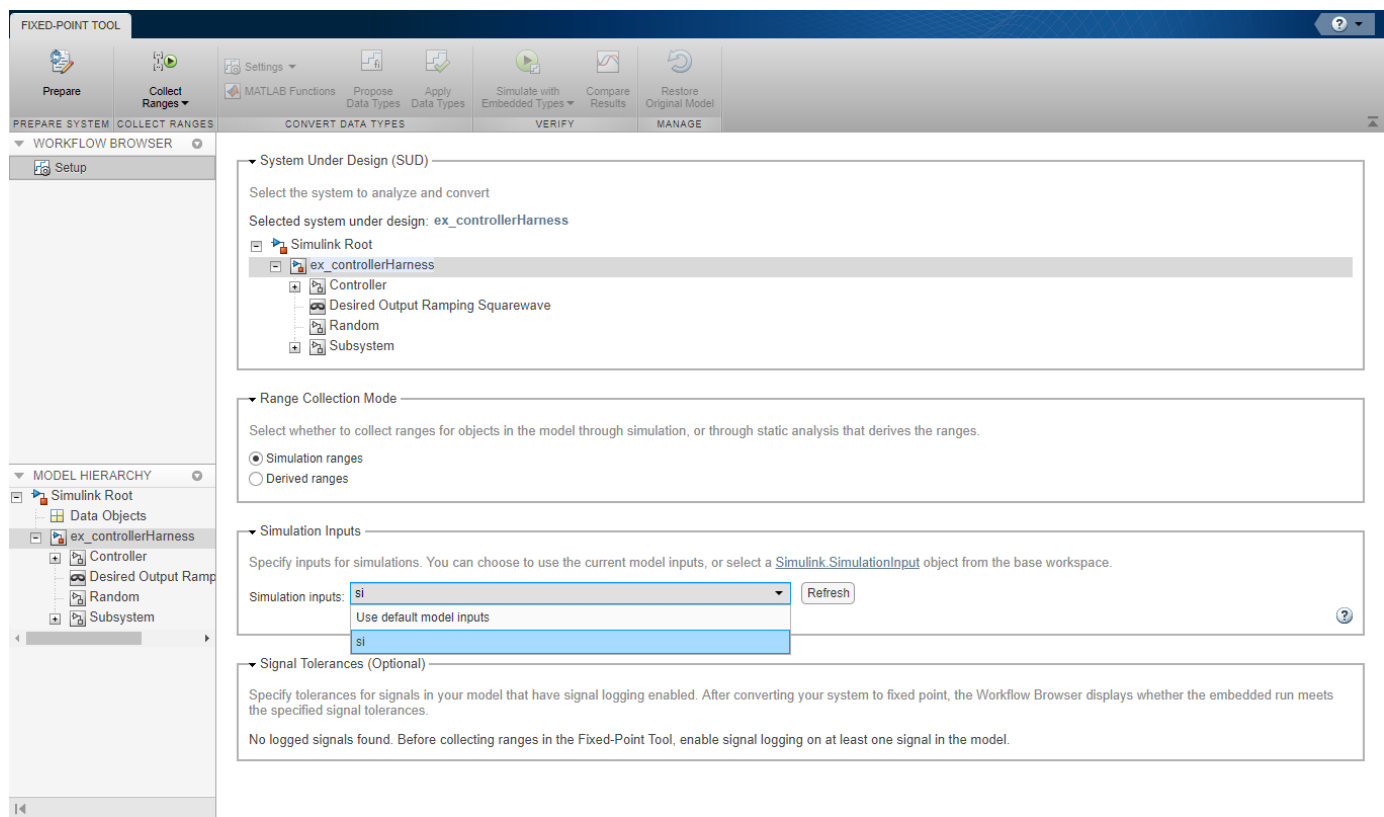
You can now use a `Simulink.SimulationInput` object to author different simulation scenarios in the Fixed-Point Tool. The Fixed-Point Tool proposes data types based on merged ranges from the simulation scenarios.

Define a `SimulationInput` object in the base workspace and specify the conditions for each scenario.

```
si = Simulink.SimulationInput
% Scan through different seeds for a random input
rng(1);
seeds = randi(1e6, [1 4]);

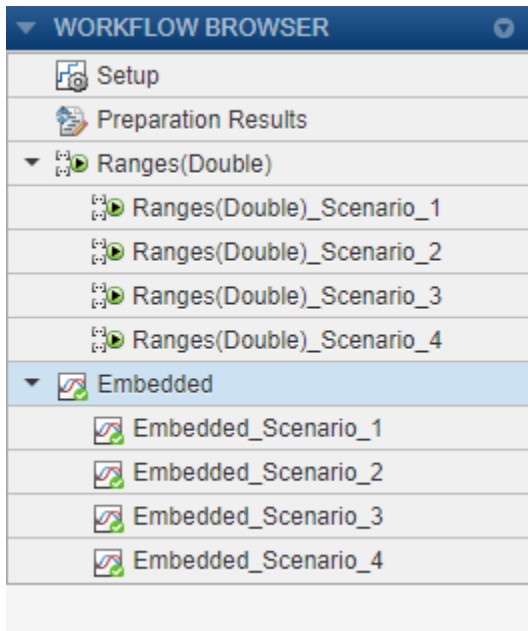
for sIndex = 1:length(seeds)
    si(sIndex) = Simulink.SimulationInput(model);
    si(sIndex) = si(sIndex).setVariable('SOURCE', 2);
    si(sIndex) = si(sIndex).setBlockParameter([model '/Random/uniformRandom'],...
        'Seed', num2str(seeds(sIndex))); % scan through the seeds
    si(sIndex) = si(sIndex).setUserString(sprintf('random_%i', seeds(sIndex)));
end
```

In the Fixed-Point Tool, select the `SimulationInput` object specified in the base workspace under **Simulation Inputs**.



The tool proposes data types based on the merged ranges collected from all simulation scenarios.

You can see the details of both the individual simulations as well as the merged simulation by selecting the run in the **Workflow Browser**.



Restore model to original design

The preparation stage of fixed-point conversion in the Fixed-Point Tool now generates a restore point for your model by saving a copy of the model in its current state. To restore your model back to its original state, in the Fixed-Point Tool, at any time during the conversion click the **Restore Original Model** button.

Quantize and generate fixed-point C/C++ code for a trained SVM model (requires MATLAB Coder and Statistics and Machine Learning Toolbox)

You can quantize a trained model and generate C/C++ code for the prediction of a support vector machine (SVM) classification and SVM regression. To generate fixed-point code, create a structure that defines fixed-point data types using the `generateLearnerDataTypeFcn` and use the structure as an input argument of the `loadLearnerForCoder` function in an entry-point function.

Allow off-curve table values in optimized lookup tables

You can now generate an optimized lookup table with off-curve table values.

In past releases, the optimization required table values to match the quantized output values of the original function being approximated. By allowing off-curve table values, you may be able to reduce the memory of the lookup table while maintaining the same error tolerances, or maintain the same memory while reducing the error tolerances.

To allow off-curve table values, in the `FunctionApproximation.Options` object, set the `OnCurveTableValues` property to `0`.

```
options = FunctionApproximation.Options;
options.OnCurveTableValues = 0;
```

Generate optimized AUTOSAR-compliant lookup table

Generate an AUTOSAR-compliant optimized lookup table using a Curve or Map block. Set the `AUTOSARCompliant` property to 1 in the `FunctionApproximation.Options` object.

```
options = FunctionApproximation.Options;
options.AUTOSARCompliant = 1;
```

You can also access this property in the **Lookup Table Optimizer**.

Setting this property to 1 (true) checks out an AUTOSAR Blockset license when you use the `approximate` or `replaceWithApproximate` methods.

Generate simulation inputs to test full operating range of design

Use the `fixed.DataGenerator` and `fixed.DataSpecification` objects to generate simulation inputs to test the full operating range of your design.

You can generate intervals of data with values focusing on cases such as values close to boundaries, values close to powers-of-two, `inf` and `NaN`, negative zero, and denormal numbers. The generated data can be of any dimension or complexity, and it can have a double, single, integer, or fixed-point data type.

You can specify the data type, interval, and other properties of the data you want to generate using the `fixed.DataSpecification` object.

```
dataspec = fixed.DataSpecification('fixdt(1,16,8)', 'Interval', {-1, 1})
```

```
dataspec =
```

```
    fixed.DataSpecification with properties:
```

```
        DataTypeStr: 'sfix16_En8'
           Interval: [-1,1]
MandatoryValues: <empty>
        Complexity: 'real'
           Dimensions: 1
```

Use the `fixed.DataGenerator` object to generate the data according to the specifications, and access the output.

```
datagen = fixed.DataGenerator('DataSpecifications', dataspec);
testData = outputAllData(datagen)
```

```
testData =
```

```
Columns 1 through 7
-1.0000 -0.9961 -0.5039 -0.5000 -0.4961 -0.2539 -0.2500
Columns 8 through 14
-0.2461 -0.1289 -0.1250 -0.1211 -0.0664 -0.0625 -0.0586
Columns 15 through 21
-0.0352 -0.0313 -0.0273 -0.0195 -0.0156 -0.0117 -0.0078
Columns 22 through 28
```

```
-0.0039      0      0.0039      0.0078      0.0117      0.0156      0.0195
Columns 29 through 35
 0.0273     0.0313     0.0352     0.0586     0.0625     0.0664     0.1211
Columns 36 through 42
 0.1250     0.1289     0.2461     0.2500     0.2539     0.4961     0.5000
Columns 43 through 45
 0.5039     0.9961     1.0000
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 8
```

Features under tech preview

Tech Preview: HDL-optimized fixed-point matrix operations blocks

The Real Burst QR Decomposition and Complex Burst QR Decomposition blocks use Givens rotations to efficiently compute the QR decomposition of an input matrix.

Use the Complex Burst Matrix Solve Using QR Decomposition or the Real Burst Matrix Solve Using QR Decomposition blocks to compute the value of x in the equation $Ax = b$.

The blocks use hardware-friendly control signals and provide an efficient hardware implementation. The block supports HDL code generation using HDL Coder.

Tech Preview: Half-precision data types in Simulink

To simulate half-precision floating-point data types in Normal mode in your Simulink model, contact rcheruku@mathworks.com.

R2019a

Version: 6.3

New Features

Bug Fixes

Emulate hardware handling of denormal numbers

If your target hardware uses flush-to-zero behavior for denormal numbers, you can now emulate this behavior during accelerated simulation of your system.

To enable flush-to-zero behavior, in the Configuration Parameters, on the **Math and Data Types** pane, set the **Simulation behavior for denormal numbers** parameter to `Flush to zero (FTZ)`. The default behavior for simulation of denormal numbers is `Gradual underflow`.

You can simulate a top-level model using gradual underflow with any simulation mode. Models referenced by the top-level model can simulate the flush-to-zero behavior only if the instance of the referenced model uses an accelerated simulation mode and has the **Simulation behavior for denormal numbers** parameter set to `Flush to zero (FTZ)`.

New data type propagation rules for Sum, Gain, and Product blocks

There are now new output data type choices for the Sum, Gain, and Product blocks. These new data type propagation rules give you more control over the range and scaling of the output.

- **Inherit: Keep MSB** - This rule selects an output data type that maintains the full range of the operation and then reduces the precision of the output value to a size appropriate for the target hardware. This rule will never produce an overflow.

This rule is available for the Sum, Product, and Gain blocks.

- **Inherit: Keep LSB** - This rule selects an output data type that maintains the precision of the operation but reduces the range if the full type does not fit on the target hardware. This rule can produce overflows.

This rule is available for the Sum block.

- **Inherit: Match Scaling** - This rule attempts to maintain the scaling of the output data type. This rule can produce overflows.

This rule is available for the Product and Gain blocks.

Automatically prepare Simulink systems for conversion to fixed point

Using the Fixed-Point Tool, you can prepare a model for conversion from a floating-point model or subsystem to an equivalent fixed-point representation. During the preparation stage of the conversion, the Fixed-Point Tool checks the system under design for compatibility with the conversion process and reports any issues found in the model. When possible, the Fixed-Point Tool automatically changes settings that are not compatible. In cases where the tool is not able to automatically change the settings, the tool notifies you of the changes you must make manually to help the conversion process be successful.

Complex support for half-precision

You can now represent complex values using a half-precision floating-point data type in MATLAB. To cast a variable to half precision, use the `half` function.

```
a_double = 3 + 4i;  
a_half = half(a_double)
```

```
a_half =  
    half  
    3.0000 + 4.0000i
```

Most functions which support half-precision inputs also support complex half-precision inputs.

Specify multiple simulation scenarios for data type optimization

You can now specify multiple simulation scenarios to use for collecting ranges and verifying your design during fixed-point optimization. Specifying multiple simulation scenarios enables you to optimize the data types of your system using a range of input stimuli to ensure that the system is exercised over its entire operating range. The optimization uses the defined simulation scenarios to verify the solutions based on the tolerances specified in the options object.

Lookup table optimization options available in the app

You can now specify the following options from the Lookup Table Optimizer app.

- **Interpolation** - Method to use when an input falls between breakpoint values
Setting the **Interpolation** to **None** generates a Direct Lookup Table (n-D) block.
- **Breakpoint specification** - Spacing of breakpoint data
- **Saturate to output type** - Whether to saturate the output of the function being approximated to the range of the output type

The app is also now able to approximate any MATLAB function handle, Math Function block, or stateless subsystem. It can also optimize the breakpoints and spacing of any existing Lookup Table block.

Specify new constraints for lookup table optimization

Using the Lookup Table Optimizer, you can now specify additional options to control the optimization behavior.

- **Max Memory Usage** - Specify the maximum amount of memory, in bytes, that the lookup table approximation can use.
- **Max Time** - Specify the maximum amount of time, in seconds, to allow the approximation to run. The approximation runs until it reaches the time specified, finds an ideal solution, or reaches another stopping criteria.

You can specify these options in the Advanced Options dialog on the **Create** page of the **Lookup Table Optimizer** app or using the `FunctionApproximation.Options` object.

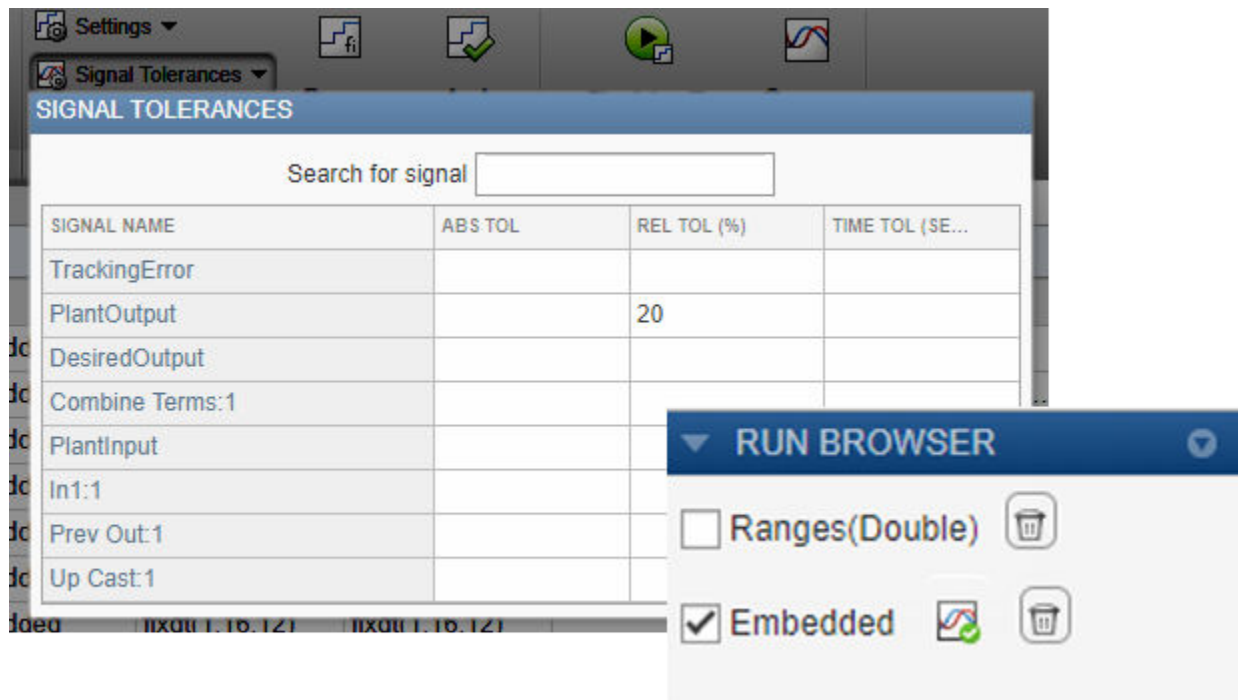
Derived range analysis support for fixed-point optimization

When using `fxpopt` to optimize the fixed-point data types of a Simulink system, you can now specify whether the optimization should consider ranges derived from design ranges specified in your model when assessing a solution. To enable derived range analysis while optimizing data types, set the `UseDerivedRangeAnalysis` property of the `fxpOptimizationOptions` object to `true`.

```
opt = fxpOptimizationOptions;
opt.AdvancedOptions.UseDerivedRangeAnalysis = true;
```

Specify tolerances of signals in system for conversion

After performing a range collection run, you can specify absolute, relative, and time tolerances for signals in your model that have signal logging enabled. After you simulate an embedded run, the Run Browser displays whether the embedded run meets the specified signal tolerances compared to the range collection run. You can view the comparison plots in the Simulation Data Inspector.



New functions supported for half-precision inputs

The following functions now support half-precision inputs.

- fma - new in R2019a
- hypot
- min
- max
- mean
- dot

In addition, the relational operators (gt, lt, eq, ge, le, ne) are now able to compare half and integer types.

For more information, see `half`.

R2018b

Version: 6.2

New Features

Bug Fixes

Lookup Table Optimization: Automatically replace subsystems with a direct lookup table and other enhancements

Approximate a Subsystem with a lookup table

You can now replace an entire subsystem with a lookup table. To approximate a subsystem, specify the subsystem you want to approximate in the `FunctionApproximation.Problem` object. This functionality is available only through the command line.

Generate a direct lookup table to approximate a function or subsystem

You can now approximate a function, subsystem, or math function with a Direct Lookup Table (n-D) block. Direct Lookup Table (n-D) blocks do not use breakpoint data, and instead index directly into the table data. To generate a Direct Look Table (n-D) block, in a `FunctionApproximation.Options` object, set the `Interpolation` property to `None`. This functionality is available only through the command line.

Generate a lookup table approximation from a function handle using the Lookup Table Optimizer app

Using the Lookup Table Optimizer app, you can now generate a lookup table that approximates a function handle. In previous releases, lookup table approximation of function handles was available only through the command line.

Generate lookup tables with flat and nearest interpolation methods

When an input falls between breakpoint values, the lookup table interpolates the output value using neighboring breakpoints. Using the `FunctionApproximation.Options` object, you can now specify `Flat` and `Nearest` interpolation methods. For more information on these interpolation methods, see `FunctionApproximation.Options`. This functionality is available only through the command line.

Automatically replace blocks with an optimized lookup table block

Using the Lookup Table Optimizer app, you can now automatically replace a block with an optimized lookup table. In previous releases you had to manually insert the optimized lookup table approximation into your model.

Data Type Optimization: Using parallel simulations, automatically select and apply heterogeneous data types for your system under design

Parallel support for data type optimization

The new `UseParallel` property of the `fxpOptimizationOptions` object allows you to specify whether to run iterations of the optimization in parallel. The default value of this property is `false`. Running the iterations in parallel requires a Parallel Computing Toolbox license. If you do not have a Parallel Computing Toolbox license, or if you specify `false`, the iterations run in serial.

New method for specifying required behavior of optimized design

Using the `addTolerance` method, you can now specify a time tolerance for your optimized design.

When the `tolerance_type` input argument is set to `'TimeTol'`, then `tolerance_value` defines a time interval, in seconds, in which the maximum and minimum values define the upper and lower values to compare against. For more information, see [How the Simulation Data Inspector Compares Data \(Simulink\)](#).

Single Precision Converter: Convert MATLAB Function blocks to single precision

Using the Single Precision Converter, you can automatically convert Simulink models and subsystems from double precision to single precision. Beginning in R2018b, the Single Precision Converter also converts MATLAB Function blocks from double precision to single precision.

To use the Single Precision Converter, from the Simulink **Analysis** menu, select **Data Type Design > Single Precision Converter**. Under **System under design**, select the system to convert to single-precision, then click **Convert to Single**.

For more information, see [Getting Started with Single Precision Converter](#).

cordicacos and cordicasin Functions: Compute fixed-point CORDIC inverse sine and cosine

The `cordicacos` and `cordicasin` functions provide a CORDIC-based approximation of the inverse cosine and inverse sine for use in fixed-point applications. For syntax and examples, see `cordicacos` and `cordicasin`.

Simulation Analysis and Performance: Instrumentation support for Fast Restart mode

Using the Fixed-Point Tool, you can now view instrumentation data for your model when it simulates in Fast Restart mode. In previous releases, only Normal mode simulation was supported for instrumentation in the Fixed-Point Tool. For more information about Fast restart mode, see [Get Started with Fast Restart \(Simulink\)](#)

Explore and debug Fixed-Point Tool results with sorting and filtering functionalities

Using the new **Explore** tab in the Fixed-Point Tool, you can now sort and filter results. The **Explore** tab enables you to sort results based on the following criteria:

- Block execution order
- Magnitude of logged simulation values
- Dynamic range of logged simulation values
- Data type properties, such as word length, integer length, or fraction length

You can filter results based on the following criteria:

- Data type
- Numerical issues, such as overflows or underflows

- Whether the logged simulation values are always whole numbers
- Signedness

To use the new sorting and filtering options, simulate a system using the Fixed-Point Tool with fixed-point instrumentation or signal logging turned on. The **Explore** tab is visible when the Fixed-Point Tool contains at least one run of instrumentation data.

Design and simulate half-precision systems in MATLAB

You can now specify half-precision floating-point data types in MATLAB. Half-precision data types occupy only 16 bits of memory, but their floating-point representation enables them to handle wider dynamic ranges than integer or fixed-point data types of the same size.

To cast a variable to half precision, use the `half` function.

```
a = half(pi)
```

```
half
```

```
3.1406
```

R2018a

Version: 6.1

New Features

Bug Fixes

Compatibility Considerations

Lookup table optimization: Approximate functions using a lookup table and optimize existing lookup tables to minimize RAM usage

Use the Lookup Table Optimizer to obtain an optimized (memory-efficient) lookup table that approximates an existing lookup table or math function. By replacing a floating-point math function block with a fixed-point lookup table, or optimizing the spacing and data types of an existing lookup table, you can improve the memory-efficiency of your algorithm.

To open the Lookup Table Optimizer, in your Simulink model, select **Analysis > Data Type Design > Lookup Table Optimizer**.

You can also use the command line interface to generate a memory-efficient lookup table. The command-line workflow also enables you to generate a lookup table from a MATLAB math function or function handle.

```
p = FunctionApproximation.Problem('sin')

p =

FunctionApproximation.Problem with properties

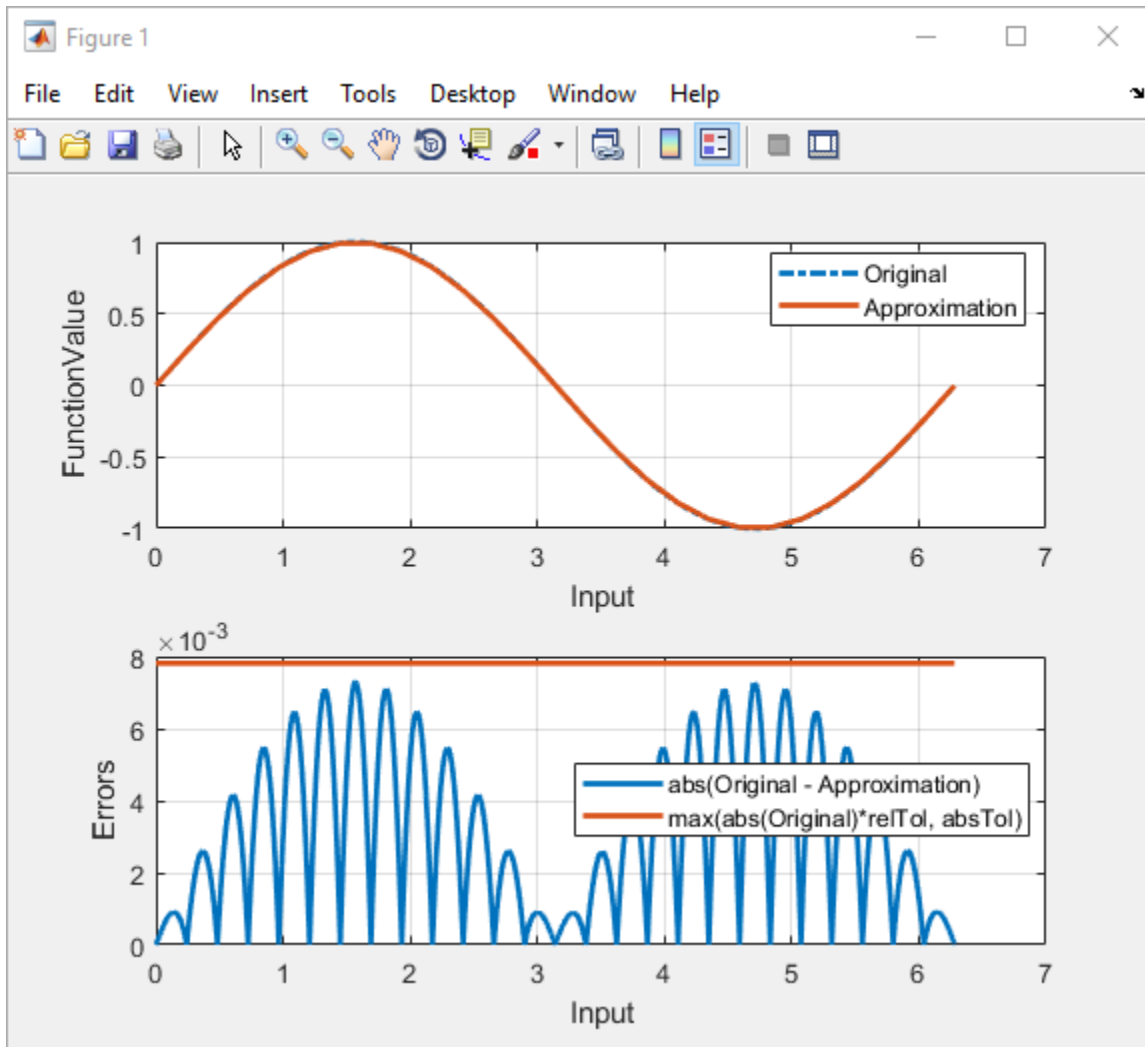
    FunctionToApproximate: @(x)sin(x)
      NumberOfInputs: 1
        InputTypes: "numerictype(0,16,13)"
  InputLowerBounds: 0
  InputUpperBounds: 6.2832
      OutputType: "numerictype(1,16,14)"
        Options: [1x1 FunctionApproximation.Options]
```

Specify additional options and constraints, such as the breakpoint specification.

```
p.Options.BreakpointSpecification = 'EvenSpacing'
```

Solve the optimization and compare the output of the original function with the output of the newly generated lookup table.

```
s = solve(p);
data = compare(s)
```



Data type optimization: Automatically select and apply heterogeneous data types for your system under design, optimizing bit width.

Use the `fxpopt` function to optimize the data types used in your system under design. You can specify constraints and tolerances to meet your design goals using the `fxpOptimizationOptions` object. The software analyzes ranges of objects in your system and your specified constraints, such as tolerances, to apply heterogeneous data types to your system while minimizing total bit width.

Redesigned code generation reports: View `fiaccel` and instrumentation results with improved user interface

In R2018a, the code generation reports for `fiaccel`, `buildInstrumentedMex`, and `showInstrumentationResults` have a new user interface.

Some benefits of the new use interfaces are:

- Improved navigation. For example, if you double-click a variable in the MATLAB code, you see the variable in the **Variables** tab.

- More information in the **Summary** tab of the `fiaccl` and `buildInstrumentedMex` reports. The **Summary** tab now includes code generation settings and your entry-point functions with the input argument data types that you specified.
- Easier to use pop-up displays data type information in the `showInstrumentationResults` report. For example, you can pin the pop-up display to the report.

In R2018a, the reports are located in the same folders as in previous releases, but have a different file format. In previous releases, a report was saved with an HTML format and consisted of many files. In R2018a, a report is saved as one file with an `.mldatx` file extension. You can open a file with an `.mldatx` extension in MATLAB.

Compatibility Considerations

If you generate a report in R2018a, you cannot open it in a previous release. In R2018a, you can open reports that you generated in a previous release, but they look and behave as they did in that release.

R2017b

Version: 6.0

New Features

Bug Fixes

Compatibility Considerations

Simplified Fixed-Point Tool: Convert Simulink systems to fixed point using the updated tool that provides guidance at each step of the workflow

The redesigned Fixed-Point Tool enables you to easily convert floating-point Simulink systems to fixed point. The new tool features a simplified, linear workflow, with better representation of the data.

Traceability between entries in the table, columns of the new data type visualization, and the model enable you to efficiently debug numerical issues and find the ideal fixed-point design for your system.

Launch the Fixed-Point Tool from any model from the **Analysis > Data Type Design > Fixed-Point Tool**, or by right-clicking the system you want to convert to fixed point and selecting **Fixed-Point Tool**.

The screenshot displays the Fixed-Point Tool interface. The main window shows a table of results for various data types in the model. The table includes columns for Name, Run, CompiledDT, SpecifiedDT, SimMin, and SimMax. Below the table, a histogram visualization shows the distribution of data values for each object, with a legend indicating Overflows, Representable, In-Range, and Underflows. The right-hand panel provides detailed information for the selected object, including its SpecifiedDT, Range Information, and Visualization of Simulation Data.

Name	Run	CompiledDT	SpecifiedDT	SimMin	SimMax
Combine Terms : Ac...	Ranges(Double)	double	Inherit: Inherit via inte...	-6.475416336873576	4.32700180757929
Combine Terms : O...	Ranges(Double)	double	fixdt(1,32,12)	-2.4135009037899...	4.32700180757929
Denominator Terms...	Ranges(Double)	double	fixdt(1,32,12)	-8.516638478410028	5.39648751512156
Denominator Terms...	Ranges(Double)	double	fixdt(1,32,12)	-6.475416336873576	3.4877081684371363
Denominator Terms...	Ranges(Double)	double	fixdt(1,32,12)	-8.516638478410028	5.39648751512156
Down Cast	Ranges(Double)	double	fixdt(1,16,5)	-2.4135009037899...	4.32700180757929
In1	Ranges(Double)	double	Inherit: auto		
Numerator Terms : ...	Ranges(Double)	double	fixdt(1,32,12)	-5.677304459288715	5.700524518426912
Numerator Terms : ...	Ranges(Double)	double	fixdt(1,32,12)	-3.367372640959928	3.5439615259925983
Numerator Terms : ...	Ranges(Double)	double	fixdt(1,32,12)	-5.677304459288715	5.700524518426912
Prev Out	Ranges(Double)	double			
Up Cast	Ranges(Double)	double	fixdt(1,16,14)	-2	3.999999999711746

Range Information

Property	Minimum	Maximum
Simulation	-2	3.999999999...

Visualization of Simulation Data

	Potential Overflows	In-Range	Potential Underflows
Positive Values	7	53	139
Negative Values	1	64	135

Number of times zero occurred: 0

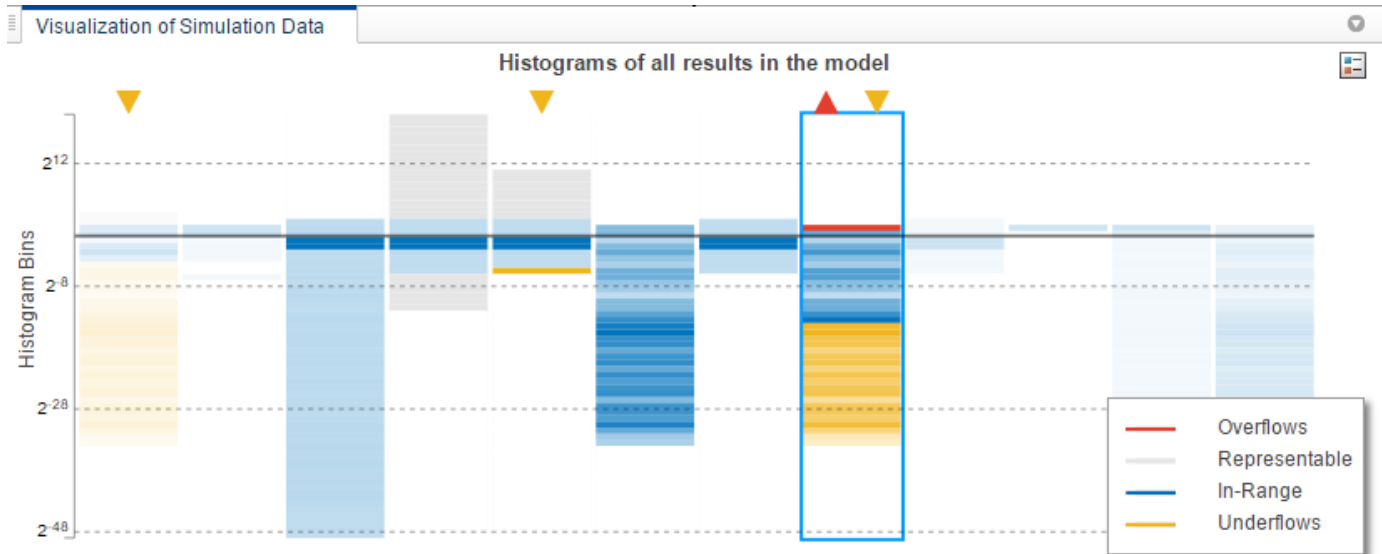
For more information, see [Autoscaling Using the Fixed-Point Tool](#).

Data Type Visualizer: Understand and analyze data type choices by viewing histograms of the dynamic range of signals in your model

Using the Fixed-Point Tool, you can now view a summary of histograms of the bits used by each object in your model. Each column in the data type visualization represents a histogram for one object in your model. Each bin in a histogram corresponds to a bit in the binary word.

Selecting a column highlights the corresponding model object in the spreadsheet of the Fixed-Point Tool, and populates the **Result Details** pane with more detailed information about the selected result.

Use this data type visualization to see a summary of the ranges of objects in your model and to quickly spot sources of overflows, underflows, and inefficient data types. To view the data type visualization, simulate a system with fixed-point instrumentation or signal logging turned on. Overflows are marked with a red triangle above the column representing the model object. Underflows are marked with a yellow triangle. For an example, see [Debug a Fixed-Point Model](#).



Data Type Exploration: Iteratively explore multiple floating point to fixed-point conversions to determine the optimal choice

In past releases, after applying fixed-point data types using the Fixed-Point Tool, you could no longer explore new default word length or fraction length choices. The tool would only rescale the existing fixed-point types. In R2017b, you can now propose and apply fixed-point data types using new proposal settings and default data types, and compare the behavior between runs until you find the optimal choice. For an example, see [Explore Multiple Floating-Point to Fixed-Point Conversions](#).

Function Input and Output Logging: Selectively log and plot function inputs and outputs at any level of your design in the Fixed-Point Converter app

You can now elect to log and plot all function inputs and outputs during the **Test** phase of fixed-point conversion using the Fixed-Point Converter app. In previous releases, only top-level function inputs and outputs could be logged.

To log a function input or output, on the **Convert to Fixed-Point** page, after converting your code, click the **Test** arrow and select the **Log inputs and outputs for comparison plots** check box. In the **Log Data** column of the **Variables** tab, select the check mark next to the function inputs and outputs you want to log. By default, all inputs and outputs of the top-level function are logged. To log

inputs and outputs of other functions in the call tree, select the function in the left pane, and select the variables you want logged.

The screenshot shows the MATLAB Fixed-Point Converter interface. The 'Source Code' pane on the left displays a call tree for the function 'kalman_filter', with 'kalman_filter > back_substitute' selected. The main editor shows the MATLAB code for 'forward_substitute' and 'back_substitute'. The 'Output Files' pane on the left shows generated files. The 'Variables' table at the bottom is shown with the 'Log Data' column highlighted in red, indicating that variables 'u', 'y', and 'x' are selected for logging.

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type	Log Data	Max Diff
Input							
u	double	1	2	No	numerictype(0, 16, 14)	<input checked="" type="checkbox"/>	
y	double	-0.25	1	No	numerictype(1, 16, 14)	<input checked="" type="checkbox"/>	
Output							
x	double	-0.19	0.5	No	numerictype(1, 16, 15)	<input checked="" type="checkbox"/>	
Local							
N	double		1	Yes	numerictype(0, 1, 0)	<input type="checkbox"/>	

When you are done selecting the variables you want to log, click **Test**.

The Fixed-Point Converter runs a floating-point and fixed-point simulation, then generates comparison plots and calculates the difference error for all variables logged.

Convert to Fixed Point

SETTINGS ANALYZE CONVERT TEST

Source Code kalman_filter_tb.m

```

1 function [y] = kalman_filter(z,N0)
2     %#codegen
3     A = kalman_stm();
4
5     % Measurement Matrix
6     H = [1 0];
7
8     % Process noise variance
9     Q = 0;
10    % Measurement noise variance
11    R = N0 ;
12
13    persistent x_est p_est
14    if isempty(x_est)
15        % Estimated state
16        x_est = [0; 1];
17        % Estimated error covariance
18        p_est = N0 * eye(2, 2);
19    end
20
21    % Kalman algorithm
22    % Predicted state and covariance
23    x_prd = A * x_est;
24    n_prd = A * n_est * A' + Q;

```

Output Files

- kalman_filter_fixpt.m
- kalman_filter_wrapper_fixpt.m
- index.html
- kalman_filter_fixpt_report.html
- kalman_filter_report.html
- kalman_filter_fixpt_args.mat
- kalman_filter_wrapper_fixpt_mex.r
- kalman_filter_fixpt_log.txt

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type	Log Data	Max Diff
Input							
z	double	-3.72	4.06	No	numeric(1, 16, 12)	✓	24.41e-05
N0	double	1	1	Yes	numeric(0, 1, 0)	✓	00.00e+00
Output							
y	double	-1.05	1.06	No	numeric(1, 16, 14)	✓	-29.63e-01
Persistent							
x_est	2 x 1 double	-1.05	1.06	No	numeric(1, 16, 14)		

Click the  icon in the **Max Diff** column to open the comparison plot.

For an example, see Debug Numerical Issues in Fixed-Point Conversion Using Variable Logging.

Simulink Diagnostic Management: Suppress immaterial diagnostic warnings and errors from specific blocks to efficiently discover modeling errors

You can now suppress certain diagnostics that are treated as errors for specific objects in your model. In past releases, only warning diagnostics were supported for suppression.

Click the **Suppress** button next to the error or warning in the Diagnostic Viewer to suppress the diagnostic from the specified source. You can restore the diagnostic from the source by clicking the **Restore** button.

You can also configure suppressions from the command line. For more information, see `Simulink.suppressDiagnostic` and `Simulink.restoreDiagnostic`.

Expanded Overflow Diagnostics: Comprehensive run-time diagnostics for wrapping and saturating overflows from Stateflow and MATLAB Function blocks

The Diagnostic Viewer now reports overflows due to wrap and saturation that occur within a MATLAB Function block or in a Stateflow chart that uses MATLAB as the action language. In cases of overflows that occur within a MATLAB Function block, the diagnostic includes the line number at which the overflow occurred.

You can suppress and restore these diagnostics at the block level by clicking the **Suppress** and **Restore** buttons respectively in the Diagnostic Viewer.

Autoscaling Lookup Table Objects: Propose and apply fixed-point data types for Simulink Lookup Table and Breakpoint objects

Using the Fixed-Point Tool, you can now propose and apply data types for Simulink LookupTable and Breakpoint objects used in your model, including within Lookup table, Prelookup, and Interpolation blocks. The Fixed-Point Tool detects these objects in your model and proposes a fixed-point data type based on their respective values, ranges, and constraints. The tool applies the proposed data type to the object by updating the object in the workspace in which it is defined. For more information on autoscaling data objects using the Fixed-Point Tool, see Autoscaling Data Objects Using the Fixed-Point Tool.

Check for expensive fixed-point data types in generated code

When a design contains integer or fixed-point word lengths that do not exist on your target hardware, the generated code can contain extra saturation code, shifts, and multiword operations. By changing the data type to one that is supported by your target hardware, you can improve the efficiency of the generated code. The Model Advisor flags these expensive data types in your model. For example, the Model Advisor would flag a fixed-point data type with a word length of 17 if the target hardware was 32 bits. For more information, see Optimize Generated Code with the Model Advisor.

Propose and apply data types for model reference blocks programmatically

A new syntax for the `DataTypeWorkflow.Converter` class enables you to specify a top model when converting a referenced model to fixed point. To convert a referenced model, `ref_model`, and collect ranges by simulating the referenced model from the top model, `top_model`, use the following syntax:

```
converter = DataTypeWorkflow.Converter(ref_model, 'TopModel', top_model)
```

For more information on converting systems to fixed point programmatically, see Command Line Interface for the Fixed-Point Tool.

cordictanh function for computing fixed-point CORDIC-based hyperbolic tangent

The `cordictanh` function provides a CORDIC-based approximation of the hyperbolic tangent for use in fixed-point applications. For syntax and examples, see `cordictanh`.

Functionality being removed or changed

Functionality	Result	Use This Instead	Compatibility Considerations
autofixexp	Still runs	DataTypeWorkflow.C onverter	For more information on how to use the DataTypeWorkflow.C onverter to convert a system to fixed point, see The Command-Line Interface for the Fixed-Point Tool.

R2017a

Version: 5.4

New Features

Bug Fixes

Compatibility Considerations

Simulink Diagnostic Management: Control which simulation and fixed-point diagnostic warnings you receive from specific blocks, including model reference

Select blocks with certain diagnostic suppressions by default

Beginning in R2017a, the Counter Free-Running, HDL Counter, Counter Limited, and Extract Bits blocks no longer report wrap on overflow warnings. The blocks continue to report errors due to wrap on overflows. You can restore the warning diagnostic by breaking the library link and using the `Simulink.restoreDiagnostic` function.

Diagnostic suppressor functions support `MSLDiagnostic` as input argument

You can now suppress and restore certain diagnostic warnings thrown by your model using a `Simulink.MSLDiagnostic` object as an input to the `Simulink.suppressDiagnostic` and `Simulink.restoreDiagnostic` functions.

To use simulation metadata and `MSLDiagnostic` objects, use `set_param` to set `ReturnWorkspaceOutputs` to on. Store the simulation output in a variable.

```
set_param(model_name, 'ReturnWorkspaceOutputs', 'on');
out = sim(model_name);
```

Access the `MSLDiagnostic` object through the simulation output.

```
diag = out.getSimulationMetadata.ExecutionInfo.WarningDiagnostics(1).Diagnostic
diag =
```

MSLDiagnostic with properties:

```
  identifier: 'SimulinkFixedPoint:util:fxpParameterPrecisionLoss'
  message: 'Parameter precision loss occurred for 'Value' of 'Suppressor_CLI_Demo/one'. The
  paths: {'Suppressor_CLI_Demo/one'}
  cause: {}
  stack: [0x1 struct]
```

Use the `Simulink.suppressDiagnostic` function to suppress the diagnostic warning specified by the `MSLDiagnostic` object, `diag`.

```
Simulink.suppressDiagnostic(diag)
```

You can restore the diagnostic using the `Simulink.restoreDiagnostic` function

```
Simulink.restoreDiagnostic(diag)
```

Improved workflow for suppressing diagnostics from referenced models

You can now suppress certain diagnostic warnings for specified instances of warnings in a referenced model. By accessing the `MSLDiagnostic` object of the specific instance of the warning, you can suppress the warning only when the block inside the referenced model is simulated from the specified top model.

Derived range analysis support for System objects in Simulink

Using the Fixed-Point Tool, you can now derive ranges for models that use handle objects, including System objects. For more information on range analysis in the Fixed-Point Tool, see [How Range Analysis Works](#).

Autoscaling support for Simulink.AliasType objects

Using the Fixed-Point Tool, you can now propose and apply data types for Simulink.AliasType objects used in your model. The Fixed-Point Tool detects alias type objects in your model and proposes a fixed-point data type based on their respective values and ranges. The tool applies the proposed data type to the alias type object by updating the definition of the object in the base workspace. For more information, see [Autoscale Simulink.AliasType Objects](#).

Improved data type proposals for shared data type groups across model reference

In past releases, there was limited traceability of model objects which were required to use the same data type across model reference boundaries. This often resulted in an update diagram error after applying proposed data types.

Beginning in R2017a, when the Fixed-Point Tool proposes data types for data objects in shared data type groups, the tool generates a proposal based on all collected ranges, including range information from data objects used inside referenced models. The Fixed-Point Tool can also now highlight all model elements that must use the same data type when the shared data type group crosses model reference boundaries.

More fixed-size variable information in Convert to Fixed-Point step of the Fixed-Point Converter app

In R2017a, in the Fixed-Point Converter app, after you convert floating-point MATLAB code to fixed-point MATLAB code, the app provides fixed-point type information for variables.

Variable	Type	Size	Signed	Word Length	Fraction Length
Input					
x	embedded.fi	1 x 256	Yes	16	14
Output					
y	embedded.fi	1 x 256	Yes	16	14
Persistent					
z	embedded.fi	2 x 1	Yes	16	15
Local					

In the code pane of the **Convert to Fixed-Point** step, after fixed-point conversion, if you place your cursor over a converted variable or expression, the app displays the fixed-point type information.

```

y = fi(zeros(size(x)), 1, 16, 14,
for i=1:length(x)
    y(i) = b(TYPE: (.) FIMATH ;
    z(1) = fi_signed(b(2)*x(i) + z
    z(2) Type: 1 x 256 embedded.fi : (i)
end
id
Function Rep Fraction Length: 14

```

For a variable with a fixed-point type in the original code, when you place your cursor over the variable before or after conversion, the app displays the fixed-point type information.

fimath property changes

All `fimath` property names are case-sensitive and require that you use the full property names. You cannot use truncated property names. In previous releases, when using truncated property names, a warning would appear. Beginning in R2017a, inexact property names result in an error.

Compatibility Considerations

To avoid seeing errors for `fimath` properties, update your code so it uses the full names and correct cases of all `fimath` properties. The full names and correct cases of the properties appear when you display a `fimath` object on the MATLAB command line.

R2016b

Version: 5.3

New Features

Bug Fixes

Single-Precision Conversion: Automatically convert double-precision systems to use single-precision data types in Simulink

Using the Single Precision Converter, you can now automatically convert Simulink models from double-precision to single-precision. The Converter makes these changes:

- Conversion of user-specified double-precision data types to single-precision data types (applies to block settings, Stateflow chart settings, signal objects, and bus objects.)
- Output signals and intermediate settings using inherited data types which compile to double-precision change to single-precision data types.

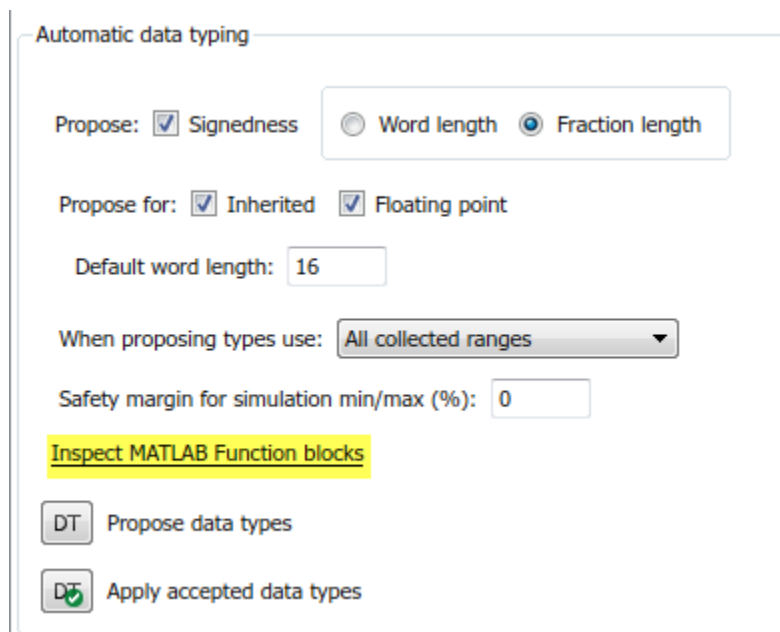
The converter does not change Boolean, built-in integer, or user-specified fixed-point data types. When the conversion is finished, the converter displays a table summarizing the compiled and proposed data types of the objects in the system under design. When the conversion is finished, a table summarizes the compiled and proposed data types of the objects in the system under design.

To use the Single-Precision Converter, from the Simulink **Analysis** menu, select **Data Type Design > Single Precision Converter**. Under **System under design**, select the system to convert to single-precision, then click **Convert to Single**.

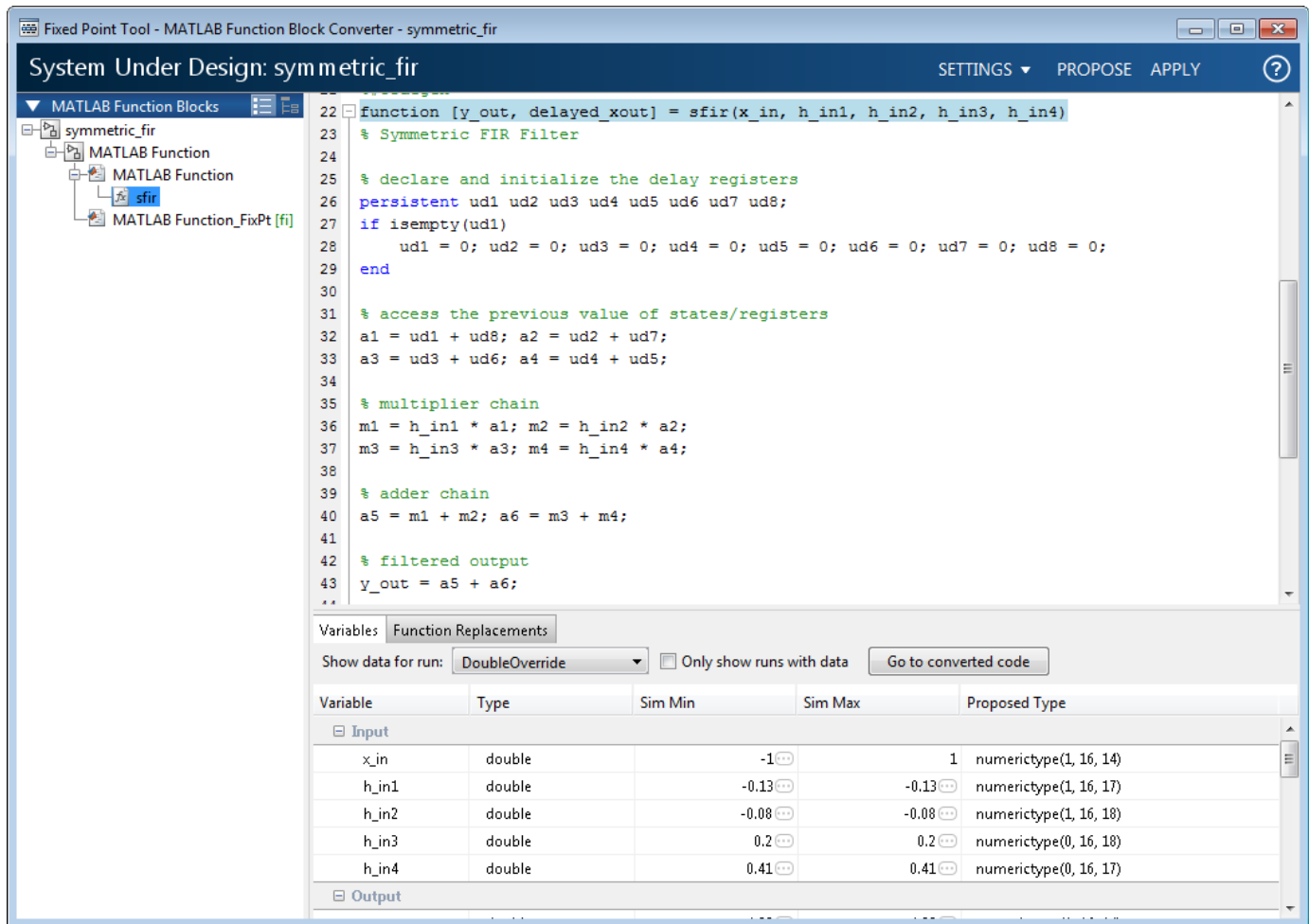
For more information, see Getting Started with Single Precision Converter.

Float to Fixed Conversion of MATLAB Function Blocks: Automatically generate fixed-point versions of floating-point MATLAB Function blocks

When converting a model that contains MATLAB Function blocks, you can now inspect type information of the MATLAB variables in the context of the code. This new code view provides a similar workflow to the Fixed-Point Converter app in MATLAB. To open the new code view, in the Fixed-Point Tool, under **Automatic Data Typing**, click **Inspect MATLAB Function blocks**.



The window that opens helps you to inspect advanced conversion settings such as `fimath` settings, and MATLAB function replacements.



Once you are satisfied with the proposed data types, click **Apply** to have the tool automatically generate a variant subsystem. The variant subsystem contains the original floating-point version of the MATLAB function block, and a fixed-point version of the block. You can refine the conversion by modifying the original floating-point MATLAB code. The fixed-point variant will automatically update after reconverting the block.

Histogram Instrumentation in Simulink: Generate log2 histograms of Simulink signals and blocks from simulation data

Using the Fixed-Point Tool, you can now view a histogram of bits used by each object in your system under design. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. The plot also includes the number of times that zero occurred. After simulating a system with fixed-point instrumentation or signal logging turned on, select an object in your model from the Contents pane of the Fixed-Point Tool and select the **Result Details** tab to view the histogram plot.

Workflow

Result Details

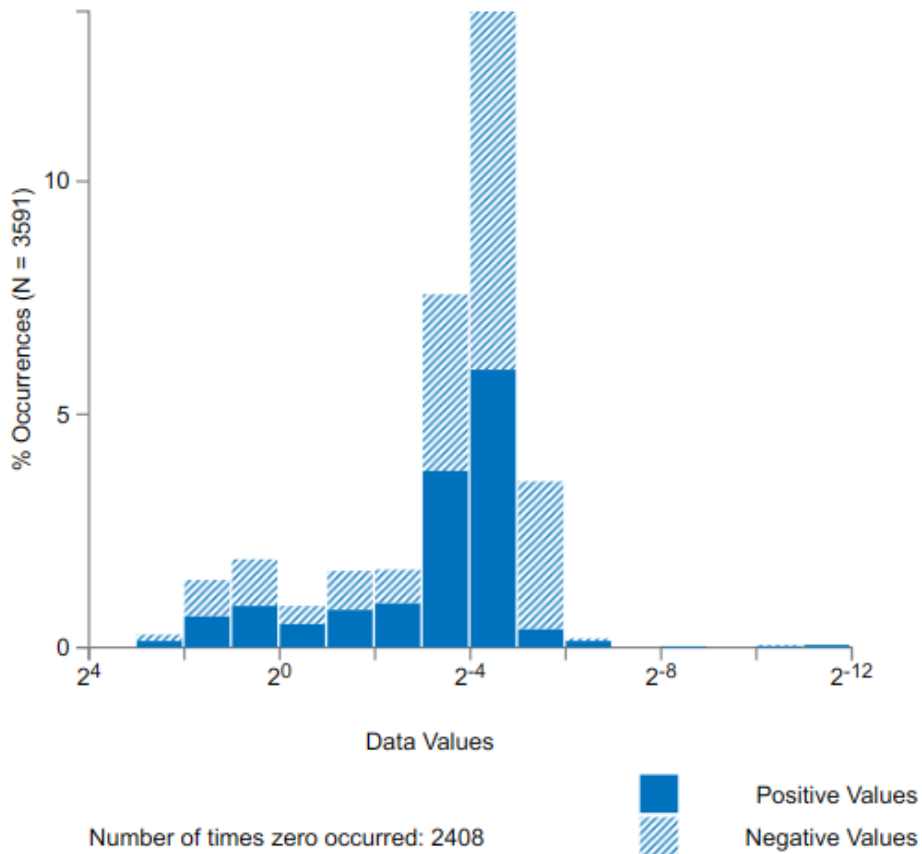
fxpdemo_feedback/Controller/Numerator Terms : Accumulator

Property	Data Type	Minimum	Maximum	Precision
SpecifiedDT	fixdt(1,32,28)	-8	7.99999999627471	3.7252902984619...

Range Information

Property	Minimum	Maximum
Simulation	-5.765953063964844	5.789508819580078

Histogram of Simulation Data



Autoscaling numerictype Objects: Propose and apply fixed-point data types for Simulink numeric type objects

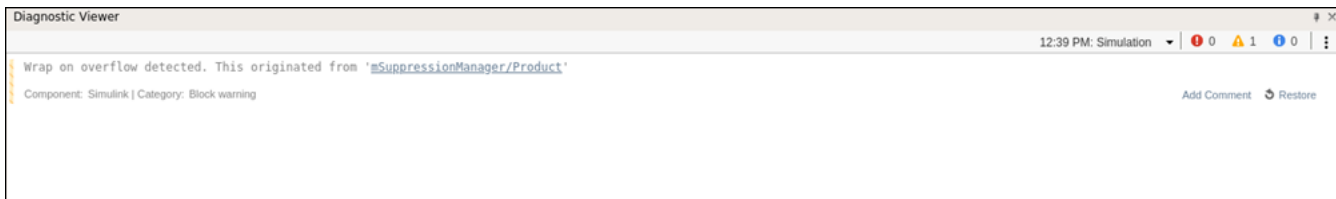
Using the Fixed-Point Tool, you can now propose and apply data types for `Simulink.NumericType` and `embedded.numerictype` objects used in your model. The Fixed-Point Tool detects numeric type objects in your model and proposes a fixed-point data type based on their respective values and ranges. The tool applies the proposed data type to the numeric type object by updating the definition of the object in the base or model workspace. For more information on autoscaling `Simulink.NumericType` objects, see [Autoscale Simulink.NumericType Objects](#).

Range analysis support for FIR filters, Dead Zone, and Rate Limiter blocks

Using the Fixed-Point Tool, you can now derive ranges for models that use Discrete FIR Filter, Dead Zone, and Rate Limiter blocks. For more information on range analysis in the Fixed-Point Tool, see [How Range Analysis Works](#).

Simulink Diagnostic Suppressor

The Diagnostic Viewer in Simulink now includes an option to suppress certain diagnostics. This feature enables you to suppress warnings for specific objects in your model. Click the **Suppress this warning** button next to the warning in the Diagnostic Viewer to suppress the warning from the specified source. You can restore the warning from the source by clicking **Restore this warning**.



You can also control the suppressions from the command line. For more information, see [Suppress Diagnostic Messages Programmatically](#).

Reduced number of multiplication helper functions

When you generate code for your model, there are now fewer generated multiplication helper functions. The new multiplication helper functions parameterize the shift amount for multiplication operations using binary-point scaling, reducing the need for separate functions in the generated code.

This change results in reduced memory consumption. This reduction in the amount of code generated from a model aids in the maintainability of your code base.

Improved accuracy of fixed-point sin, cos, and mod functions

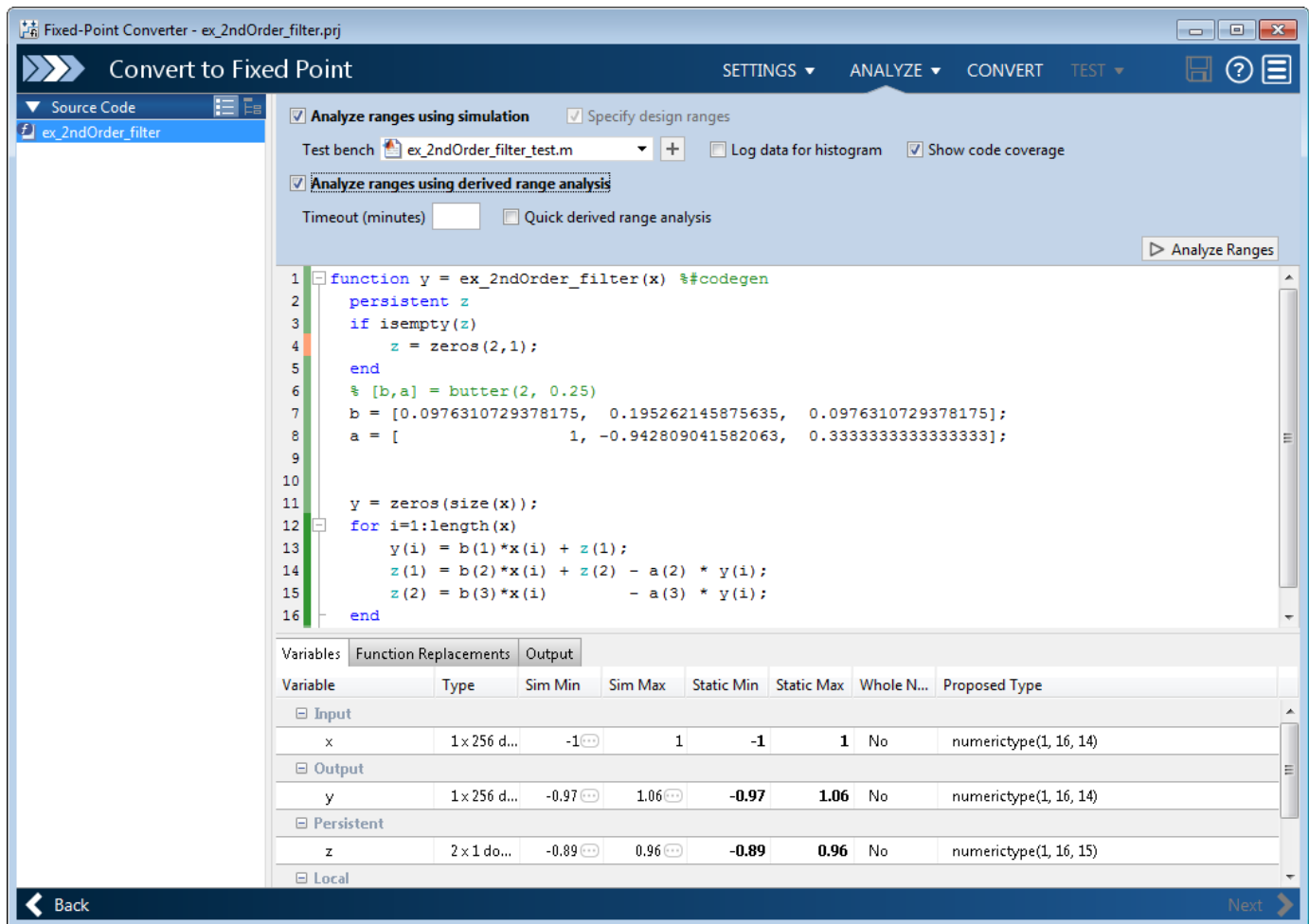
The fixed-point `sin` and `cos` functions are now more precise. In past releases these calculations were accurate only to within the top 16 most-significant bits of the input.

The mod function now has improved accuracy because it no longer limits internally-computed intermediate types to 32-bits or less.

For more information, see the `sin`, `cos`, and `mod` reference pages.

Improved workflow for collecting and analyzing ranges in the Fixed-Point Converter app

The **Simulate** and **Derive** buttons on the **Convert to Fixed Point** page of the Fixed-Point Converter app are now simplified and merged into a single **Analyze** button. This button controls which ranges (simulation ranges, design ranges, and derived ranges) are collected and used in the data type proposal phase of the conversion. When either the **Specify design ranges** or the **Analyze ranges using derived range analysis** options are selected, the **Static Min** and **Static Max** columns appear in the table. These columns do not appear when only the **Analyze ranges using simulation** option is selected, simplifying the view of the data. As in previous releases, you can still control which ranges are used for data type proposal in the **Settings** pane.



The screenshot shows the Fixed-Point Converter app interface. The main window is titled "Fixed-Point Converter - ex_2ndOrder_filter.prj". The "Convert to Fixed Point" page is active, with a settings pane on the left and a code editor on the right. The settings pane includes options for "Analyze ranges using simulation" (checked), "Specify design ranges" (checked), "Test bench" (set to "ex_2ndOrder_filter_test.m"), "Log data for histogram" (unchecked), "Show code coverage" (checked), "Analyze ranges using derived range analysis" (checked), and "Timeout (minutes)" (set to 0). The code editor shows the MATLAB function `ex_2ndOrder_filter` with the following code:

```

1 function y = ex_2ndOrder_filter(x) %#codegen
2     persistent z
3     if isempty(z)
4         z = zeros(2,1);
5     end
6     % [b,a] = butter(2, 0.25)
7     b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
8     a = [          1, -0.942809041582063, 0.333333333333333];
9
10
11    y = zeros(size(x));
12    for i=1:length(x)
13        y(i) = b(1)*x(i) + z(1);
14        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15        z(2) = b(3)*x(i)          - a(3) * y(i);
16    end

```

Below the code editor is a table showing the ranges for variables. The table has columns for Variable, Type, Sim Min, Sim Max, Static Min, Static Max, Whole N..., and Proposed Type. The variables are categorized into Input, Output, Persistent, and Local.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole N...	Proposed Type
Input							
x	1 x 256 d...	-1	1	-1	1	No	numerictype(1, 16, 14)
Output							
y	1 x 256 d...	-0.97	1.06	-0.97	1.06	No	numerictype(1, 16, 14)
Persistent							
z	2 x 1 do...	-0.89	0.96	-0.89	0.96	No	numerictype(1, 16, 15)
Local							

R2016a

Version: 5.2

New Features

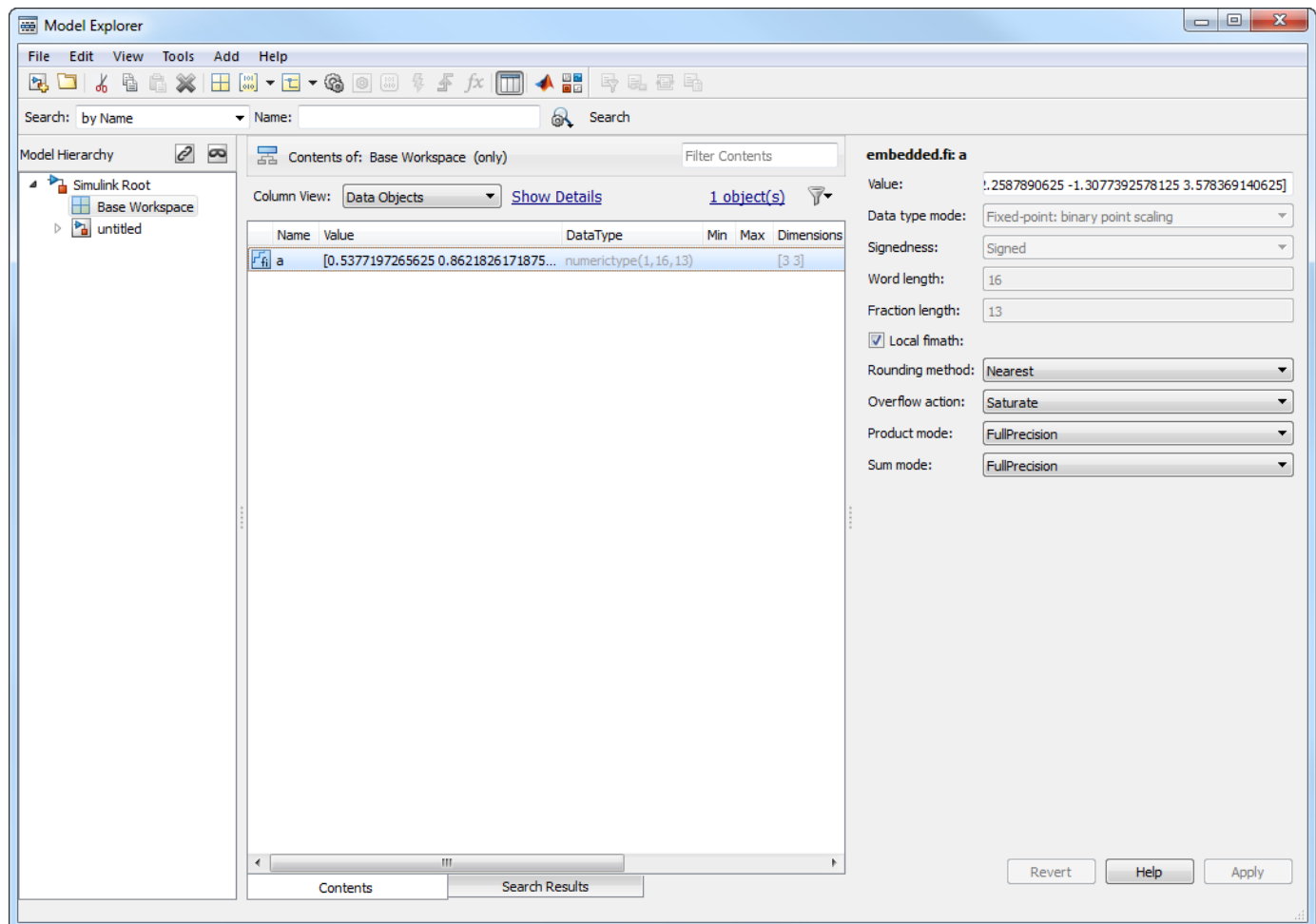
Bug Fixes

Autoscaling Parameter Objects: Automatically propose and apply data types for parameter objects

Using the Fixed-Point Tool, you can now propose and apply data types for parameter objects used in your model. The Fixed-Point Tool detects parameter objects in your model and proposes a fixed-point data type based on their respective values and ranges. The tool applies the proposed data type to the parameter object by updating the definition of the parameter object in the base or model workspace. For more information, see [Autoscale Simulink.Parameter Objects](#).

View and edit fi objects in Model Explorer

You can now view and edit `fi` objects and their local `fi` properties using Model Explorer in Simulink. You can change the writable properties of `fi` objects from the Model Explorer. You cannot change the numeric type properties of `fi` objects after creation.



Simulate system level designs that integrate referenced models targeting an assembly of heterogeneous embedded devices

When modeling larger systems, models are often composed of referenced models that target various embedded devices. You can now simulate a parent system model that includes referenced models

configured with mismatching hardware settings for different embedded devices. In past releases, Simulink required the hardware settings on referenced models to match to simulate the top-level model. You can configure the hardware implementation settings in the **Configuration Parameters > Hardware Implementation** pane.

Enhancements to Fixed-Point Converter app

Support for arrays of structures

You can now convert arrays of structures to fixed point using the Fixed-Point Converter app. For more information on language features supported by the Fixed-Point Converter app, see MATLAB Language Features Supported for Automated Fixed-Point Conversion.

Structures in generated fixed-point code

The Fixed-Point Converter now proposes a unified data type for structures that are similar. Similar structures are structures which contain fields with the same name, number and type. The Fixed-Point Converter app no longer generates copies of structures, making the generated fixed-point code more efficient. See Convert Code Containing Structures to Fixed Point.

Revert changes to input type definitions

You can now revert and restore changes to type definitions in the **Define Input Types** step of the Fixed-Point Converter app. You can revert or restore changes in the entry-point input arguments table or the global variables table.

Use the undo and redo buttons for the table that you want to change. Alternatively, use the keyboard shortcuts for undo and redo. The keyboard shortcuts apply to the selected table. The shortcuts are defined in your MATLAB preferences. The default keyboard shortcuts for undo and redo on a Windows® platform are **Ctrl+Z** and **Ctrl+Y**.

View complete error message in error table

In previous releases, the Fixed-Point Converter app truncated a message that did not fit on one line of the error messages table on the **Convert to Fixed-Point** step. In R2016a, the app displays a long message on multiple lines so that you can see the entire message.

Additional keyboard shortcuts in the code generation report

You can now use keyboard shortcuts to perform the following actions in a code generation report.

Action	Default Keyboard Shortcut for a Windows platform
Zoom in	Ctrl+Plus
Zoom out	Ctrl+Minus
Evaluate selected MATLAB code	F9
Open help for selected MATLAB code	F1
Step backward through files that you opened in the code pane	Alt+Right
Refresh	F5

Action	Default Keyboard Shortcut for a Windows platform
Find	Ctrl+F

Your MATLAB preferences define the keyboard shortcuts associated with these actions. You can also select these actions from a context menu. To open the context menu, right-click anywhere in the report.

Changes to Fixed-Point Conversion Code Coverage

If you use the Fixed-Point Converter app to convert your MATLAB code to fixed-point code and propose types based on simulation ranges, the app shows code coverage results. In previous releases, the app showed the coverage as a percentage. In R2016a, the app shows the coverage as a line execution count.

11	<code>persistent current_state</code>	
12	<code>if isempty(current_state)</code>	
13	<code>current_state = S1;</code>	1 calls
14	<code>end</code>	51 calls
15		
16	<code>% switch to new state based on the value state register</code>	
17	<code>switch uint8(current_state)</code>	
18	<code>case S1</code>	
19	<code> % value of output 'Z' depends both on state and inputs</code>	
20	<code> if (A)</code>	
21	<code> Z = true;</code>	37 calls
22	<code> current_state(1) = S1;</code>	
23	<code> else</code>	7 calls
24	<code> Z = false;</code>	
25	<code> current_state(1) = S2;</code>	
26	<code> end</code>	
27	<code>case S2</code>	51 calls
28	<code> if (A)</code>	
29	<code> Z = false;</code>	7 calls
30	<code> current_state(1) = S1;</code>	
31	<code> else</code>	0 calls
32	<code> Z = true;</code>	
33	<code> current_state(1) = S2;</code>	
34	<code> end</code>	
35	<code>case S3</code>	51 calls
36	<code> if (A)</code>	
37	<code> Z = false;</code>	0 calls
38	<code> current_state(1) = S2;</code>	
39	<code> else</code>	
40	<code> Z = true;</code>	
41	<code> current_state(1) = S3;</code>	
42	<code> end</code>	

For more information, see [Code Coverage](#).

R2015aSP1

Version: 5.0.1

Bug Fixes

R2015b

Version: 5.1

New Features

Bug Fixes

Simulink Fixed-Point Tool workflow simplification: Propose signedness and data types for inherited and floating-point types

System under design (SUD) specification

Upon opening the Fixed-Point Tool, you must now select the system under design for fixed-point conversion. Once selected, the system name will appear highlighted in green in the **Model Hierarchy** pane. The Fixed-Point Tool will propose and apply data types for the selected system only.

To change the system under design, click **Change**. In the dialog, select the system you want to convert.

Signedness proposals

The Fixed-Point Tool now proposes signedness for blocks in your system under design. To get signedness proposals for blocks in your model, in the **Automatic data typing** pane, select the **Signedness** check box.

Automatic data typing

Propose: Signedness Word length Fraction length

Propose for: Inherited Floating point

Default word length:

When proposing types use:

Safety margin for simulation min/max (%):

Propose data types

Apply accepted data types

The Fixed-Point Tool bases its signedness proposals on collected range information and block constraints. Signals that are always strictly positive now get an unsigned data type proposal, gaining an additional bit of precision compared to previous releases.

By default, the **Signedness** check box is selected. If you clear the check box, the Fixed-Point Tool proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the Fixed-Point Tool will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

Proposals for objects using inherited and floating-point types

You can now elect to receive proposals for objects in your model that use floating-point data types or one of the inherited data types for block outputs. To get proposals for objects using floating-point or inherited data types, in the **Automatic data typing** pane, select the corresponding check boxes.

Automatic data typing

Propose: Signedness Word length Fraction length

Propose for: Inherited Floating point

Default word length:

When proposing types use:

Safety margin for simulation min/max (%):

Propose data types

Apply accepted data types

By default, the **Inherited** and **Floating point** check boxes are selected. If you clear the **Inherited** or **Floating point** check boxes, the Fixed-Point Tool will not propose a fixed-point data type for results that use an inherited or floating-point data type respectively.

Two-way traceability between model and Fixed-Point Tool

You can now trace between Simulink blocks in your model and their corresponding results in the Fixed-Point Tool. This capability simplifies the task of debugging overflows and other data type propagation issues in your model. Right-click on a block in your Simulink model and select **Fixed-Point Tool Result** to highlight the result in the **Contents** pane of the Fixed-Point Tool. You can also trace a result back to the model by right-clicking a result in the **Contents** pane and selecting **Highlight in Editor**.

New configurations for model settings

Under **Configure model settings** in the Fixed-Point Tool, use the configurations to set up your model for range collection.

- The **Range collection using double override** configuration overrides the data types in your model to doubles and enables instrumentation of your model. Use these settings to collect simulation ranges using ideal floating-point data types.
- The **Range collection with specified data types** configuration removes data type override and enables instrumentation of your model. Use this shortcut to collect simulation ranges using the data types specified in your model and to validate current behavior.
- The **Remove overrides and disable range collection** configuration restores your model to its specified numeric behavior and disables instrumentation to restore maximum speed. Use this shortcut to clean up model settings after conversion.

Double-precision to single-precision conversion: Convert double-precision MATLAB code to single-precision MATLAB code using the command line

In R2015b, you can use the `convertToSingle` function to convert double-precision MATLAB code to single-precision MATLAB code.

You can verify the behavior of a single-precision version of your code without modifying the original algorithm. When a double precision operation cannot be removed, the report highlights the MATLAB expression that results in that operation.

For example, to generate single-precision MATLAB code from a double-precision function `myfunction` that takes two double arguments:

```
convertToSingle myfunction -args {1 2}
```

To use verification options, create a `coder.SingleConfig` object that you pass to `convertToSingle`. You can:

- Test numerics by running the test file with the single-precision types applied.
- Compare double-precision and single-precision test results using the Simulation Data Inspector or your own plotting functions.

```
scfg = coder.config('single');  
scfg.TestBenchName = 'myfunction_test';  
scfg.TestNumerics = true;  
scfg.LogIOForComparisonPlotting = true;  
convertToSingle -config scfg myfunction -args {1 2}
```

If you also have a MATLAB Coder license, you can:

- Generate single-precision C code using the MATLAB Coder app. Use this workflow if your goal is to generate single-precision C code in the most direct way and you do not want to see the intermediate single-precision MATLAB code.
- Generate single-precision C code using `codegen` with the `-singleC` option. Use this workflow when you want to generate single-precision C code in the most direct way and you do not want to see the intermediate single-precision MATLAB code
- Generate single-precision MATLAB code using `codegen` with a `coder.SingleConfig` object. Use this workflow if you want to see the single-precision MATLAB code or use verification options.
- Generate single-precision C code using `codegen` with a `coder.SingleConfig` object and a code configuration object. Use this workflow to generate single-precision C code when you also want to see the single-precision MATLAB code or use verification options.

For more information about single-precision conversion using MATLAB Coder, see the MATLAB Coder release notes.

MATLAB Fixed-Point Converter app streamlined workflow: Restore project state and minimize regeneration of MEX files

Saving and restoring fixed-point conversion workflow state in the app

If you close a project before completing the fixed-point conversion process, the app saves your work. When you reopen the project, the app restores the state. You do not have to repeat the fixed-point

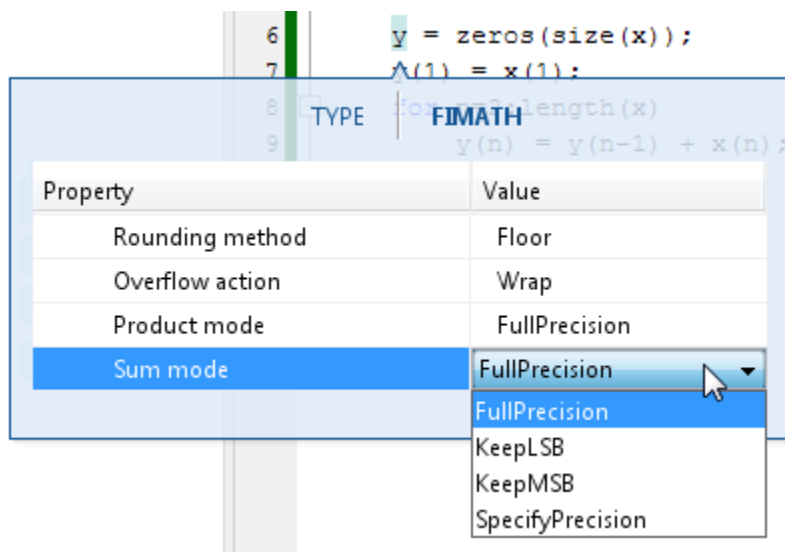
conversion steps that you completed in a previous session. For example, suppose you close the project after data type proposal. When you reopen the project, the app shows the results of the data type proposal and enables conversion. You can continue where you left off.

Minimized regeneration of MEX files

The Fixed-Point Converter app now optimizes when it regenerates MEX files. The app will only rebuild the MEX file when required by changes in your code.

Specification of additional fimath properties in app editor

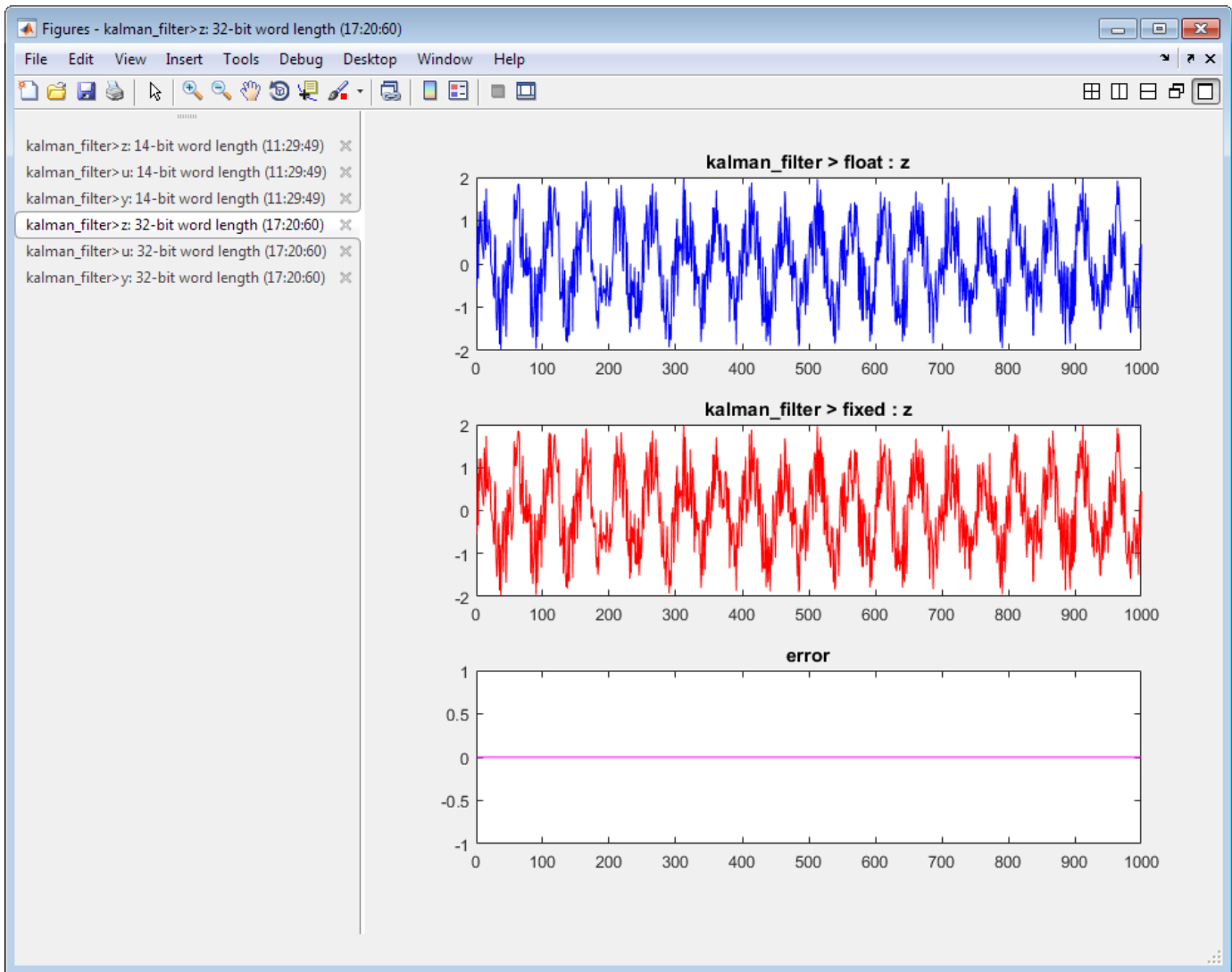
You can now control all `fimath` properties of variables in your code from within the Fixed-Point Converter app editor. To modify the `fimath` settings of a variable, select a variable and click **FIMATH** in the dialog that appears. You can alter the Rounding method, Overflow action, Product mode, and Sum mode properties. You can also modify these properties from the settings pane. For more information on these properties, see `fimath`.



Improved management of comparison plots

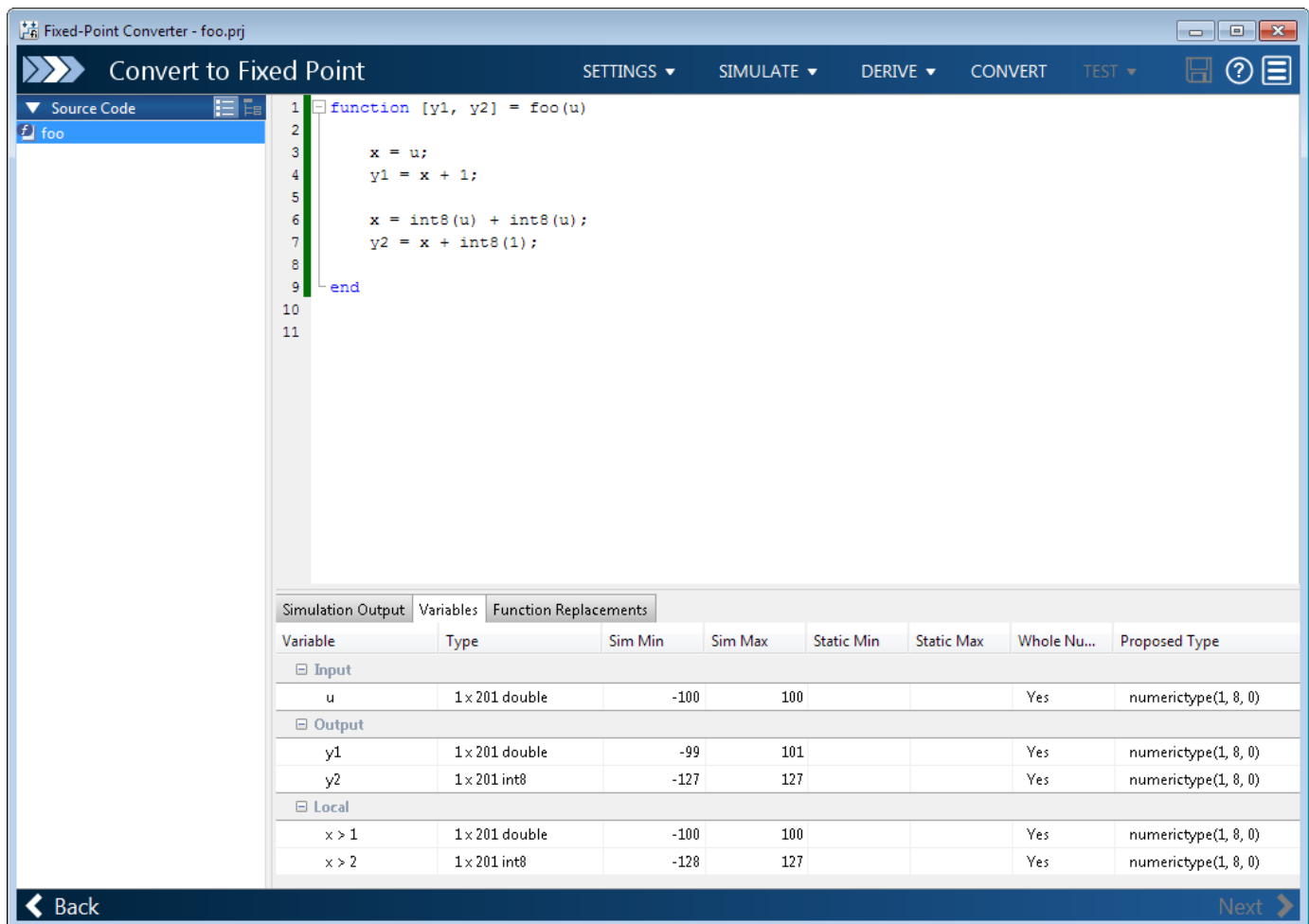
The Fixed-Point Converter app now docks plots generated during the testing phase of your fixed-point code into separate tabs of one figure window. Each tabbed figure represents one input or output variable and is labeled with the function, variable, word length, and a timestamp. Each tab contains three sub plots. The plots use a time series based plotting function to show the floating-point and fixed-point results and the difference between them.

Subsequent iterations are also plotted in the same figure window.



Variable specializations

On the **Convert to Fixed Point** page, in the **Variables** table of the app, you can now view variable specializations.



Improvements to Readability of Generated Code

Structs

- When struct copies exist in the design, a separate function is now created to perform the copy.
- Copies of structs are now avoided when the types of all fields match, improving both readability and efficiency of the generated code.

fimath

- fimath settings are now specified in a separate function to improve the readability of the generated fixed-point code.
- To avoid a mismatch of fimath settings in an expression, the generated code now uses the removefimath function.

```
function [y] = my_add_fixpt(a,b)
%Adds a and b
fm = getConversionFimath();

y=fi(removefimath(a)+b, 0, 8, 0, fm);
end
```

```
function fm = getConversionFimath()
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
        'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
        'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);
end
```

Matrices

Growth and deletion of matrices within a design are now supported for fixed-point conversion.

```
function matrix_deletion_fixpt(a,i)
    fm = getConversionFimath();

    var = fi([1, 2, 3], 0, 2, 0, fm);
    coder.varsize('var');
    var(2) = []; % matrix deletion.
    var(2) = fi(2, 0, 2, 0, fm);
end

function [out] = matrix_growth_fixpt( x )
    fm = getConversionFimath();
    out = fi([], 0, 4, 0, fm);
    for ii = 1:10
        out = [ out x];
    end
end
```

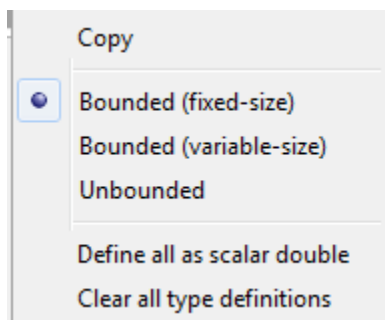
Tab completion for specifying files

On the **Select Source Files** and **Define Input Types** pages of the Fixed-Point Converter app, you can now use tab completion to specify your entry-point functions and test bench file.

Improvements for manual type definition

Improvements for manual type definition include:

- New right-click menus options to specify array size.



- Easier definition of structure types. You can:
 - Use the new **+** icon to add fields.
 - See the structure type name in the table of input variables.

x	struct(1 x 2)	myname
field1	double[1 x 1]	

- Easier definition of embedded `.fi` types. You can:
 - See the `numerictype` properties in the table of input variables.



- Use the new  icon to change the `numerictype` properties.

Compatibility between the app colors and MATLAB preferences

The app uses colors that are compatible with the **Desktop tool colors** preference in the MATLAB preferences. For information about MATLAB preferences, see Preferences.

Range analysis for Delay blocks: Improve accuracy and speed of range analysis on models using Delay blocks

Using the Fixed-Point Tool, you can now derive ranges for models that use Delay blocks with greater precision. The Fixed-Point Tool can also derive ranges for certain configurations of cascading Delay blocks with greater theoretical accuracy and speed. For more information on range analysis in the Fixed-Point Tool, see How Range Analysis Works.


Control of signed shifts in fixed-point scaling operations: Control the use of signed shifts in generated code

You can now control the use of signed right shifts in your generated code. Some coding standards do not allow bitwise operations on signed integers. Disabling the use of signed shifts in generated code increases the likelihood of compliance with MISRA. When you specify that signed right shifts should not be used in your generated code, the software replaces signed shifts with a call to a function that performs the operation without the use of signed shifts.

This feature requires an Embedded Coder license.

MATLAB

To specify that MATLAB Coder not use signed right shifts:

- Using the MATLAB Coder app:
 - 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
 - 2 Set **Build type** to one of the following:
 - Source Code
 - Static Library (.lib)
 - Dynamic Library (.dll)
 - Executable (.exe)
 - 3 Click **More Settings**.
 - 4 On the **Code Appearance** tab, clear the **Allow right shifts on signed integers** check box.
- Using the command-line interface:

- 1 Create a code configuration object for 'lib', 'dll', or 'exe'.

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

- 2 Set the EnableSignedRightShifts property to false.

```
cfg.EnableSignedRightShifts = false;
```

Simulink

To specify that the code generator not use signed right shifts, in the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane, clear Allow right shifts on signed integers or set the parameter EnableSignedRightShifts to off.

To improve coding standard compliance for bitwise operations on signed integers, run the following checks:

- Check for bitwise operations on signed integers - Check to identify blocks that contain bitwise operations on signed integers.
- Check configuration parameters for MISRA C:2012 - Check that verifies that you cleared **Code Generation > Code Style > Allow right shifts on signed integers**.

Access full-precision value of fi object in decimal and string format

You can now set and get full-precision real-world values of `fi` objects using the new `Value` property. This provides easy access to exact values in decimal format.

The `toString` function now accepts `fi` object inputs allowing you to convert `fi` objects to a string that you can copy and paste into a MATLAB script or function. The `mat2str` function now also supports `fi` object inputs allowing you to convert `fi` objects to strings without first converting to a double value.

Detection of multiword operations

When an operation has an input or output larger than the largest word size of your processor, the generated code contains multiword operations. Multiword operations can be inefficient on hardware. In both MATLAB and Simulink, you can now detect operations that will result in multiword code.

MATLAB

The expensive fixed-point operations check now highlights expressions in your MATLAB code that could result in multiword operations in generated code. For more information on enabling this check, see Find and Address Multiword Operations.

Simulink

The Identify questionable fixed-point operations check in the Model Advisor now detects multiword operations in generated code. For more information, see Identify Questionable Fixed-Point Operations.

Enhanced Model Advisor check for implementing strict single-precision designs

The Model Advisor **Modeling Single-Precision Systems > Identify questionable operations for strict single-precision design** check now verifies the status of additional model settings that will help you achieve a strict single-precision design.

- The Model Advisor warns you if **Configuration Parameters > Optimization > Default for underspecified data type** is set to Double.
- The Model Advisor warns you if your model uses library standard that is not optimal for strict-single precision designs.
- The Model Advisor warns you if **Configuration Parameters > Optimization > Implement logic signals as Boolean data** is not selected.

The settings suggested by the Model Advisor prevent the introduction of doubles into your generated code, which is optimal for strict-single designs.

System object instrumentation in Fixed-Point Tool

The Fixed-Point Tool now collects simulation ranges and proposes data types for select DSP System Toolbox™ System objects used inside a MATLAB Function block. You cannot propose data types based on derived range data.

Use of these System objects requires a DSP System Toolbox license. To learn more about using the Fixed-Point Tool to convert System objects and to learn which System objects are supported, see [Convert a System Object to Fixed Point Using the Fixed-Point Tool](#).

R2015a

Version: 5.0

New Features

Bug Fixes

Derived Ranges for MATLAB Function Blocks in Simulink

Using the Fixed-Point Tool, you can now derive ranges for variables inside a MATLAB Function block in Simulink. The Fixed-Point Tool uses design ranges to derive ranges for MATLAB variables in a MATLAB Function block. The tool can also propose data types for the variables based on the derived range data. You must manually apply the proposed data types to the variables. For more information, see [Derive Ranges of MATLAB Function Block Variables](#).

Fixed-Point Converter app enhancements, including detection of dead and constant folded code, support for projects with multiple entry point functions and support for global variables

The following enhancements have been added to the Fixed-Point Converter app:

Support for projects with multiple entry-point functions

You can now specify multiple entry-point functions in a Fixed-Point Converter app project. If your end goal is to generate fixed-point C/C++ library functions, conversion with multiple entry-point functions facilitates integration with larger applications. For more information, see [Generate Fixed-Point MATLAB Code for Multiple Entry-Point Functions](#).

Support for global variables

You can now specify global variables in the Fixed-Point Converter app workflow and convert algorithms which contain global variables without modifying your code. For more information, see [Convert Code Containing Global Variables to Fixed-Point](#).

Code coverage based translation

The Fixed-Point Converter app now detects dead and constant folded code within your project and warns you if any parts of your code were not executed during the simulation of your test file. This can help you verify if your test file is testing the algorithm over the intended operating range. The app uses this code coverage information during the translation of your code from floating-point MATLAB code to fixed-point MATLAB code. The app inserts inline comments in the fixed-point code to mark the dead and untranslated regions and includes the code coverage information in the generated fixed-point conversion html report. This code coverage information is also available from the command-line workflow. For more information, see [Detect Dead and Constant-Folded Code](#).

Conversion from project to MATLAB scripts for command-line fixed-point conversion

Using the `-tocode` option of the `fixedPointConverter` command, you can convert a fixed-point conversion project to the equivalent MATLAB code in a MATLAB script. You can use the script to repeat the project workflow in a command-line workflow. For more information, see [Convert Fixed-Point Conversion Project to MATLAB Scripts](#).

Generated fixed-point code enhancements

The generated fixed-point code now:

- Uses colon syntax for multi-output assignments, reducing the number of `fi` casts in the generated fixed-point code.
- Preserves the indentation and formatting of your original algorithm, improving the readability of the generated fixed-point code.

Integration with MATLAB Coder app interface

The Fixed-Point Converter app has been integrated into the new MATLAB Coder app workflow. This integration allows for a smoother conversion process from floating-point MATLAB code to fixed-point C/C++ code.

Automated conversion of additional DSP System objects using the Fixed-Point Converter app

You can now convert the following DSP System Toolbox System objects to fixed-point using the Fixed-Point Converter app:

- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter`, direct form and direct form transposed only
- `dsp.LUFactor`
- `dsp.VariableFractionalDelay`
- `dsp.Window`

You can propose and apply data types for these System objects based on simulation range data. During the conversion process, you can view simulation minimum and maximum values and proposed data types for these System objects. You can also view whole number information and histogram data. You cannot propose data types for these System objects based on static range data. This requires a DSP System Toolbox license.

Fixed-Point SimState logging and root logging improvements

The Simulink SimState feature allows you to save all run-time data necessary for restoring the simulation state of the model. A SimState includes both the logged and internal state of every block and the internal state of the Simulink engine. The Fixed-Point Tool now supports SimState logging while fixed-point instrumentation is turned on. For more information, see Save and Restore Simulation State as SimState.

Flexible structure assignment of buses

When a non-tunable structure is assigned to a bus signal (such as a block which uses a structure for its initial condition parameter), the data type of the fields of the structure no longer need to match the data type of the bus elements. The software now performs an automatic casting of the data type of the structure field so that it matches the data type of the bus signal. This flexible structure assignment simplifies the fixed-point conversion workflow by automatically casting the data type of the fields of the structure when using data type override and autoscaling your model.

eye(m,'like',a) syntax supported for fixed-point inputs

The `eye` function now works with fixed-point data types as well as built-in data types. The function can now return an output whose class matches that of a specified numeric variable or `fi` object. For built-in data types, the output assumes the numeric data type, sparsity, and complexity (real or complex) of the specified numeric variable. For `fi` objects, the output assumes the `numericType`, complexity (real or complex), and `fimath` of the specified `fi` object.

New interpolation method for generating lookup table MATLAB function replacements

The `coder.approximation` function now offers a 'Flat' interpolation method for generating lookup table MATLAB function replacements. This fully-specified lookup table achieves high speeds by discarding the pre-lookup step and reducing the use of multipliers in the data path. This interpolation method is available from both the command-line workflow, and in the **Function Replacements** tab of the Fixed-Point Converter app.

Fixed-point scaling information in Code Interface Report

Fixed-point scaling information is added to the code generation report in the Code Interface Report section. Better accessibility to this information makes it easier for you to integrate with generated code containing fixed-point data types. Each fixed-point entry in the report table has a value in the new **Scaling** column giving its data type and fraction length using Simulink fixed-point data type notation.

Access to the Code Interface Report requires an Embedded Coder license.

R2014b

Version: 4.3

New Features

Bug Fixes

Compatibility Considerations

Fixed-Point Converter app for automated conversion of floating-point MATLAB code


The Fixed-Point Converter app enables you to convert floating-point MATLAB code to fixed-point MATLAB code.

You can choose to propose data types based on simulation range data, static range data, or both.

During fixed-point conversion, you can:

- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Test numerics by running the test file with the fixed-point types applied.
- Compare floating-point and fixed-point test results using the Simulation Data Inspector or your own plotting functions.
- View a histogram of bits used by each variable.
- Specify replacement functions and generate approximate functions for functions in the original MATLAB algorithm that are not supported for fixed point.

To open the app:

- In the MATLAB Toolstrip, on the **Apps** tab, under **Code Generation**, click .
- At the MATLAB command prompt, enter `fixedPointConverter`.

For more information, see Fixed-Point Converter.

Commands for scripting fixed-point conversion and accessing the collected data in Simulink

You can now use the `DataTypeWorkflow.Converter` class to collect simulation and derived data, propose and apply data types to the model, and analyze results.

This class performs the same fixed-point conversion tasks as the Fixed-Point Tool. This facilitates scripting of the automatic conversion workflow and accessing data for analysis. For more information, see [Convert a Model to Fixed Point Using the Command-Line](#).

Automated fixed-point conversion for commonly used DSP System objects, including Biquad Filter, FIR Filter, and FIR Rate Converter

You can now convert the following DSP System Toolbox System objects to fixed point using the Fixed-Point Converter app.

- `dsp.BiquadFilter`
- `dsp.FIRFilter`, direct form only
- `dsp.FIRRateConverter`

-
- `dsp.LowerTriangularSolver`
 - `dsp.UpperTriangularSolver`
 - `dsp.ArrayVectorAdder`

You can propose and apply data types for these System objects based on simulation range data. During the conversion process, you can view simulation minimum and maximum values and proposed data types for these System objects. You can also view whole number information and histogram data. You cannot propose data types for these System objects based on static range data. This requires a DSP System Toolbox license. For more information, see [Convert a System object to Fixed-Point Using the Fixed-Point Converter App](#).

Simulation range collection and data type proposals for MATLAB Function blocks in Simulink

The Fixed-Point Tool can now collect and display simulation ranges for variables inside a MATLAB Function block. The tool can also propose data types for the variables based on the simulation data. You must manually apply the proposed data types to the variables. For more information, see [Convert Model Containing MATLAB Function Block to Fixed Point](#).

Overflow diagnostics to distinguish between wrap and saturation in Simulink

You can now separately control the diagnostics for overflows that wrap and overflows that saturate by setting each diagnostic to **error**, **warning**, or **none**. These controls simplify debugging models in which only one type overflow is of interest. For example, if you need to detect only overflows that wrap, in the **Data Validity** pane of the Configuration Parameters dialog box you can set **Wrap on overflow** to **error** or **warning**, and set **Saturate on overflow** to **none**.

Highlighting of potential data type issues in generated HTML report

You can now highlight potential data type issues in the generated HTML report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations. The expensive fixed-point operations check identifies optimization opportunities by highlighting expressions in the MATLAB code that require cumbersome multiplication or division, or expensive rounding.

For more information, see [Find Potential Data Type Issues in Generated Code](#)

Code generation of for loops using fixed-point loop indices

Fixed-point data types are now supported as for-loop indices in codegen. This capability requires a MATLAB Coder license. For more information, see [for](#).

Cast net slope computations using rational numbers

This new option improves the numerical accuracy and the readability of the C code generated for certain fixed-point conversions having nonbinary net slopes. Normally, net slope computation uses an integer multiplication followed by shifts. Enabling this optimization replaces the multiply and shift operation with a multiply and divide sequence that uses a rational number under certain simplicity and accuracy conditions.

For example, applying a net slope of 0.9, which traditionally would have generated

```
Vc = (int16_T)(Va * 115 >> 7);
```

becomes

```
Vc = (int16_T)(Va * 9/10);
```

This optimization affects both simulation and code generation. For more information, see [Handle Net Slope Computation](#).

Lock Column View option in the Fixed-Point Tool

This option prevents the Fixed-Point Tool from automatically changing the column view of the contents pane. To enable this option, in the Fixed-Point Tool menu, click **View > Lock Column View**. This setting is preserved across sessions.

Fixed-Point Advisor enhancements

- Improved support for interaction with Simulink data objects, including bus objects
- Block replacement recommendations for blocks with CORDIC support

hdlram renamed hdl.RAM

The `hdlram` System object has been renamed `hdl.RAM`. This System object no longer requires a Fixed-Point Designer license.

Compatibility Considerations

If you open a design that uses `hdlram`, the software displays a warning. For continued compatibility with future releases, replace instances of `hdlram` with `hdl.RAM`.

Changes to data type strings

Signal data type display

Signals using fixed-point data types with slope and bias scaling now always display the slope value in the data type name. In previous releases, the display decomposed the slope into slope adjustment factor and fixed exponent when it led to a more compact string. For example, the data type `fixdt(1,32,0.01953125,0)` now gets the name `sfix32_S0p01953125`. In previous releases, the name was in the decomposed format `sfix32_F1p25_en6`.

tostring function now uses 0 and 1 to represent signedness

The string representation of `numerictype` and `fixdt` objects returned by the `tostring` function now use 0 and 1 to represent signedness rather than `true` and `false`.

```
T = numerictype(true,16,15);  
T.tostring
```

```
ans =
```

```
numerictype(1,16,15)
```

When programmatically processing data types, best practice is to convert string representations to `numerictype` objects. The string changes for this release do not change the object that the strings are converted to. To convert a data type name string to an object, pass the string as the input argument to `fixdt` or `numerictype`. For example, `fixdt('sfix32_S0p01953125')` and `fixdt('sfix32_F1p25_En6')` return identical `numerictype` objects. To convert the results of the `tostring` function back to an object, use the `eval` function. For example, the `numerictype` objects returned by `eval('numerictype(1,16,15)')` and `eval('numerictype(true,16,15)')` are identical.

Compatibility Considerations

If your code converts data type strings to objects before doing any processing, then you will not have any compatibility issues related to the string changes. If you depend on the exact text returned by the `tostring` function or the exact text of a Simulink data type name, then you must modify your code to account for the changes described here. Alternatively, you can convert the string to a `numerictype` object before doing any additional processing.

New featured examples

The Fixed-Point Conversion Using Fixed-Point Tool and Derived Range Analysis example demonstrates using derived range analysis and the Fixed-Point Tool to convert a corner detection model to fixed point.

R2014a

Version: 4.2

New Features

Bug Fixes

Data type override and automatic data typing for bus objects

Data type override for bus objects

You can now apply data type override to models and subsystems that use virtual and non-virtual buses. The bus element types obey the data type override settings. This capability allows you to:

- Obtain the idealized floating-point behavior of models that use buses.
- Obtain the ideal derived ranges for models that use buses.
- Easily compare the idealized floating-point behavior with the fixed-point behavior of models that use buses.
- Use data type override to share fixed-point models that use buses with users who do not have a fixed-point license.

Autoscaling for bus objects

You can autoscale models that use virtual and non-virtual buses. This capability facilitates fixed-point conversion and optimization of models. The Fixed-Point Tool automatically proposes fixed-point data types for bus elements which removes the need to perform manual analysis and conversion of bus element data types.

For more information, see [Refine Data Types of a Model with Buses Using Simulation Data](#).

Derived ranges for complex signals in Simulink

Using the Fixed-Point Tool, you can now derive ranges for complex signals in Simulink. For more information, see [Conversion Using Range Analysis](#).

cordicsqrt function for fixed-point CORDIC-based square root functionality

The `cordicsqrt` function provides a CORDIC-based approximation of square root for use in fixed-point applications. For more information, see [cordicsqrt](#) and [Compute Square Root Using CORDIC](#).

Overflow detection with scaled double data types in MATLAB Coder projects

The MATLAB Coder Fixed-Point Conversion tool now provides the capability to detect overflows. At the numerical testing stage in the conversion process, the tool simulates the fixed-point code using scaled doubles. It then reports which expressions in the generated code produce values that would overflow the fixed-point data type. For more information, see [Detect Overflows Using the Fixed-Point Conversion Tool and Detecting Overflows](#).

You can also detect overflows when using the `codegen` function. For more information, see [`coder.FixptConfig`](#) and [Detect Overflows at the Command Line](#).

These capabilities require a MATLAB Coder license.

Fixed-point ARM Cortex-M code replacement support for DSP System Toolbox FIR filters

Fixed-point ARM Cortex[®]-M code replacement library support is now available for the Discrete FIR block and the `dsp.FIRFilter` System object.

These capabilities require a DSP System Toolbox license.

Fixed-Point Advisor support for referenced configuration sets

The Fixed-Point Advisor now supports referenced configuration sets. For more information, see [Preparing for Data Typing and Scaling](#).

Enhancements to automated conversion of MATLAB code

R2014a includes the following enhancements to the fixed-point conversion capability in MATLAB Coder projects.

These capabilities require a MATLAB Coder license.

Support for MATLAB classes

You can now use the MATLAB Coder Fixed-Point Conversion tool to convert floating-point MATLAB code that uses MATLAB classes. For more information, see [Fixed-Point Code for MATLAB Classes](#).

Generated fixed-point code enhancements

The generated fixed-point code now:

- Uses subscripted assignment (the colon(:) operator). This enhancement produces concise code that is more readable.
- Has better code for constant expressions. In previous releases, multiple parts of an expression were quantized to fixed point. The final value of the expression was less accurate and the code was less readable. Now, constant expressions are quantized only once at the end of the evaluation. This new behavior results in more accurate results and more readable code.

For more information, see [Generated Fixed-Point Code](#).

Fixed-point report

In R2014a, when you convert floating-point MATLAB code to fixed-point C/C++ code, the code generation software generates a fixed-point report in HTML format. For the variables in your MATLAB code, the report provides the proposed fixed-point types and the simulation or derived ranges used to propose those types. For a function, `my_fcn`, and code generation output folder, `out_folder`, the location of the report is `out_folder/my_fcn/fixpt/my_fcn_fixpt_Report.html`. If you do not specify `out_folder` in the project settings or as an option of the `codegen` command, the default output folder is `codegen`.

Automatic C compiler setup

In earlier releases, to set up a compiler before using `fiaccl` to accelerate MATLAB algorithms, you were required to run `mex -setup`. Now, the code generation software automatically locates and uses

a supported installed compiler. You can use `mex -setup` to change the default compiler. See [Changing Default Compiler](#).

More flexible control of `dsp.LMSFilter` System object fixed-point settings

For all `dsp.LMSFilter` System object fixed-point settings, you can now specify independent fixed-point data types.

This capability requires a DSP System Toolbox license.

Derived ranges for For Each and For Each Subsystem blocks

Range analysis supports For Each and For Each Subsystem blocks, with the following limitations:

- When For Each Subsystem contains another For Each Subsystem, not supported.
- When For Each Subsystem contains one or more Simulink Design Verifier™ Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported.

R2013b

Version: 4.1

New Features

Bug Fixes

Compatibility Considerations

C99 long long integer data type for embedded code generation

If your target hardware and your compiler support the C99 long long integer data type, you can use this data type for code generation. Using long long results in more efficient generated code that contains fewer cumbersome operations. Multi-line fixed-point helper functions can be replaced by simple expressions. This data type also provides more accurate simulation results for fixed-point and integer simulations. If you are using Microsoft® Windows (64-bit), using long long improves performance for many workflows including:

- Using Accelerator mode in Simulink
- Working with Stateflow® software
- Generating C code with Simulink Coder
- Accelerating fixed-point code using `fiaccel`
- Generating C code and MEX functions with MATLAB Coder

For more information about enabling long long in Simulink, see the **Enable long long** and **Number of bits: long long** configuration parameters on the Hardware Implementation Pane.

For more information about enabling long long for MATLAB Coder, see `coder.HardwareImplementation`.

Model Advisor fixed-point checks with additional coverage and optimization awareness

The Model Advisor fixed-point checks now cover additional blocks in base Simulink and System Toolboxes. The checks also now include the MATLAB Function block, System objects, Stateflow, and `fi` objects. These improved checks consider model settings such as hardware configuration and code generation settings. These updated checks also avoid false negative results.

These checks require an Embedded Coder license.

For more information, see:

- Identify blocks that generate expensive rounding code
- Identify questionable fixed-point operations
- Identify blocks that generate expensive fixed-point and saturation code

`fi` object as an index in colon expressions and an argument to `numel` and bit index functions

`fi` object as an index in colon expressions

You can now use `fi` objects in colon expressions. When you use `fi` in a colon expression, all colon operands must have integer values. See the `fi` and `colon` reference pages for examples.

`fi` objects as bit index input argument

The `bitget`, `bitset`, `bitsliceget`, `bitandreduce`, `bitorreduce`, and `bitxorreduce` functions now accept `fi` objects as the bit index argument.

fi objects as shift-value input argument

The `bitsra`, `bitsrl`, `bitsll`, `bitrol`, and `bitror` functions now accept `fi` objects as the shift-value input argument. You can use `fi` and built-in data type shift values interchangeably in MATLAB functions. This new capability facilitates fixed-point conversion.

numel function support for fi inputs

Effective R2013b, the `numel` function returns the number of elements in a `fi` array. Using `numel` in your MATLAB code returns the same result for built-in types and `fi` objects. Use `numel` to write data-type independent MATLAB code for array handling; you no longer need to use the `numberofelements` function.

The `numel` function is supported for simulation and code generation and with the MATLAB Function block in Simulink.

For more information, see `numel`.

Improved efficiency of data type internal rules for Lookup Table blocks

Blocks in the Lookup Tables library have a new internal rule for fixed-point data types to enable faster hardware instructions for intermediate calculations (with the exception of the Direct Lookup Table (n-D), Prelookup and Lookup Table Dynamic blocks). To use this new rule, select **Speed** for the **Internal Rule Priority** parameter in the dialog box. To use the R2013a internal rule, select **Precision**.

Derived ranges for complex variables in MATLAB Coder projects

Using the Fixed-Point Conversion tool in MATLAB Coder projects, you can now derive ranges for complex variables. For more information, see [Propose Data Types Based on Derived Ranges](#). This capability requires a MATLAB Coder license.

Simplified modeling of single-precision designs

Fixed-Point Designer now uses strict single-precision algorithms for operations between singles and integer or fixed-point data types. Operations, such as cast, multiplication and division, use single-precision math instead of introducing higher-precision doubles for intermediate calculations in simulation and code generation. You no longer have to explicitly cast integer or fixed-point inputs of these operations to single precision. To detect the presence of double data types in your model, use the Model Advisor **Identify questionable operations for strict single-precision design check**.

Compatibility Considerations

In R2013b, for both simulation and code generation, Fixed-Point Designer avoids the use of double data types to achieve strict single design for operations between singles and integers or fixed-point types. In previous releases, Fixed-Point Designer used double data types in intermediate calculations for higher precision. You might see a difference in numerical behavior of an operation between earlier releases and R2013b.

For example, when you cast from a fixed-point or integer data type to single or vice versa, the type used for intermediate calculations can significantly affect numerical results. Consider:

- Input type: `ufix128_En127`
- Input value: 1.99999999254942 — Stored integer value is $(2^{128} - 2^{100})$.
- Output type: `single`

Release	Calculation performed by Fixed-Point Designer	Output Result	Design Goal
R2013b	$Y = \text{single}(2^{-127}) * \text{single}(2^{128} - 2^{100})$ $= \text{single}(2^{-127}) * \text{Inf}$	Inf	Strict singles
Previous releases	$Y = \text{single}(\text{double}(2^{-127}) * \text{double}(2^{128} - 2^{100}))$ $= \text{single}(2^{-127} * 3.402823656532e+38)$	2	Higher-precision intermediate calculation

There is also a difference in the generated code. Previously, Fixed-Point Designer allowed the use of doubles in the generated code for a mixed multiplication that used single and integer types.

```
m_Y.Out1 = (real32_T)((real_T)m_U.In1*(real_T)m_U.In2);
```

In R2013b, it uses strict singles.

```
m_Y.Out1=(real32_T)m_U.In1*m_U.In2;
```

You can revert to the numerical behavior of previous releases, if necessary. To do so, insert explicit casting from integer and fixed-point data types to doubles for the inputs of these operations.

Range analysis support on Mac platforms

You can now perform derived range analysis of your model on Mac platforms. For more information, see [Conversion Using Range Analysis](#).

Changes to `showInstrumentationResults` function options

New option to suppress display of MATLAB code

When generating a printable instrumentation report, you can now choose to display only the tables that show information about logged variables. Used with the `-printable` option, the `-nocode` option suppresses display of the MATLAB code. Displaying only the logged variable information is useful for large projects with many lines of code.

Removal of `-browser` option

The `showInstrumentationResults` function `-browser` option has been removed. Use the `-printable` option instead. The `-printable` option creates a printable report and opens it in the system browser.

For more information, see `showInstrumentationResults`.

Changes to Continuous state-space block family range analysis support

The Continuous Simulink blocks State-Space, Transfer Fcn, and Zero-Pole are not supported and not stubbable for range analysis. For more information on blocks that are supported for range analysis, see Supported and Unsupported Simulink Blocks.

Compatibility Considerations

If you have a model that contains one or more continuous State-Space, Transfer Fcn, or Zero-Pole blocks, your model is incompatible with range analysis. Consider analyzing smaller portions of your model to work around this incompatibility.

Enhanced fiaccel support for int64 and uint64 functions

The `fiaccel` function now supports `int64` and `uint64` with `fi` inputs.

Support for LCC compiler on Microsoft Windows (64-bit) machines

If you are using Microsoft Windows (64-bit), LCC-64 is now available as the default compiler. You no longer have to install a separate compiler to perform fixed-point acceleration using `fiaccel`.

Warning for use of inexact `fi` and `fimath` property names

All `fi` and `fimath` property names are case sensitive and require that you use the full property names. Effective R2013b, if you use inexact property names, Fixed-Point Designer generates a warning.

Compatibility Considerations

To avoid seeing warnings for `fi` and `fimath` properties, update your code so that it uses the full names and correct cases of all these properties. The full names and correct cases of the properties appear when you display a `fi` or `fimath` object on the MATLAB command line.

Conversion of numeric variables into `Simulink.Parameter` objects

You can now convert a numeric variable into a `Simulink.Parameter` object using a single step.

```
% Define numerical variable in base workspace
myVar = 5;
%
% Create data object and assign variable value to data object value
myObject = Simulink.Parameter(myVar);
```

Previously, you did this conversion using two steps.

```
% Define numerical variable in base workspace
myVar = 5;
%
% Create data object
myObject = Simulink.Parameter;
%
% Assign variable value to data object value
myObject.Value = myVar;
```

Fixed-point conversion test file coverage results

The MATLAB Coder Fixed-Point Conversion tool now provides test file coverage results. After simulating your design using a test file, the tool provides an indication of how often the code is executed. If you run multiple test files at once, the tool provides the cumulative coverage. This information helps you determine the completeness of your test files and verify that they are exercising the full operating range of your algorithm. The completeness of the test file directly affects the quality of the proposed fixed-point types.

This capability requires a MATLAB Coder license.

For more information, see Code Coverage.

Fixed-point conversion workflow supports designs that use enumerated types

Using the Fixed-Point Conversion tool in MATLAB Coder projects, you can now propose data types for enumerated data types using derived and simulation ranges.

For more information, see Propose Fixed-Point Data Types Based on Derived Ranges and Propose Fixed-Point Data Types Based on Simulation Ranges. This capability requires a MATLAB Coder license.

Fixed-point conversion of variable-size data using simulation ranges

Using the Fixed-Point Conversion tool in MATLAB Coder projects, you can propose data types for variable-size data using simulation ranges.

For more information, see Propose Fixed-Point Data Types Based on Simulation Ranges. This capability requires a MATLAB Coder license.

Error checking improvements for `bitconcat`, `bitandreduce`, `bitorreduce`, `bitxorreduce`, `bitsliceget` functions

The `bitconcat`, `bitandreduce`, `bitorreduce`, `bitxorreduce`, and `bitsliceget` functions now check that all input arguments are real. If any inputs are complex, these functions generate an error.

The `bitconcat` function now generates an error in the unary syntax case, `bitconcat(a)`, if the input argument `a` is a scalar or is empty. To use `bitconcat` with one input argument, the argument must have more than one array element available for bit concatenation (that is, `length(a)>1`).

Legacy data type specification functions return numeric objects

In previous releases, the following functions returned a MATLAB structure describing a fixed-point data type:

- `float`
- `sfix`
- `sfrac`

- sint
- ufix
- ufrac
- uint

Effective R2013b, they return a Simulink.NumericType object. If you have existing models that use these functions as parameters to dialog boxes, the models continue to run as before and there is no need to change any model settings.

These functions do not offer full Data Type Assistant support. To benefit from this support, use fixdt instead.

Function	Return Value in Previous Releases – MATLAB structure	Return Value Effective R2013b – NumericType
float('double')	Class: 'DOUBLE'	DataTypeMode: 'Double'
float('single')	Class: 'SINGLE'	DataTypeMode: 'Single'
sfix(16)	Class: 'FIX' IsSigned: 1 MantBits: 16	DataTypeMode: 'Fixed-point: unspecified scaling' Signedness: 'Signed' WordLength: 16
ufix(7)	Class: 'FIX' IsSigned: 0 MantBits: 7	DataTypeMode: 'Fixed-point: unspecified scaling' Signedness: 'Unsigned' WordLength: 7
sfrac(33,5)	Class: 'FRAC' IsSigned: 1 MantBits: 33 GuardBits: 5	DataTypeMode: 'Fixed-point: binary point scaling' Signedness: 'Signed' WordLength: 33 FractionLength: 27
ufrac(44)	Class: 'FRAC' IsSigned: 0 MantBits: 44 GuardBits: 0	DataTypeMode: 'Fixed-point: binary point scaling' Signedness: 'Unsigned' WordLength: 44 FractionLength: 44
sint(55)	Class: 'INT' IsSigned: 1 MantBits: 55	DataTypeMode: 'Fixed-point: binary point scaling' Signedness: 'Signed' WordLength: 55 FractionLength: 0
uint(77)	Class: 'INT' IsSigned: 0 MantBits: 77	DataTypeMode: 'Fixed-point: binary point scaling' Signedness: 'Unsigned' WordLength: 77 FractionLength: 0

Compatibility Considerations

MATLAB Code

MATLAB code that depends on the return arguments of these functions being a structure with fields named Class, MantBits or GuardBits no longer works correctly. Change the code to access the appropriate properties of a NumericType object, for example, DataTypeMode, Signedness, WordLength, FractionLength, Slope and Bias.

C Code

Update C code that expects the data type of parameters to be a legacy structure to handle `NumericType` objects instead. For example, if you have S-functions that take legacy structures as parameters, update these S-functions to accept `NumericType` objects.

MAT-files

Effective R2013b, if you open a Simulink model that uses a MAT-file that contains a data type specification created using the legacy functions, the model uses the same data types and behaves in the same way as in previous releases but Simulink generates a warning. To eliminate the warning, recreate the data type specifications using `NumericType` objects and save the MAT-file.

You can use the `fixdtupdate` function to update a data type specified using the legacy structure to use a `NumericType`. For example, if you saved a data type specification in a MAT-file as follows in a previous release:

```
oldDataType = sfrac(16);  
save myDataTypeSpecification oldDataType
```

use `fixdtUpdate` to recreate the data type specification to use `NumericType`:

```
load DataTypeSpecification  
fixdtUpdate(oldDataType)
```

```
ans =
```

```
    NumericType with properties:
```

```
    DataTypeMode: 'Fixed-point: binary point scaling'  
    Signedness: 'Signed'  
    WordLength: 16  
    FractionLength: 15  
    IsAlias: 0  
    DataScope: 'Auto'  
    HeaderFile: ''  
    Description: ''
```

For more information, at the MATLAB command line, enter:

```
fixdtUpdate
```

numberofelements function being removed in a future release

The `numberofelements` function will be removed in a future release of Fixed-Point Designer software. Use `numel` instead.

R2013a

Version: 4.0

New Features

Bug Fixes

Compatibility Considerations

Product restructuring

The Fixed-Point Designer product replaces two pre-existing products: Fixed-Point Toolbox™ and Simulink Fixed Point™. You can access archived documentation for both products on the MathWorks® Web site.

Histogram logging in instrumented MATLAB Code Generation report

The `buildInstrumentedMex` and `showInstrumentationResults` instrumentation functions now can generate log2 histograms. A histogram is generated for each named and intermediate variable and for each expression in your code. The code generation report **Variables** tab includes a link to the histogram for each variable. You can use this histogram to determine the word and fraction lengths for your fixed-point values. Refer to the `buildInstrumentedMex` and `showInstrumentationResults` reference pages for information.

fi object in indexing and switch-case expressions

Effective this release, you can use `fi` objects as indices to arrays of built-in types and `fi` types. You can also use `fi` objects in switch-case expressions. These changes let you use `fi` objects without having to convert them. See the `fi` reference page for examples.

zeros, ones, and cast code reuse for floating-point and fixed-point types

The `zeros`, `ones`, and `cast` functions now work with fixed-point data types as well as built-in data types. The functions can now return an output whose class matches that of a specified numeric variable or `fi` object. For built-in data types, the output assumes the numeric data type, sparsity, and complexity (real or complex) of the specified numeric variable. For `fi` objects, the output assumes the `numericType`, complexity (real or complex), and `fimath` of the specified `fi` object.

For example:

```
>> a = fi([],1,24,12);
>> c = cast(pi,'like',a)
```

c =

```
3.1416
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 24
      FractionLength: 12
```

```
>> z = zeros(2,3,'like',a)
```

z =

```
0    0    0
0    0    0
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
```

```

        WordLength: 24
        FractionLength: 12

>> o = ones(2,3,'like',a)

o =

     1     1     1
     1     1     1

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 24
        FractionLength: 12

```

This capability allows you to cleanly separate algorithm code in MATLAB from data type specifications. Using separate data type specifications enables you to:

- Reuse your algorithm code with different data types.
- Switch easily between fixed-point and floating-point data types to compare fixed-point behavior to a floating-point baseline.
- Try different fixed-point data types to determine their effect on the behavior of your algorithm.
- Write clean, readable code.

For more information, see [Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros](#).

Code generation for $x.^n$ when n is a variable and x is a `fi` object

If the output type can be derived from the input settings, the `mpower` and `power` functions no longer require a constant exponent input. For more information, see [mpower](#) and [power](#).

Fixed-Point Advisor support for model reference

The Fixed-Point Advisor now performs checks on referenced models. It checks the entire model reference hierarchy against fixed-point guidelines. The Advisor also provides guidance about model configuration settings and unsupported blocks to help you prepare your model for conversion to fixed point.

Automated conversion of floating-point to fixed-point types in MATLAB Coder projects

You can now convert floating-point MATLAB code to fixed-point C code using the fixed-point conversion capability in MATLAB Coder projects. You can choose to propose data types based on simulation range data, static range data, or both.

Note You must have a MATLAB Coder license.

During fixed-point conversion, you can:

- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Validate that you can build your project with the proposed data types.
- Test numerics by running the test file with the fixed-point types applied.
- View a histogram of bits used by each variable.

For more information, see [Propose Fixed-Point Data Types Based on Simulation Ranges](#) and [Propose Fixed-Point Data Types Based on Derived Ranges](#).

Improved autoscaling for models with virtual bus signals

Autoscaling with the Fixed-Point Tool now handles data type constraints for virtual buses that do not have any associated bus objects. The data type proposals take into account the constraints introduced by these bus signals.

This improved autoscaling reduces data type mismatch errors. It also enables the Fixed-Point Tool to provide additional diagnostic information when you accept autoscaling proposals. For more information, see [Shared Data Type Summary](#).

Data Type Override for MATLAB Function block using built-in doubles and singles

The data type override rules for MATLAB Function block input signals and parameters have changed. If the input signals and parameters are `double` or `single`, and you specify data type override to be `Double` or `Single`, the overridden data types are now built-in `double` or built-in `single`, not `fi double` and `fi single` as in previous releases. If the input signals and parameters are `fi` objects or fixed-point signals, and you specify data type override to be `Double` or `Single`, the overridden data types are `fi double` and `fi single` as in previous releases. For more information, see [MATLAB Function Block with Data Type Override](#).

Compatibility Considerations

If you have MATLAB Function block code from previous releases that contains special cases for `fi double` or `fi single`, and you specify data type override to be `Double` or `Single`, you might have to update this code to handle built-in `double` and `single`.

Instrumentation for arrays of structs

The `buildInstrumentedMex` and `showInstrumentationResults` instrumentation functions now show instrumentation results for arrays of structs. Each field of each struct is logged and appears in the code generation report on the **Variables** tab.

File I/O function support

The following file I/O functions are now supported for code acceleration and generation:

-
- `fclose`
 - `fopen`
 - `fprintf`

To view implementation details, see [Functions Supported for Code Acceleration or Generation](#).

Support for nonpersistent handle objects

You can now accelerate code using `fiaccel` for local variables that contain references to handle objects or System objects. In previous releases, accelerating code for these objects was limited to objects assigned to persistent variables.

Load from MAT-files for code acceleration

`fiaccel` now supports a subset of the `load` function for loading run-time values from a MAT-file. It also provides a new function, `coder.load`, for loading compile-time constants. This support facilitates code generation from MATLAB code that uses `load` to load constants into a function. You no longer have to manually type in constants that were stored in a MAT-file.

To view implementation details for the `load` function, see [Functions Supported for Code Acceleration or Generation](#).

New toolbox functions supported for code acceleration and generation

To view implementation details, see [Functions Supported for Code Acceleration or Generation](#).

Bitwise Operation Functions

- `flintmax`

Computer Vision System Toolbox Classes and Functions

- `binaryFeatures`
- `insertMarker`
- `insertShape`

Data File and Management Functions

- `computer`
- `fclose`
- `fopen`
- `fprintf`
- `load`

Image Processing Toolbox Functions

- `conndef`
- `imcomplement`

- `imfill`
- `imhmax`
- `imhmin`
- `imreconstruct`
- `imregionalmax`
- `imregionalmin`
- `iptcheckconn`
- `padarray`

Interpolation and Computational Geometry

- `interp2`

MATLAB Desktop Environment Functions

- `ismac`
- `ispc`
- `isunix`

String Functions

- `strfind`
- `strrep`

Function to be removed in a future release

The `saveglobalfimathpref` will be removed in a future release.

Compatibility Considerations

Do not save `globalfimath` as a MATLAB preference. If you have previously saved `globalfimath` as a MATLAB preference, use `removeglobalfimathpref` to remove it.

Function being removed

The `emlmex` function has been removed.

Compatibility Considerations

The `emlmex` function generates an error in R2013a. Use `fiaccel` instead.