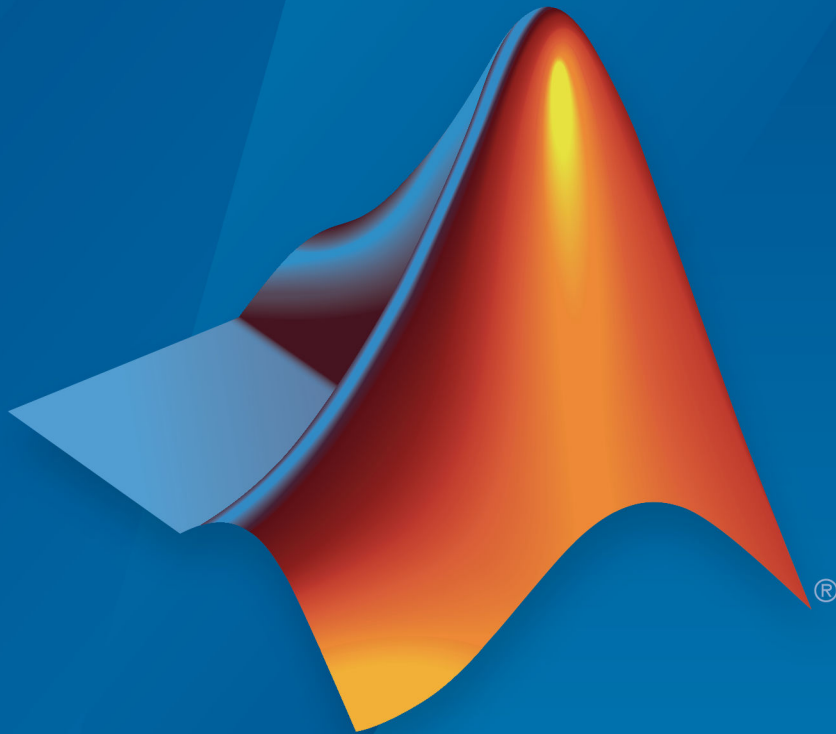


MATLAB® Coder™ Release Notes



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.
3 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Coder™ Release Notes

© COPYRIGHT 2011–2017 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.

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Fast Fourier Transforms: Generate code that takes advantage of the FFTW library	1-2
Strings: Generate code for MATLAB code that represents text as a string scalar	1-2
Statistics and Machine Learning Toolbox Code Generation: Generate C code for prediction by using discriminant analysis, k-nearest neighbor, SVM regression, regression tree ensemble, and Gaussian process regression models	1-2
Cell Arrays and Classes in Structures: Generate code for structures that contain cell arrays and classes	1-4
Class Folders: Generate code for MATLAB classes defined by using multiple files	1-4
Property Validation: Generate code for classes that restrict property values	1-4
Value Class Inputs: Pass objects of value classes to and from extrinsic functions and as constant inputs to entry-point functions	1-5
memcpy and memset for Variable-Size Arrays and Variable Number of Elements: Optimize code for more copies and assignments	1-5
Global Variables for Constant Values of Aggregate Types: Reduce memory usage in generated code	1-6

Reduction of Duplicate Functions and Types: Generate more compact code	1-7
App Support for Variable Number of Output Arguments: Specify the number of entry-point function output arguments to generate	1-7
Clear MEX in App: Reset the state of the Check for Run-Time Issues step	1-8
I/O Logging for Fixed-Point Conversion in App: Selectively log and plot function inputs and outputs at any level of your design	1-8
Code generation for more MATLAB functions	1-10
Characters and Strings	1-10
Data Type Conversion	1-11
Data Types	1-11
Fourier Analysis and Filtering	1-11
Moving Statistics	1-11
Preprocessing Data	1-12
Programming Utilities	1-12
Property Validation Functions	1-12
Code generation for more Audio System Toolbox System objects	1-13
Code generation for more Control System Toolbox objects	1-13
Code generation for more DSP System Toolbox System objects	1-13
Code generation for more Phased Array System Toolbox System objects and functions	1-13
Code generation for more Robotics System Toolbox functions	1-13
Code generation for more System Identification Toolbox objects	1-14

Code Generation for more WLAN System Toolbox functions	1-14
Check bug reports for issues and fixes	1-15

R2017a

Value Classes as Entry-Point Function Arguments: Generate code for more language constructs	2-2
Nested Functions: Generate code for more language constructs	2-3
Potential Differences Reporting: Identify MATLAB code that might behave differently in generated code	2-3
Automated Driving System Toolbox Code Generation: Generate code for sensor fusion and tracking workflow	2-3
Loop Invariant Code Motion: Generate optimized code for loops	2-5
Handle classes in value classes	2-5
Constant folding of value classes	2-6
Class properties and structure fields passed by reference to external C functions	2-6
Function specialization prevention with <code>coder.ignoreConst</code>	2-7
New <code>coder.unroll</code> syntax for more readable code	2-7
Size argument for <code>coder.opaque</code>	2-8
More flexible specification of number of entry-point function arguments	2-9

MEX function generation and testing in one step with codegen -test option	2-10
emxArray interface and utility files generated with single-file partitioning	2-10
Additional C and C++ Keywords in List of Reserved Keywords	2-11
More fixed-size variable information in Convert to Fixed- Point step of MATLAB Coder app	2-13
Code generation for more MATLAB functions	2-14
Code generation for more Audio System Toolbox System objects	2-14
Code generation for more Communications System Toolbox System objects	2-14
Code generation for more DSP System Toolbox System objects	2-14
Code generation for more Phased Array System Toolbox functions and System objects	2-14
Code generation for more Robotics System Toolbox functions and classes	2-15
Code generation for more Signal Processing Toolbox functions	2-16
Statistics and Machine Learning Toolbox Code Generation: Generate C code for prediction by using linear models, generalized linear models, decision trees, and ensembles of classification trees	2-16
Code generation for more WLAN System Toolbox functions and System objects	2-17
Check bug reports for issues and fixes	2-18

Recursive Functions and Anonymous Functions: Generate code for more MATLAB language constructs	3-2
Recursive Functions	3-2
Anonymous Functions	3-2
I/O Support: Generate code for fseek, ftell, fwrite	3-2
Statistics and Machine Learning Toolbox Code Generation: Generate code for prediction by using SVM and logistic regression models	3-3
Communications and DSP Code Generation: Generate code for more functions	3-3
Communications System Toolbox	3-3
DSP System Toolbox	3-4
Phased Array System Toolbox	3-4
WLAN System Toolbox	3-4
Wavelet Toolbox Code Generation: Generate code for discrete wavelet analysis, synthesis, and denoising functions	3-5
Variable-Size Cell Array Support: Use cell to create a variable-size cell array for code generation	3-5
Targeted Include Statements for coder.cinclude: Generate include statements only where indicated	3-5
Generated Code Readability: Generate more readable code for control flow	3-6
JIT MEX Compilation: Use JIT compilation to reduce code generation times for MEX	3-7
Change in default value for preserve variable names option	3-8
Code generation error for testing equality between enumeration and character array	3-9

Change to default standard math library for C++	3-10
Simplified type definition in the MATLAB Coder app	3-10
More discoverable build log and errors in MATLAB Coder app	3-11
Improved workflow for collecting and analyzing ranges in MATLAB Coder app	3-12
More discoverable logs and reports for fixed-point conversion in MATLAB Coder app	3-13
Hierarchical packaging of generated code in MATLAB Coder app	3-14
Code generation for additional MATLAB functions	3-14
Code generation for additional Audio System Toolbox functions	3-15
Code generation for additional Computer Vision System Toolbox functions	3-15
Code generation for additional Robotics System Toolbox functions	3-15
Code generation for extendedKalmanFilter and unscentedKalmanFilter with Control System Toolbox or System Identification Toolbox	3-16
Check bug reports for issues and fixes	3-17

R2016a

Cell Array Support: Use additional cell array features in MATLAB code for code generation	4-2
Use of {end + 1} to grow a cell array	4-2
Value and handle objects in cell arrays	4-2

Function handles in cell arrays	4-2
Non-Power-of-Two FFT Support: Generate code for fast Fourier transforms for non-power-of-two transform lengths	4-2
Faster Standalone Code for Linear Algebra: Generate code that takes advantage of your own target-specific LAPACK library	4-2
Computer Vision System Toolbox and Image Processing Toolbox Code Generation: Generate code for additional functions	4-3
MATLAB Coder Student Access: Obtain MATLAB Coder as student-use, add-on product or with MATLAB Primary and Secondary School Suite	4-3
Concatenation of Variable-Size Empty Arrays: Generate code for concatenation when a component array is empty	4-3
memset Optimization for More Cases: Optimize code that assigns a constant value to consecutive array elements ..	4-6
Optimization for Conditional and Boolean Expressions: Generate efficient code for more cases	4-7
MATLAB Coder App Line Execution Count: See how well test exercises MATLAB code	4-7
MATLAB Coder App Undo and Redo: Easily revert changes to type definitions	4-10
MATLAB Coder App Error Table: View complete error message	4-11
Changes to Fixed-Point Conversion Code Coverage	4-12
More Keyboard Shortcuts in Code Generation Report: Navigate the report more easily	4-13
xcorr Code Generation: Generate faster code for xcorr with long input vectors	4-14

Code generation for additional MATLAB functions	4-14
Specialized Math in MATLAB	4-14
Trigonometry in MATLAB	4-15
Interpolation and Computational Geometry in MATLAB . . .	4-15
Changes to code generation support for MATLAB functions	4-15
Code generation for Audio System Toolbox functions and System objects	4-15
Code generation for additional Communications System Toolbox functions	4-15
Code generation for additional DSP System Toolbox	4-16
Code generation for additional Phased Array System Toolbox functions	4-16
Code generation for additional Robotics System Toolbox functions	4-16
Code generation for WLAN System Toolbox functions and System objects	4-16
Check bug reports for issues and fixes	4-17

R2015aSP1

Bug Fixes

Check bug reports for issues and fixes	5-2
---	------------

Cell Array Support: Generate C code from MATLAB code that uses cell arrays	6-2
Faster MEX Functions for Linear Algebra: Generate MEX functions that take advantage of LAPACK	6-2
Double-Precision to Single-Precision Conversion: Convert double-precision MATLAB code to single-precision C code	6-2
Run-Time Checks in Standalone C Code: Detect and report run-time errors while testing generated standalone libraries and executables	6-3
Multicore Capable Functions: Generate OpenMP-enabled C code from more than twenty MATLAB mathematics functions	6-4
Image Processing Toolbox and Computer Vision System Toolbox Code Generation: Generate code for additional functions in these toolboxes	6-4
Image Processing Toolbox	6-4
Computer Vision System Toolbox	6-4
Statistics and Machine Learning Toolbox Code Generation: Generate code for kmeans and randsample	6-5
Simplified hardware specification in the MATLAB Coder app	6-5
MATLAB Coder app user interface improvements	6-7
Improvements for manual type definition	6-7
Tab completion for specifying files	6-8
Compatibility between the app colors and MATLAB preferences	6-8
Progress indicators for the Check for Run-Time Issues step ..	6-8
Saving and restoring of workflow state between MATLAB Coder app sessions	6-9

Project reuse between MATLAB Coder and HDL Coder	6-9
Code generation using freely available MinGW-w64 compiler	6-10
codegen debug option for libraries and executables	6-10
Code generation for additional MATLAB functions	6-11
Data Types in MATLAB	6-11
String Functions in MATLAB	6-11
Code generation for additional Communications System Toolbox, DSP System Toolbox, and Phased Array System Toolbox System objects	6-11
Communications System Toolbox	6-11
DSP System Toolbox	6-11
Phased Array System Toolbox	6-12
Code generation for Robotics System Toolbox functions and System objects	6-12
Code generation for System Identification Toolbox functions and System objects	6-12
Fixed-Point Conversion Enhancements	6-12
Saving and restoring fixed-point conversion workflow state in the app	6-12
Reuse of MEX files during fixed-point conversion using the app	6-13
Specification of additional fimath properties in app editor	6-13
Improved management of comparison plots	6-13
Variable specializations	6-14
Detection of multiword operations	6-15
Check bug reports for issues and fixes	6-2

Improved MATLAB Coder app with integrated editor and simplified workflow	7-2
Generation of example C/C++ main for integration of generated code into an application	7-3
Better preservation of MATLAB variable names in generated code	7-4
More efficient generated code for logical indexing	7-5
Code generation for additional Image Processing Toolbox and Computer Vision System Toolbox functions	7-5
Image Processing Toolbox	7-5
Computer Vision System Toolbox	7-5
Code generation for additional Communications System Toolbox, DSP System Toolbox, and Phased Array System Toolbox System objects	7-6
Communications System Toolbox	7-6
DSP System Toolbox	7-6
Phased Array System Toolbox	7-6
Code generation for additional Statistics and Machine Learning Toolbox functions	7-7
Code generation for additional MATLAB functions	7-7
Linear Algebra	7-7
Statistics in MATLAB	7-7
Code generation for additional MATLAB function options	7-8
Conversion from project to MATLAB script using MATLAB Coder app	7-8
Improved recognition of compile-time constants	7-8

Improved emxArray interface function generation	7-9
emxArray interface functions for variable-size arrays that external C/C++ functions use	7-9
Functions to initialize output emxArrays and emxArrays in structure outputs	7-10
External definition of a structure that contains emxArrays . .	7-10
Code generation for casts to and from types of variables declared using coder.opaque	7-11
Generation of reentrant code without an Embedded Coder license	7-12
Code generation for parfor-loops with stack overflow	7-12
Change in default value of the PassStructByReference code configuration object property	7-12
Change in GLOBALS variable in scripts generated from a project	7-13
Target build log display for command-line code generation when hyperlinks disabled	7-14
Removal of instrumented MEX output type	7-14
Truncation of long enumerated type value names that include the class name prefix	7-14
Fixed-point conversion enhancements	7-16
Support for multiple entry-point functions	7-16
Support for global variables	7-16
Code coverage-based translation	7-16
Generated fixed-point code enhancements	7-16
Automated fixed-point conversion of additional DSP System Toolbox objects	7-16
New interpolation method for generating lookup table MATLAB function replacements	7-17
Check bug reports for issues and fixes	7-18

Code generation for additional Image Processing Toolbox and Computer Vision System Toolbox functions	8-2
Image Processing Toolbox	8-2
Computer Vision System Toolbox	8-2
Code generation for additional Communications System Toolbox and DSP System Toolbox functions and System objects	8-2
Communications System Toolbox	8-2
DSP System Toolbox	8-2
Code generation for enumerated types based on built-in MATLAB integer types	8-3
Code generation for function handles in structures	8-4
Change in enumerated type value names in generated code	8-4
Code generation for ode23 and ode45 ordinary differential equation solvers	8-5
Code generation for additional MATLAB functions	8-5
Data and File Management in MATLAB	8-5
Linear Algebra in MATLAB	8-6
String Functions in MATLAB	8-6
Code generation for additional MATLAB function options	8-6
Collapsed list for inherited properties in code generation report	8-6
Change in length of exported identifiers	8-6
Intel Performance Primitives (IPP) platform-specific code replacement libraries for cross-platform code generation	8-7

Fixed-point conversion enhancements	8-8
Conversion from project to MATLAB scripts for command-line fixed-point conversion and code generation	8-8
Lookup table approximations for unsupported functions	8-8
Enhanced plotting capabilities	8-9
Automated fixed-point conversion for commonly used System objects in MATLAB including Biquad Filter, FIR Filter, and Rate converter	8-10
Additional fixed-point conversion command-line options	8-11
Type proposal report	8-11
Generated fixed-point code enhancements	8-11
Highlighting of potential data type issues in generated HTML report	8-12
 Check bug reports for issues and fixes	 8-15

R2014a

Code generation for additional Image Processing Toolbox and Computer Vision System Toolbox functions	9-2
Image Processing Toolbox	9-2
Computer Vision System Toolbox	9-2
 Code generation for additional Signal Processing Toolbox, Communications System Toolbox, and DSP System Toolbox functions and System objects	 9-2
Signal Processing Toolbox	9-2
Communications System Toolbox	9-3
DSP System Toolbox	9-3
 Code generation for fminsearch optimization function and additional interpolation functions in MATLAB	 9-3
Optimization Functions in MATLAB	9-3
Interpolation and Computational Geometry in MATLAB	9-3
 Conversion from project to MATLAB script for command-line code generation	 9-4
 Code generation for fread function	 9-4

Automatic C/C++ compiler setup	9-4
Compile-time declaration of constant global variables	9-5
Enhanced code generation support for switch statements ..	9-5
Code generation support for value classes with set.prop methods	9-6
Code generation error for property that uses AbortSet attribute	9-6
Independent configuration selections for standard math and code replacement libraries	9-6
Restrictions on bit length for integer types in a coder.HardwareImplementation object	9-9
Change in location of interface files in code generation report	9-9
Compiler warnings in code generation report	9-9
Removal of date and time comment from generated code files	9-10
Removal of two's complement guard from rtwtypes.h	9-10
Removal of TRUE and FALSE from rtwtypes.h	9-10
Change to default names for structure types generated from entry-point function inputs and outputs	9-10
Toolbox functions supported for code generation	9-11
Fixed-point conversion enhancements	9-13
Overflow detection with scaled double data types in MATLAB Coder projects	9-13
Support for MATLAB classes	9-13
Generated fixed-point code enhancements	9-14
Fixed-point report for float-to-fixed conversion	9-14
Check bug reports for issues and fixes	9-15

Code generation for Statistics Toolbox and Phased Array System Toolbox	10-2
Toolbox functions supported for code generation	10-2
parfor function for standalone code generation, enabling execution on multiple cores	10-3
Persistent variables in parfor-loops	10-3
Random number generator functions in parfor-loops	10-3
External code integration using coder.ExternalDependency	10-3
Updating build information using coder.updateBuildInfo	10-4
Generation of simplified code using built-in C types	10-4
Conversion of MATLAB expressions into C constants using coder.const	10-4
Highlighting of constant function arguments in the compilation report	10-4
Code Generation Support for int64, uint64 data types	10-5
C99 long long integer data type for code generation	10-5
Change to passing structures by reference	10-6
coder.runTest new syntax	10-6
coder.target syntax change	10-6
Changes for complex values with imaginary part equal to zero	10-7

Subfolder for code generation interface files	10-7
Support for LCC compiler on Windows 64-bit machines . . .	10-7
Fixed-Point conversion enhancements	10-8
Check bug reports for issues and fixes	10-10

R2013a

Automatic fixed-point conversion during code generation (with Fixed-Point Designer)	11-2
File I/O function support	11-2
Support for nonpersistent handle objects	11-3
Structures passed by reference to entry-point functions . . .	11-3
Include custom C header files from MATLAB code	11-3
Load from MAT-files	11-4
coder.opaque function enhancements	11-4
Automatic regeneration of MEX functions in projects	11-4
MEX function signatures include constant inputs	11-5
Custom toolchain registration	11-5
Complex trigonometric functions	11-6
parfor function reduction improvements and C support . . .	11-6
Support for integers in number theory functions	11-6
Enhanced support for class property initial values	11-7

Optimized generated code for $x=[x \ c]$ when x is a vector . . .	11-8
Default use of Basic Linear Algebra Subprograms (BLAS) libraries	11-9
Changes to compiler support	11-9
New toolbox functions supported for code generation	11-10
Functions being removed	11-11
Check bug reports for issues and fixes	11-12

R2012b

parfor function support for MEX code generation, enabling execution on multiple cores	12-2
Code generation readiness tool	12-2
Reduced data copies and lightweight run-time checks for generated MEX functions	12-2
Additional string function support for code generation . . .	12-2
Visualization functions in generated MEX functions	12-3
Input parameter type specification enhancements	12-3
Project import and export capability	12-4
Package generated code in zip file for relocation	12-5
Fixed-point instrumentation and data type proposals	12-5
New toolbox functions supported for code generation	12-5
New System objects supported for code generation	12-6

Check bug reports for issues and fixes	12-8
--	------

R2012a

Code Generation for MATLAB Classes	13-2
Dynamic Memory Allocation Based on Size	13-2
C/C++ Dynamic Library Generation	13-2
Automatic Definition of Input Parameter Types	13-2
Verification of MEX Functions	13-3
Enhanced Project Settings Dialog Box	13-3
Projects Infer Input Types from assert Statements in Source Code	13-4
Code Generation from MATLAB	13-4
New Demo	13-4
Check bug reports for issues and fixes	13-5

R2011b

Support for Deletion of Rows and Columns from Matrices	14-2
Code Generation from MATLAB	14-2
Check bug reports for issues and fixes	14-3

New User Interface for Managing Projects	15-2
To Get Started	15-2
Migrating from Real-Time Workshop emlc Function	15-2
New codegen Options	15-2
New Code Generation Configuration Objects	15-4
The codegen Function Has No Default Primary Function Input Type	15-5
The codegen Function Processes Compilation Options in a Different Order	15-5
New coder.Type Classes	15-5
New coder Package Functions	15-6
Script to Upgrade MATLAB Code to Use MATLAB Coder Syntax	15-6
Embedded MATLAB Now Called Code Generation from MATLAB	15-6
MATLAB Coder Uses rtwTargetInfo.m to Register Target Function Libraries	15-7
New Getting Started Tutorial Video	15-7
New Demos	15-7
Functionality Being Removed in a Future Version	15-8
Function Elements Being Removed in a Future Release ...	15-8
Check bug reports for issues and fixes	15-10

R2017b

Version: 3.4

New Features

Bug Fixes

Fast Fourier Transforms: Generate code that takes advantage of the FFTW library

In previous releases, when you generated code for the MATLAB fast Fourier transform (FFT) functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, and `ifftn`), the code generator produced code for the FFT algorithms.

In R2017b, to improve the execution speed of code generated for FFT functions, the code generator can produce calls to an FFT library. For MEX functions, the code generator uses the library that MATLAB uses. For standalone C/C++ code (static library, dynamically linked library, or executable program), to generate calls to a specific installed FFTW library, provide an FFT library callback class. See “Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”.

For more information about FFTW, see www.fftw.org.

In R2017b, for MEX functions, you can generate code for the MATLAB `fftw` function. For standalone code, to specify a planning method, implement a `getPlanMethod` method in an FFT library callback class.

Strings: Generate code for MATLAB code that represents text as a string scalar

In previous releases, in MATLAB code for code generation, you represented text as a character vector. For example:

```
c = 'Hello World';
```

In R2017b, you can represent text as a string scalar (a 1-by-1 MATLAB string array). For example:

```
s = "Hello World";
```

Code generation does not support string arrays that have more than one element.

See “Code Generation for Strings”.

Statistics and Machine Learning Toolbox Code Generation: Generate C code for prediction by using discriminant analysis, k-nearest neighbor,

SVM regression, regression tree ensemble, and Gaussian process regression models

You can generate code for these Statistics and Machine Learning Toolbox™ functions:

- `predict (CompactClassificationDiscriminant)` — Classify observations or estimate classification scores and costs by applying a discriminant analysis classification to new data.
- `predict (ClassificationKNN)` — Classify observations or estimate classification scores and costs by applying k -nearest neighbor classification, based on an exhaustive search, to new data.
- `predict (CompactRegressionSVM)` — Predict responses by applying a support vector machine (SVM) regression to new data.
- `predict (CompactRegressionEnsemble)` — Predict responses by applying ensembles of regression trees to new data.
- `predict (RegressionLinear)` — Predict responses by applying a linear regression to new data.
- `predict (CompactRegressionGP)` — Predict responses or estimate confidence intervals on predictions by applying a Gaussian process regression to new data.
- `knnsearch (ExhaustiveSearcher)` and `knnsearch` — Identify the k -nearest neighbors using the exhaustive search algorithm.
- `rangesearch (ExhaustiveSearcher)` and `rangesearch` — Identify all neighbors within a specified distance using the exhaustive search algorithm.
- `pdist2` — Compute the pairwise distance between two sets of observations.

When you train an SVM model by using `fitsvm` for code generation, you can now specify a score transformation function by using the 'ScoreTransform' name-value pair argument or by assigning the `ScoreTransform` object property. Therefore, `saveCompactModel` can accept compact SVM models equipped to estimate class posterior probabilities, that is, models returned by `fitposterior` or `fitSVMPosterior`. Also, you can now implement one-class learning.

When you train a linear classification model by using `fitcllinear` for code generation, you can now specify either 'svm' or 'logistic' for the 'Learner' name-value pair argument.

Cell Arrays and Classes in Structures: Generate code for structures that contain cell arrays and classes

In previous releases, for code generation, you could not assign a cell array or object to a structure field. In R2017b, structures can contain cell arrays and classes. For example, you can now generate code for the function `assignToStruct`:

```
function result = assignToStruct(in1)
%#codegen
x = MyClass;
x.prop = in1;
y.val = x;           % object in struct
y.val2 = {1,2,3};    % cell in struct
result = y.val.prop;
end
```

Class Folders: Generate code for MATLAB classes defined by using multiple files

You can generate code for MATLAB code that uses a class that is defined in a class folder. When you define a class in a class folder, you can put the class definition in one file and the methods in other, separate files. The class folder name consists of the @ character followed by the class name. For example, the class folder `@MyClass` contains the class definition file `MyClass.m`. The folder can also contain separate files for the methods. For more information about class folders, see “Folders Containing Class Definitions” (MATLAB).

Property Validation: Generate code for classes that restrict property values

You can generate code for classes that restrict property values according to size, class, and other criteria. To establish criteria that a property value must conform to, use MATLAB validation functions or write your own validation functions. For information about property validation, see “Validate Property Values” (MATLAB).

MEX functions report errors that result from property validation. Standalone C/C++ code reports these errors only if you enable run-time error reporting. See “Run-Time Error Detection and Reporting in Standalone C/C++ Code”. Before you generate standalone C/C

++ code, it is a best practice to test property validation by running a MEX function over the full range of input values.

Value Class Inputs: Pass objects of value classes to and from extrinsic functions and as constant inputs to entry-point functions

In R2017b, you can now use value class inputs in these ways:

- Pass an object of a value class as an input to or output from an extrinsic function.
- Specify that an object of a value class is a constant entry-point function input argument.

If you use `codegen`, to specify that an object is constant, use `coder.Constant`. See “Specify Objects as Inputs at the Command Line”. In the MATLAB Coder app, to specify that an object is constant, see “Specify Objects as Inputs in the MATLAB Coder App”.

`memcpy` and `memset` for Variable-Size Arrays and Variable Number of Elements: Optimize code for more copies and assignments

By using the `memcpy` and `memset` optimizations, the code generator can produce faster, more compact, and more readable code. In previous releases, the code generator used these optimizations only for fixed-size arrays, when the number of array elements to copy or assign was known at compile time. In R2017b, the code generator can use these optimizations for:

- Variable-size arrays.
- A variable number of elements (the number of elements to copy or assign is determined at run time).

From a previous release, here is an example of generated C code that copies a variable number of elements without the `memcpy` optimization:

```
for (i0 = 0; i0 <= loop_ub; i0++) {
    Y[i0] = 1.0;
}
```

From R2017b, here is the equivalent C code that copies a variable number of elements with the `memcpy` optimization:

```
memcpy(&Y[0], &tmp_data[0], (unsigned int)(loop_ub * (int)sizeof(double)));
```

When the number of elements to copy or assign is unknown at compile time:

- The code generator invokes the optimizations without regard to the `memcpy/memset` threshold parameter.
- The code generator does not use the optimizations in code generated for copies or assignments inside a MATLAB `for`-loop. For example, the code generator does not use the `memcpy` optimization for MATLAB code such as:

```
for i = 1:n
    Y(i) = X(i);
end
```

The code generator tries to use the `memcpy` optimization for MATLAB code such as:

```
Y(1:n) = X(1:n)
```

For more information, see “`memcpy` Optimization” and “`memset` Optimization”.

Global Variables for Constant Values of Aggregate Types: Reduce memory usage in generated code

In R2017b, to reduce memory usage, the code generator identifies opportunities for functions in the generated code to use global variables for assignment of constant values from aggregate types. Aggregate types include arrays and structures. If the code generator detects that large variables in multiple functions would have the same aggregate type and constant values, then it produces a global variable that contains the constant values. The functions assign values from the global variable, instead of creating a local copy of the values. For example, in this code, functions `f` and `g` assign values from the global variable `iv0` to the local variables `m1` and `m2`.

```
extern const int32_T iv0[5];
const int32_T iv0[5] = { 1,2,3,4,5 };
void f(void)
{
    int32_T m1;
    int32_T m2;
    m1 = iv0[1];
    m2 = iv0[1];
}
```

```

void g(void)
{
    int32_T m1;
    m1 = iv0[1];
}

```

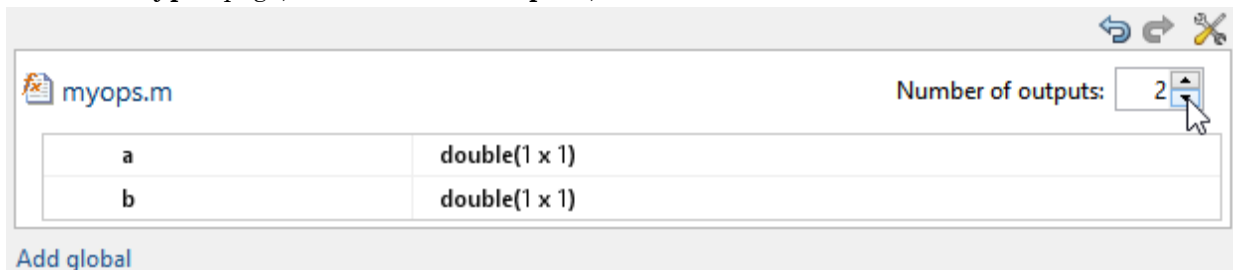
Reduction of Duplicate Functions and Types: Generate more compact code

In previous releases, the code generator could produce duplicate functions and types with the same syntactic content. Duplication causes an increase in code size and compilation time.

In R2017b, the code generator can find and merge duplicate types and functions. If you define two identical functions, the code generator does not merge them.

App Support for Variable Number of Output Arguments: Specify the number of entry-point function output arguments to generate

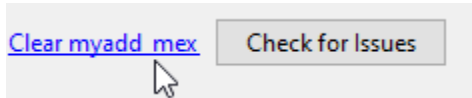
In R2017a, when you generated code with `codegen`, you could use the `-nargout` option to specify the number of entry-point function output arguments to generate. In R2017b, you can also specify the number of entry-point function output arguments in the MATLAB Coder app. To specify the number of outputs when a function returns `varargout`, or to generate fewer outputs than the function defines, on the **Define Input Types** page, in **Number of outputs**, select the number.



See “Specify Number of Entry-Point Function Input or Output Arguments to Generate”.

Clear MEX in App: Reset the state of the Check for Run-Time Issues step

In the MATLAB Coder app, after you check for run-time issues, you can clear the generated MEX function from memory. Next to the **Check for Issues** button, click the hyperlink.



Clearing the MEX function resets data, such as persistent variables or line execution counts, that the **Check for Run-Time Issues** step accumulates.

I/O Logging for Fixed-Point Conversion in App: Selectively log and plot function inputs and outputs at any level of your design

You can now elect to log and plot all function inputs and outputs during the **Test** phase of fixed-point conversion in the MATLAB Coder app. In previous releases, you could log only top-level function inputs and outputs.

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 that 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 then select the variables that you want to log.

The screenshot shows the 'Convert to Fixed Point' application. The main window displays MATLAB code for a Kalman filter. Below the code is a table with columns: Variable, Type, Sim Min, Sim Max, Whole..., Proposed Type, Log Data, and Max Diff. The 'Log Data' column has checkboxes for variables u, y, and x, which are currently checked. A red box highlights the 'Log Data' column, and a mouse cursor is clicking the checkbox for variable 'y'.

Variable	Type	Sim Min	Sim Max	Whole ...	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"/>	

After you select the variables that you want to log, click **Test**.

The app runs a floating-point and fixed-point simulation. Then, it generates comparison plots and calculates the difference error for all logged variables.

Convert to Fixed Point

SETTINGS ANALYZE CONVERT TEST

Source Code

kalman_filter_tb.m

```


1 function [y, tmp] = 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    p_prd = A * p_est * A' + Q;
25
26    % Estimation

```

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.m
kalman_filter_fixpt_log.txt

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

To open the comparison plot, click the  icon in the **Max Diff** column.

Code generation for more MATLAB functions

Characters and Strings

- contains
- convertCharsToStrings
- convertStringsToChars
- count
- endsWith

- erase
- eraseBetween
- extractAfter
- extractBefore
- insertAfter
- insertBefore
- isstring
- replace
- replaceBetween
- reverse
- startsWith
- string
- strip
- strlen

Data Type Conversion

- int2str

Data Types

- enumeration

Fourier Analysis and Filtering

- fftw

Moving Statistics

- movmad
- movmax
- movmean
- movmedian
- movmin
- movprod

- `movstd`
- `movsum`
- `movvar`

Preprocessing Data

- `isoutlier`
- `filloutliers`

Programming Utilities

- `builtin`

Property Validation Functions

- `mustBeFinite`
- `mustBeGreaterThan`
- `mustBeGreaterThanOrEqual`
- `mustBeInteger`
- `mustBeLessThan`
- `mustBeLessThanOrEqual`
- `mustBeMember`
- `mustBeNegative`
- `mustBeNonempty`
- `mustBeNonNan`
- `mustBeNonnegative`
- `mustBeNonpositive`
- `mustBeNonsparse`
- `mustBeNonzero`
- `mustBeNumeric`
- `mustBeNumericOrLogical`
- `mustBePositive`
- `mustBeReal`

Code generation for more Audio System Toolbox System objects

- `graphicEQ`

Code generation for more Control System Toolbox objects

- `particleFilter`

Code generation for more DSP System Toolbox System objects

- `dsp.BlockLMSFilter`
- `dsp.FrequencyDomainFIRFilter`
- `dsp.ZoomFFT`

Code generation for more Phased Array System Toolbox System objects and functions

- `phased.HeterogeneousConformalArray`
- `phased.HeterogeneousULA`
- `phased.HeterogeneousURA`
- `phased.UnderwaterRadiatedNoise`
- `range2t1`
- `sonareqsl`
- `sonareqsnr`
- `sonareqtl`
- `t12range`

Code generation for more Robotics System Toolbox functions

- `lidarScan`
- `matchScans`

Code generation for more System Identification Toolbox objects

- `particleFilter`

Code Generation for more WLAN System Toolbox functions

- `wlanBCCDecode`
- `wlanBCCEncode`
- `wlanBCCDeinterleave`
- `wlanBCCInterleave`
- `wlanConstellationDemap`
- `wlanConstellationMap`
- `wlanDMGDataBitRecover`
- `wlanDMGHeaderBitRecover`
- `wlanScramble`
- `wlanGolaySequence`
- `wlanSegmentDeparseBits`
- `wlanSegmentDeparseSymbols`
- `wlanSegmentParseBits`
- `wlanSegmentParseSymbols`
- `wlanStreamDeparse`
- `wlanStreamParse`

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2017a

Version: 3.3

New Features

Bug Fixes

Compatibility Considerations

Value Classes as Entry-Point Function Arguments: Generate code for more language constructs

In R2017a, for code generation, an object of a value class can be an entry-point function argument. An entry-point function is a top-level function that you call from or from external C code.

For example, suppose that you define a value class `mySquare` and a function `getarea` that has an input argument that is a value class object.

```
classdef mySquare
    properties
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                if isnumeric(val)
                    obj.side = val;
                else
                    error('Value must be numeric')
                end
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end

function z = getarea(s)
    %#codegen
    z = calcarea(s);
end
```

In R2017a, you can generate code for `getarea`. When you generate code, specify that the input argument `s` has the type of an object of the value class `mySquare`.

See [Specify Objects as Inputs at the Command Line](#) and [Specify Objects as Inputs in the MATLAB Coder™ App](#).

Nested Functions: Generate code for more language constructs

In R2017a, you can generate code for nested functions. For code generation, when you use nested functions, adhere to these restrictions:

- If the parent function declares a persistent variable, it must assign the persistent variable before it calls a nested function that uses the persistent variable.
- A nested recursive function cannot refer to a variable that the parent function uses.
- If a nested function refers to a structure variable, you must define the structure by using `struct`.
- If a nested function uses a variable defined by the parent function, you cannot use `coder.ysize` with the variable in either the parent or the nested function.

Also, you must adhere to the code generation restrictions for value classes and handle classes.

Potential Differences Reporting: Identify MATLAB code that might behave differently in generated code

Generation of efficient C/C++ code sometimes results in behavior differences between the generated code and the original MATLAB code. In R2017a, the code generator detects and reports some of these differences as potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

When potential differences reporting is enabled, the code generation report and the MATLAB Coder app list potential differences messages on the **Potential Differences** tab. To highlight the MATLAB code that corresponds to the message, click the message.

Reviewing and addressing potential differences before you deploy code helps you to avoid errors and incorrect answers.

See Potential Differences Reporting and Potential Differences Messages.

Automated Driving System Toolbox Code Generation: Generate code for sensor fusion and tracking workflow

You can generate code for these Automated Driving System Toolbox™ tracking and sensor fusion functions and classes.

cameas
cameasjac
constacc
constaccjac
constturn
constturnjac
constvel
constveljac
ctmeas
ctmeasjac
cvmeas
cvmeasjac
getTrackPositions
getTrackVelocities
initcaekf
initcakf
initcaukf
initctekf
initctukf
initcvekf
initcvkf
initcvukf
multiObjectTracker
objectDetection
trackingEKF
trackingKF
trackingUKF

For C/C++ code generation usage notes and limitations, see the function or class reference page.

Loop Invariant Code Motion: Generate optimized code for loops

In R2017a, MATLAB Coder uses loop invariant code motion to optimize `for`-loops and `while`-loops in generated C code. Invariant code is code that does not change inside a loop. The loop invariant code motion optimization moves invariant code outside of a loop so that it executes only once before the loop instead of with each loop iteration.

Here is an example of a `for`-loop in C code generated in a previous release:

```
for (k = 0; k < 64; k++) {
    *offset = offsetFactor * params[4];
    outData[k] = (double)mask[k] * (*offset + inData[k]);
}
```

Here is the C code generated in R2017a:

```
*offset = offsetFactor * params[4];
for (k = 0; k < 64; k++) {
    outData[k] = (double)mask[k] * (*offset + inData[k]);
}
```

In R2017a, the loop invariant code motion optimization moves the invariant code outside of the loop.

Handle classes in value classes

In R2017a, you can generate code for value classes that contain handle classes. The handle class can be one that you define or a predefined handle class that is available with MATLAB or a MATLAB toolbox. Predefined handle classes, such as toolbox System objects, must be supported for C/C++ code generation. See [Functions and Objects Supported for C/C++ Code Generation — Category List](#).

For example, suppose that `myclass` is a value class and `myhandle` is a handle class. You can generate C/C++ code for MATLAB code such as:

```
obj = myclass;
obj.p1 = myhandle;
obj.p2 = dsp.Mean;
```

The code generation limitations for handle class objects apply to handle class objects in value classes. See [Handle Object Limitations for Code Generation](#).

Constant folding of value classes

In R2017a, you can use `coder.const` to constant-fold value classes.

The code generator tries to fold constant expressions into the generated code. Constant folding uses the value of a constant expression instead of the expression in the generated code. Constant folding can improve execution time because the generated code does not have to evaluate the expression multiple times. You can try to force the code generator to constant-fold an expression by using `coder.const`.

To constant-fold a value class object `obj`, use this syntax:

```
coder.const(obj)
```

To constant-fold the property `prop`, use this syntax:

```
coder.const(obj.prop)
```

You cannot constant-fold a value class object that is an entry-point function input argument.

Class properties and structure fields passed by reference to external C functions

To pass arguments by reference to an external C function, you use `coder.ref`, `coder.rref`, or `coder.wref` in a `coder.ceval` call. For example:

```
...  
x = 1;  
y = coder.ceval('myCFunction', coder.ref(x));  
...
```

In previous releases, to pass a class property or structure field by reference using `coder.ref`, `coder.rref`, or `coder.wref`, you had to first assign the property or field to a variable. For example:

```
...  
x = myClass;  
x.prop = 1;  
v = x.prop;  
coder.ceval('foo', coder.ref(v));  
...
```

In R2017a, you can directly pass a class property or structure field by reference. For example:

- Pass a class property

```
...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.ref(x.prop));
...
```

- Pass a structure field

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.wref(s.s1.a));
...
```

- Pass a field of an element of an array of structures

```
...
s = struct('c', [1 2], 'd', 2);
s1 = struct('a', [s s]);
coder.ceval('foo', coder.rref(s1.a(1).d));
...
```

Function specialization prevention with `coder.ignoreConst`

At compile time, if an input argument to a function call evaluates to a constant, the code generator can use the constant value to produce function specializations. A function specialization is a version of a function in which the input type, size, complexity, or value is customized for a particular invocation of the function. To prevent function specializations due to constant arguments, instruct the code generator to treat the value of the argument as a nonconstant value by using `coder.ignoreConst`.

With compile-time recursion, the code generator produces function specializations instead of a recursive call. If the specializations are due to a constant input argument to the recursive function, you might be able to force run-time recursion by using `coder.ignoreConst`. See [Force Code Generator to Use Run-Time Recursion](#).

New `coder.unroll` syntax for more readable code

In R2017a, `coder.unroll` has a new syntax that helps make your code more readable.

In previous releases, you put `coder.unroll` inside a `for`-loop. For example:

```
...
for i = coder.unroll(1:n)
    y(i) = rand();
end
...
```

With the new syntax, you put `coder.unroll` on a line by itself, immediately before the loop that it unrolls. For example:

```
...
coder.unroll();
for i = 1:n
    y(i) = rand();
end
...
```

Here is an example of the new syntax with the `flag` argument:

```
...
unrollflag = n < 10;
coder.unroll(unrollflag);
for i = 1:n
    y(i) = rand();
end
...
```

Both the new syntaxes and the syntaxes from previous releases are supported. For more readable code, use the new syntax.

For more information about `coder.unroll` and `for`-loop unrolling, see `coder.unroll` and `Unroll for-Loops`.

Size argument for `coder.opaque`

In R2017a, you can specify the size of a variable that you declare with `coder.opaque`. The syntax with the size argument is:

```
x = coder.opaque(type,value,'Size', size)
```

Specify the size in bytes. For example, declare `x1` to be a 4-byte integer with initial value 0.


```
x1 = coder.opaque('int','0', 'Size', 4);
```

More flexible specification of number of entry-point function arguments

In R2017a, you can generate a MEX or a C/C++ function that has a different number of input or output arguments than the original MATLAB function definition specifies.

Consider this function:

```
function [x, y] = myops(a,b)
%#codegen
if (nargin > 1)
    x = a + b;
    y = a * b;
else
    x = a;
    y = -a;
end
```

To generate a function that takes only one argument, provide one argument with `-args`.

```
codegen myops -args {3} -report
```

To generate a function that returns only one argument, use the `-nargout` option of the `codegen` command.

```
codegen myops -args {2 3} -nargout 1 -report
```

You can also use `-nargout` to specify the number of arguments for a function that uses `varargout`.

Rewrite `myops` to use `varargout`.

```
function varargout = myops(a,b)
%#codegen
if (nargin > 1)
    varargout{1} = a + b;
    varargout{2} = a * b;
else
    varargout{1} = a;
    varargout{2} = -a;
end
```

Generate code for one output argument.

```
codegen myops -args {2 3} -nargout 1 -report
```

See Specify Number of Entry-Point Function Input or Output Arguments to Generate.

MEX function generation and testing in one step with `codegen -test` option

In R2017a, you can generate a MEX function and test it in one step by using the `codegen -test` option. Provide a test file that calls the original MATLAB function. For example:

```
codegen myfunction -test myfunction_test
```

Before you generate standalone C/C++ code for your MATLAB code, it is a best practice to generate a MEX function from your entry-point functions. Running the MEX function helps you to detect and fix run-time errors that are much harder to diagnose in the generated code. It also helps you to verify that the MEX function provides the same functionality as the original MATLAB code. It is also a best practice to write a test file that calls your original MATLAB functions. If you have a test file, you can use `coder.runTest` to run the test file, replacing the call to the original MATLAB function with a call to the MEX function. By using the `codegen -test` option, you combine MEX generation and testing in one step instead of generating the MEX function, and then calling `coder.runTest`.

The `-test` option is supported only when generating MEX functions or when using a configuration object with `VerificationMode` set to `'SIL'`. Creation of a configuration object that has the `VerificationMode` parameter requires the Embedded Coder® product.

This option is not supported with fixed-point conversion or single-precision conversion.

See Verify MEX Functions at the Command Line.

emxArray interface and utility files generated with single-file partitioning

When the code generator uses dynamic memory allocation for variable-size arrays, it produces utility functions that the generated code uses. For a function `foo`, these functions are in `foo_util.c`. The declarations are in `foo_util.h`. If the variable-size arrays are entry-point function inputs or outputs, the code generator produces functions

for interfacing with `emxArrays` in the generated code. These interface functions are in `foo_emxAPI.c`. The declarations are in `foo_emxAPI.h`.

In previous releases, if you chose to generate all C/C++ functions into a single file, the code generator included these utility and `emxArray` interface functions, and their declarations, in that file. It did not put the functions and declarations in separate files. In R2017a, the code generator always produces separate files for these functions and their declarations, even if you choose single-file partitioning. For example, it produces `foo_util.c`, `foo_util.h`, `foo_emxAPI.c`, and `foo_emxAPI.h`.

Compatibility Considerations

In previous releases, if you chose to generate all C/C++ functions into a single file, you did not have to include the header file for the `emxArray` interface functions in your C main file. In R2017a, regardless of the file partitioning method, you must include this header file in your C main file. For example, if the code generator produces `foo_emxAPI.c` and `foo_emxAPI.h`, include `foo_emxAPI.h` in your C main file.

If you use MATLAB Coder to package your files, the packaging software includes the files generated for the utility and `emxArray` interface functions. If you manually package the generated files, include the utility and interface function files with the other files.

For information about `emxArray` interface functions, see [C Code Interface for Arrays](#). For information about changing the file partitioning method, see [How MATLAB Coder™ Partitions Generated Code](#). For information about packaging files, see [Package Code for Other Development Environments](#).

Additional C and C++ Keywords in List of Reserved Keywords

If your MATLAB code uses C or C++ reserved keywords for function or variable names, the code generator tries to rename the generated identifiers. If renaming is not possible, then the code generator produces an error. For example, if you use a reserved keyword for an entry-point function name, the code generator produces an error.

In R2017a, the list of C and C++ reserved keywords contains additional keywords.

Here are the additional C reserved keywords.

<code>assert</code>	<code>limits</code>	<code>stdatomic</code>	<code>string</code>
---------------------	---------------------	------------------------	---------------------

complex	locale	stdbool	tgmath
ctype	math	stddef	threads
errno	setjmp	stdint	time
fenv	signal	stdio	uchar
float	stdalign	stdlib	wchar
inttypes	stdarg	stdnoreturn	wctype
iso646			

Here are the additional C++ reserved keywords.

algorithm	csignal	future	ratio
any	cstdalign	initializer_list	regex
array	cstdarg	iomanip	scoped_allocator
atomic	cstdbool	ios	set
bitset	cstddef	iosfwd	shared_mutex
cassert	cstdint	iostream	sstream
ccomplex	cstdio	istream	stack
cctype	cstdlib	iterator	stdexcept
cerrno	cstring	limits	streambuf
cfenv	ctgmath	list	string_view
cfloat	ctime	locale	stringstream
chrono	cuchar	map	system_error
cinttypes	wchar	memory	thread
ciso646	cwctype	memory_resource	tuple
climits	deque	mutex	type_traits
clocale	exception	new	typeid
cmath	execution	numeric	typeid
codecvt	filesystem	optional	unordered_map
complex	forward_list	ostream	unordered_set
condition_variable	fstream	queue	utility

csetjmp	functional	random	valarray
---------	------------	--------	----------

Compatibility Considerations

If your MATLAB code uses any of the additional C or C++ reserved keywords, in R2017a, code generation might result in an error.

More fixed-size variable information in Convert to Fixed-Point step of MATLAB Coder app

In R2017a, in the MATLAB Coder 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(x(i)), FIMATH);
    z(1) = fi_signed(b(2)*x(i) + z
    z(2) = Type: 1 x 256 embedded.fi (i)
end
id
Function Repl

```

Type: 1 x 256 embedded.fi (i)

Signedness: Signed

Word Length: 16

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.

Code generation for more MATLAB functions

- `cholupdate`
- `histcounts`
- `ismethod`

For C/C++ code generation usage notes and limitations, see the function reference page.

Code generation for more Audio System Toolbox System objects

`audioPlayerRecorder`

For C/C++ code generation usage notes and limitations, see the reference page.

Code generation for more Communications System Toolbox System objects

`comm.RBDSWaveformGenerator`

For C/C++ code generation usage notes and limitations, see the reference page.

Code generation for more DSP System Toolbox System objects

- `dsp.HampelFilter`
- `dsp.AsyncBuffer`

For C/C++ code generation usage notes and limitations, see the System object™ reference page.

Code generation for more Phased Array System Toolbox functions and System objects

- `bw2range`

- `diagbfweights`
- `scatteringchanmtx`
- `waterfill`
- `phased.BackScatterSonarTarget`
- `phased.DopplerEstimator`
- `phased.IsoSpeedUnderWaterPaths`
- `phased.IsotropicHydrophone`
- `phased.IsotropicProjector`
- `phased.MultipathChannel`
- `phased.RangeEstimator`
- `phased.RangeResponse`
- `phased.ScatteringMIMOChannel`

For C/C++ code generation usage notes and limitations, see the function or System object reference page.

Code generation for more Robotics System Toolbox functions and classes

- `robotics.AimingConstraint`
- `robotics.Cartesianbounds`
- `robotics.GeneralizedInverseKinematics`
- `robotics.InverseKinematics`
- `robotics.Joint`
- `robotics.JointPositionBounds`
- `robotics.PoseTarget`
- `robotics.PositionTarget`
- `robotics.OrientationTarget`
- `robotics.RigidBody`
- `robotics.RigidBodyTree`
- `transformScan`

For C/C++ code generation usage notes and limitations, see the function or class reference page.

Code generation for more Signal Processing Toolbox functions

- `alignsignals`
- `cconv`
- `convmtx`
- `corrmtx`
- `envelope`
- `finddelay`
- `hilbert`
- `sgolayfilt`
- `sinc`
- `xcorr2`
- `xcov`

For C/C++ code generation usage notes and limitations, see the function reference page.

Statistics and Machine Learning Toolbox Code Generation: Generate C code for prediction by using linear models, generalized linear models, decision trees, and ensembles of classification trees

You can generate C code that predicts responses by using trained linear models, generalized linear models (GLM), decision trees, or ensembles of classification trees. The following prediction functions support code generation.

- `predict` — Predict responses or estimate confidence intervals on predictions by applying a linear model to new predictor data.
- `predict` or `glmval` — Predict responses or estimate confidence intervals on predictions by applying a GLM to new predictor data.
- `predict` or `predict` — Classify observations or estimate classification scores by applying a classification tree or ensemble of classification trees, respectively, to new data.

- `predict` — Predict responses by applying a regression tree to new data.

Additionally, you can generate C code to simulate responses from a linear model or generalized linear model using `random` or `random`, respectively.

Code generation for more WLAN System Toolbox functions and System objects

- `wlanDMGConfig`
- `wlanSymbolTimingEstimate`
- `wlanTGahChannel`

For C/C++ code generation usage notes and limitations, see the function or class reference page.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2016b

Version: 3.2

New Features

Bug Fixes

Compatibility Considerations

Recursive Functions and Anonymous Functions: Generate code for more MATLAB language constructs

Recursive Functions

In R2016b, you can use recursive functions in MATLAB code that is intended for code generation. To generate code for recursive functions, MATLAB Coder uses compile-time recursion or run-time recursion. With compile-time recursion, the code generator creates multiple copies of the function in the generated code. The inputs to the copies have different sizes or constant values. With run-time recursion, the code generator produces recursive functions in the generated code. You can influence whether the code generator uses compile-time or run-time recursion by modifying your MATLAB code. You can disallow recursion or disable run-time recursion by modifying configuration parameters. See *Code Generation for Recursive Functions*.

Anonymous Functions

In R2016b, you can use anonymous functions in MATLAB code that is intended for code generation. For example, you can generate code for this MATLAB code that defines an anonymous function that finds the square of a number:

```
sqr = @(x) x.^2;  
a = sqr(5);
```

Anonymous functions are useful for creating a function handle to pass to a MATLAB function that evaluates an expression over a range of values. For example, this MATLAB code uses an anonymous function to create the input to the `fzero` function:

```
b = 2;  
c = 3.5;  
x = fzero(@(x) x^3 + b*x + c, 0);
```

For code generation limitations for anonymous functions, see *Code Generation for Anonymous Functions*.

I/O Support: Generate code for `fseek`, `ftell`, `fwrite`

- `fseek`
- `ftell`
- `fwrite`

See [Data and File Management in MATLAB in Functions and Objects Supported for C/C++ Code Generation — Category List](#).

Statistics and Machine Learning Toolbox Code Generation: Generate code for prediction by using SVM and logistic regression models

You can generate C code that classifies new observations by using trained, binary support vector machine (SVM) or logistic regression models, or multiclass SVM or logistic regression via error-correcting output codes (ECOC).

- `saveCompactModel` compacts and saves the trained model to disk.
- `loadCompactModel` loads the compact model in a prediction function that you declare. The prediction function can, for example, accept new observations and return labels and scores.
- `predict` classifies and estimates scores for the new observations in the prediction function.
 - To classify by using binary SVM models, see `predict`.
 - To classify by using binary logistic regression models, see `predict`.
 - To classify by using multiclass SVM or logistic regression via ECOC, see `predict`.

Communications and DSP Code Generation: Generate code for more functions

Communications System Toolbox

- `iqimbal`
- `comm.BasebandFileReader`
- `comm.BasebandFileWriter`
- `comm.EyeDiagram`
- `comm.PreambleDetector`

See [Communications System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List](#).

DSP System Toolbox

- `dsp.MovingAverage`
- `dsp.MovingMaximum`
- `dsp.MovingMinimum`
- `dsp.MovingRMS`
- `dsp.MovingStandardDeviation`
- `dsp.MovingVariance`
- `dsp.MedianFilter`
- `dsp.BinaryFileReader`
- `dsp.BinaryFileWriter`
- `dsp.Channelizer`
- `dsp.ChannelSynthesizer`

See DSP System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Phased Array System Toolbox

- `musicdoa`
- `pambgfun`
- `taylortaperc`
- `phased.GSCBeamformer`
- `phased.WidebandBackscatterRadarTarget`
- `phased.WidebandTwoRayChannel`
- `phased.MUSICEstimator`
- `phased.MUSICEstimator2D`

See Phased Array System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

WLAN System Toolbox

- `wlanFormatDetect`
- `wlanPacketDetect`

- wlanS1GConfig

See WLAN System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Wavelet Toolbox Code Generation: Generate code for discrete wavelet analysis, synthesis, and denoising functions

In R2016b, you can use MATLAB Coder to generate code for 29 Wavelet Toolbox™ functions that support:

- 1-D and 2-D discrete wavelet analysis, synthesis, and denoising
- 1-D undecimated discrete wavelet and wavelet packet analysis and synthesis

For the list of functions, see Wavelet Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Variable-Size Cell Array Support: Use cell to create a variable-size cell array for code generation

In MATLAB code that is intended for code generation, to create a variable-size cell array, you can use the `cell` function. For example:

```
function z = mycell(n, j)
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

See Definition of Variable-Size Cell Array by Using `cell`.

Targeted Include Statements for `coder.cinclude`: Generate include statements only where indicated

In previous releases, regardless of the location of a `coder.cinclude(headerfile)` call, MATLAB Coder included the header file in almost all C/C++ source files, except for some utility files. The include statement appeared in a file even if it was not required in

that file. In R2016b, the location of the `coder.cinclude(headerfile)` call determines which files include the header file. The header file is included only in the C/C++ source files generated from the MATLAB code that contains the `coder.cinclude` call. By reducing extraneous include statements, the R2016b behavior can reduce compile time and make the generated code more readable.

To preserve the behavior from R2016a and earlier releases, use the following syntax:

```
coder.cinclude(headerfile, 'InAllSourceFiles', true)
```

In a MATLAB Function block, the R2016b behavior for `coder.cinclude(headerfile)` is the same as the behavior in previous releases. The syntax `coder.cinclude(headerfile, 'InAllSourceFiles', allfiles)` behaves the same as `coder.cinclude(headerfile)`.

Compatibility Considerations

If your code assumes that all header files specified by `coder.cinclude` calls are included in each C/C++ source file, your code might not compile in R2016b. For example, suppose that all `coder.cinclude` calls are in a separate function instead of with the `coder.ceval` calls. In R2016b, the C/C++ files that contain the code generated from the `coder.ceval` calls do not include the required header files.

To address this incompatibility, you can preserve the legacy behavior by using this syntax:

```
coder.cinclude(headerfile, 'InAllSourceFiles', true)
```

To avoid the extraneous include statements, rewrite your code to place the `coder.cinclude` calls with the `coder.ceval` calls that require them. Use this syntax:

```
coder.cinclude(headerfile)
```

See `coder.cinclude`.

Generated Code Readability: Generate more readable code for control flow

In R2016b, MATLAB Coder simplifies the generated code for certain control flow patterns such as:

- Empty true branches
- If blocks with identical conditions or branches
- Nested if blocks that check the same condition

From a previous release, here is an example of generated C code that has an empty true branch.

```
double foo(double x)
{
    double y;
    y = 0.0;
    if (x > 10.0) {
    } else {
        y = 1.0;
    }

    return y;
}
```

In R2016b, MATLAB Coder generates the following code that does not include the empty true branch.

```
double foo(double x)
{
    double y;
    y = 0.0;
    if (!(x > 10.0)) {
        y = 1.0;
    }

    return y;
}
```

JIT MEX Compilation: Use JIT compilation to reduce code generation times for MEX

In R2016b, you can speed up generation of MEX functions by specifying use of just-in-time (JIT) compilation technology. When you iterate between modifying MATLAB code and testing the MEX code, this option can save time.

By default, MATLAB Coder does not use JIT compilation. It creates a C/C++ MEX function by generating and compiling C/C++ code. When you specify JIT compilation,

MATLAB Coder creates a JIT MEX function that contains an abstract representation of the MATLAB code. When you run the JIT MEX function, MATLAB generates the executable code in memory.

JIT compilation is incompatible with some code generation features or options, such as custom code or use of the OpenMP library for parallelization of `for`-loops (`parfor`). If you specify JIT compilation and MATLAB Coder is unable to use it, it generates a C/C++ MEX function with a warning.

In the MATLAB Coder app, to specify use of JIT compilation:

- 1 In the **Generate** dialog box, set **Build type** to `MEX`.
- 2 Select the **Use JIT compilation** check box.

At the command line, to specify use of JIT compilation, use the `-jit` option of the `codegen` command. Alternatively, use the `EnableJIT MEX` code configuration parameter.

See [Speed Up MEX Generation by Using JIT Compilation](#).

When generating MEX functions in the **Check for Run-Time Issues** step, the MATLAB Coder app tries to use JIT compilation. If the app is unable to use it, it generates a C/C++ MEX function. You can disable JIT compilation in the **Check for Run-Time Issues** step. See [Check for Run-Time Issues by Using the App](#).

Change in default value for preserve variable names option

In R2016b, the default value for the `PreserveVariableNames` code configuration parameter is `'None'` instead of `'UserNames'`. When this parameter is `'None'`, to reduce memory usage, MATLAB Coder tries to reuse variables in the generated code. When this parameter is `'UserNames'`, to generate more readable, traceable code, MATLAB Coder preserves your variable names in the generated code.

The equivalent MATLAB Coder app setting is **Preserve variable names**. In R2016b, the default value for this setting is `None`.

Compatibility Considerations

In R2016b, when you use the default value for the preserve variable names option, MATLAB Coder does not preserve your variable names in the generated code. If code

readability is more important than reduced memory usage, change the value of this option. At the command line, set the `PreserveVariableNames` code configuration parameter to `'UserNames'`. In the MATLAB Coder app, project build settings, on the **All Settings** tab, set **Preserve variable names** to `User names`.

Code generation error for testing equality between enumeration and character array

For code generation, an enumeration class must derive from a built-in numerical class. In R2016b, MATLAB introduces a new behavior for testing equality between these enumerations and a character array or cell array of character arrays. In previous releases, MATLAB compared the enumeration and character array character-wise. The MATLAB Coder behavior matched the MATLAB behavior. In R2016b, MATLAB compares the enumeration name with the character array. In R2016b, code generation ends with this error message:

Code generation does not support comparing an enumeration to a character array or cell array with the operators `'=='` and `'~='`

Consider this enumeration class:

```
classdef myColors < int8
    enumeration
        RED(1),
        GREEN(2)
    end
end
```

The following code compares an enumeration with the character vector `'RED'`:

```
mode = myColors.RED;
z = (mode == 'RED');
```

In previous releases, the answer in MATLAB and generated code was:

```
0 0 0
```

In R2016b, the answer in MATLAB is:

```
1
```

In R2016b, code generation ends with an error.

Compatibility Considerations

If you want the behavior of previous releases, cast the character array to a built-in numeric class. For example, use the built-in class from which the enumeration derives.

```
mode = myColors.RED;
z = (mode == int8('RED'));
```

Change to default standard math library for C++

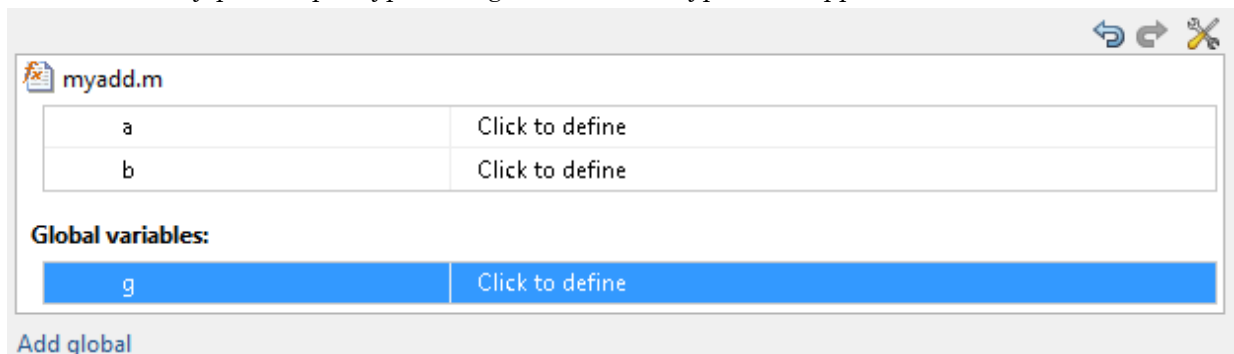
In R2016b, the default standard math library for C++ is ISO/IEC 14882:2003 C++ (C++03 (ISO)). In previous releases, the default standard math library for C++ was the same as the default standard math library for C.

See [Configure Build Settings and Change the Standard Math Library](#).

Simplified type definition in the MATLAB Coder app

In R2016b, you can more easily define input and global variable types in the MATLAB Coder app.

Entry-point input types and global variable types now appear in a combined table.

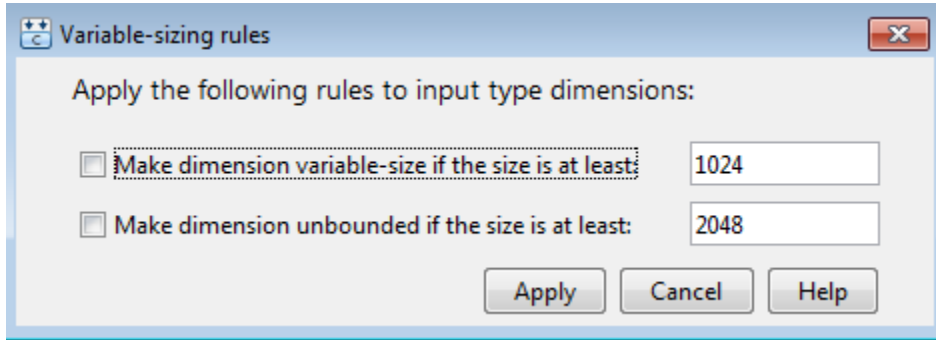


Undo/redo and tools menu actions apply to the items in the combined table.

Using new options, you can more easily define types for a group of types that meet certain conditions.

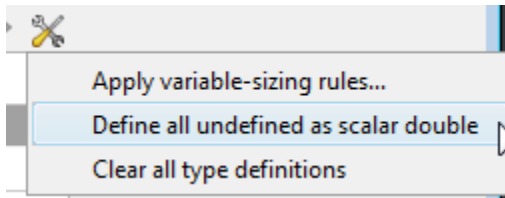
- After you define your input types, in one step, you can make types variable-size when they meet a size threshold. If the test file that you use to automatically define input types results in fixed-size types, use this option to make variable-size types.

You can specify a size threshold for making a dimension variable-size with an upper bound and a threshold for making a dimension variable-size with no upper bound.



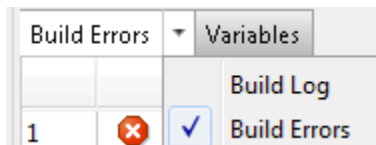
These rules apply to all current type definitions. If you change type definitions, the rules do not affect the new definitions unless you apply them. See Make Dimensions Variable-Size When They Meet Size Threshold.

- You can make all undefined types scalar double in one step. From the tools menu, select Define all undefined as scalar double.




More discoverable build log and errors in MATLAB Coder app

In previous releases, in the **Generate Code** step, the MATLAB Coder app placed the **Build Errors** and **Build Log** tabs on top of each other. To see a hidden tab, you opened a menu and selected the tab.



In R2016b, the **Build Errors** tab is named the **Errors** tab, and the **Build Log** tab is named the **Target Build Log** tab. These tabs are separate so that you can more easily find them.

Target Build Log		Variables	Errors	
		Function	Line	Description
1		foo	6	Attempt to access

Improved workflow for collecting and analyzing ranges in MATLAB Coder app

The **Simulate** and **Derive** buttons on the **Convert to Fixed Point** page of the MATLAB Coder 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 the **Specify design ranges** or the **Analyze ranges using derived range analysis** option is 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 control which ranges are used for data type proposal in the **Settings** pane.

Convert to Fixed Point

SETTINGS ANALYZE CONVERT TEST

Source Code

ex_2ndOrder_filter

Analyze ranges using simulation Specify design ranges

Test bench ex_2ndOrder_filter_test.m Log data for histogram Show code coverage

Analyze ranges using derived range analysis

Timeout (minutes) Quick derived range analysis

Analyze Ranges

```

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

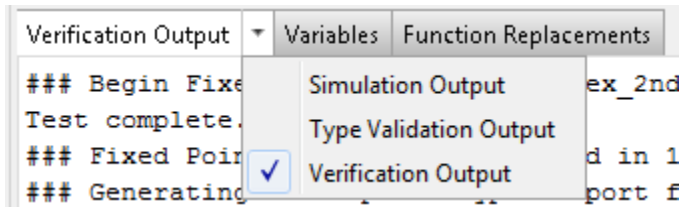
```

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							

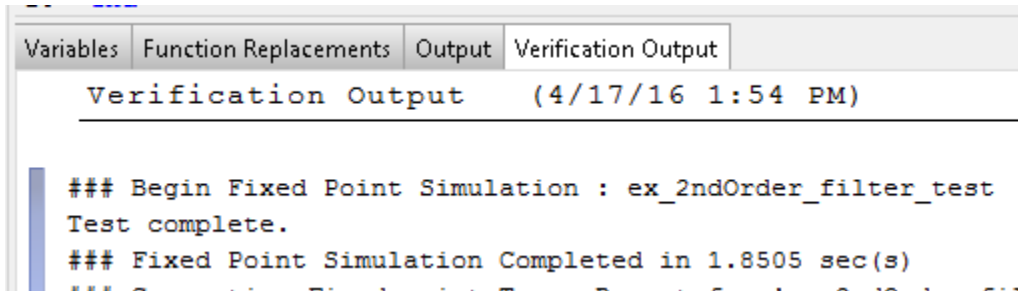
Back Next

More discoverable logs and reports for fixed-point conversion in MATLAB Coder app

In previous releases, in the **Convert to Fixed Point** step, the MATLAB Coder app displayed logs and report links for range analysis, fixed-point conversion, and verification on separate tabs that were placed on top of each other. To see a hidden tab, you opened a menu and selected the tab.



In R2016b, the app displays logs and report links for range analysis and fixed-point conversion on the **Output** tab. It displays logs and report links for verification on the **Verification Output** tab. These tabs are separate so that you can more easily find them.



Hierarchical packaging of generated code in MATLAB Coder app

In previous releases, the MATLAB Coder app packaged generated files in a zip file as a single, flat folder. In R2016b, you can choose flat or hierarchical packaging.

- 1 On the **Finish Workflow** page, click **Package**.
- 2 For **Save as type**, select **Flat zip file** or **Hierarchical zip file**. The default value is **Flat zip file**.

Code generation for additional MATLAB functions

- `cplxpair`
- `fminbnd`
- `inpolygon`
- `isenum`
- `polyeig`

- `repelem`

See [Functions and Objects Supported for C/C++ Code Generation — Alphabetical List](#).

Code generation for additional Audio System Toolbox functions

- `integratedLoudness`
- `loudnessMeter`
- `octaveFilter`
- `weightingFilter`

See [Audio System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List](#).

Code generation for additional Computer Vision System Toolbox functions

- `cameraPoseToExtrinsics`
- `extrinsicsToCameraPose`
- `worldToImage` method of the `cameraParameters` object
- `estimateEssentialMatrix`
- `estimateWorldCameraPose`
- `relativeCameraPose`

See [Computer Vision System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List](#).

Code generation for additional Robotics System Toolbox functions

- `robotics.BinaryOccupancyGrid`
- `robotics.OccupancyGrid`
- `robotics.OdometryMotionModel`
- `robotics.PRM` — The map input must be specified on creation of the `PRM` object.

See Robotics System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Code generation for `extendedKalmanFilter` and `unscentedKalmanFilter` with Control System Toolbox or System Identification Toolbox

You can generate code for the `extendedKalmanFilter` and `unscentedKalmanFilter` functions with the Control System Toolbox™ or System Identification Toolbox™ products:

- `extendedKalmanFilter` in the Control System Toolbox documentation.
- `extendedKalmanFilter` in the System Identification Toolbox documentation.
- `unscentedKalmanFilter` in the Control System Toolbox documentation.
- `unscentedKalmanFilter` in the System Identification Toolbox documentation.

See System Identification Toolbox and Control System Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2016a

Version: 3.1

New Features

Bug Fixes

Compatibility Considerations

Cell Array Support: Use additional cell array features in MATLAB code for code generation

In R2016a, code generation support for cell arrays includes:

Use of `{end + 1}` to grow a cell array

You can write code such as `X{end + 1}` to grow a cell array `X`. For example:

```
X = {1 2};  
X(end + 1) = 'a';
```

When you use `{end + 1}` to grow a cell array, follow the restrictions described in [Growing a Cell Array by Using `{end + 1}`](#).

Value and handle objects in cell arrays

Cell arrays can contain value and handle objects. You can use a cell array of objects as a workaround for the limitation that code generation does not support objects in matrices or structures.

Function handles in cell arrays

Cell arrays can contain function handles.

Non-Power-of-Two FFT Support: Generate code for fast Fourier transforms for non-power-of-two transform lengths

In previous releases, code generation required a power of two transform length for `fft`, `fft2`, `fftn`, `ifft`, `ifft2`, and `ifftn`. In R2016a, code generation allows a non-power-of-two length for these functions.

Faster Standalone Code for Linear Algebra: Generate code that takes advantage of your own target-specific LAPACK library

To improve the execution speed of code generated for algorithms that call linear algebra functions, MATLAB Coder can generate calls to LAPACK functions by using the LAPACKE C interface to LAPACK. If the input arrays for the linear algebra functions meet certain criteria, MATLAB Coder generates calls to relevant LAPACK functions. In

R2015b, only generated MEX called LAPACK functions. In R2016a, generated standalone code can call LAPACK functions.

LAPACK is a software library for numerical linear algebra. MATLAB uses this library in some linear algebra functions such as `eig` and `svd`. For MEX functions, MATLAB Coder uses the LAPACK library that is included with MATLAB. For standalone code, MATLAB Coder uses the LAPACK interface for the LAPACK library that you specify. If you do not specify a LAPACK library, MATLAB Coder generates code for the linear algebra function instead of generating a LAPACK call.

See [Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls](#).

Computer Vision System Toolbox and Image Processing Toolbox Code Generation: Generate code for additional functions

See C code generation support in the Computer Vision System Toolbox™ release notes.

See C-code generation: Generate code from 20 additional functions using MATLAB Coder in the Image Processing Toolbox™ release notes.

MATLAB Coder Student Access: Obtain MATLAB Coder as student-use, add-on product or with MATLAB Primary and Secondary School Suite

Starting with R2016a, MATLAB Coder is available for purchase as an add-on product for student-use software: MATLAB Student™ and MATLAB and Simulink® Student Suite™. Student-use software provides the same tools that professional engineers and scientists use. Students use the software to develop skills that help them excel in courses and prepare for careers.

Starting with R2016a, MATLAB Coder is included in the MATLAB Primary and Secondary School Suite.

Concatenation of Variable-Size Empty Arrays: Generate code for concatenation when a component array is empty

In R2016a, the MATLAB Coder treatment of an empty array in a concatenation more closely matches the MATLAB treatment.

For concatenation of arrays, MATLAB and MATLAB Coder require that corresponding dimensions across component arrays have the same size, except for the dimension that grows. For horizontal concatenation, the second dimension grows. For vertical concatenation, the first dimension grows.

In MATLAB, when a component array is empty, the sizes of the nongrowing dimensions do not matter because MATLAB ignores empty arrays in a concatenation. In previous releases, MATLAB Coder required that the sizes of nongrowing dimensions of a variable-size, empty array matched the sizes of the corresponding dimensions in the other component arrays. A dimension size mismatch resulted in an error in the MEX function and a possible incorrect answer in standalone code.

In R2016a, for most cases of empty arrays in concatenation, MATLAB Coder behavior matches MATLAB behavior. In some cases, if MATLAB Coder does not recognize the empty array and treats it as a variable-size array, a dimension size mismatch results in a compile-time error.

Consider the function `myconcat` that concatenates two arrays.

```
function C = myconcat(A, B)
    C = [A, B];
end
```

Define the types `IN1` and `IN2`. `IN1` is variable-size in both dimensions with no upper bounds. `IN2` is variable-size with an upper bound of 5 in each dimension.

```
IN1 = coder.typeof(1, [Inf Inf], [1 1]);
IN2 = coder.typeof(1, [5 5], [1 1]);
```

Generate MEX for `myconcat`. Use the `-args` option to indicate that the input arguments have the types defined by `IN1` and `IN2`.

```
codegen myconcat -args {IN1, IN2} -report
```

Define `R1` and `R2`.

```
R1 = zeros(0,5);
R2 = magic(3)
```

`R1` is a 0-by-5 empty matrix. `R2` is a 3-by-3 matrix.

In previous releases, `myconcat_mex(R1, R2)` resulted in a size mismatch error. The size of dimension 1 of the empty array `R1` did not match the size of dimension 1 of `R2`. In

R2016a, `myconcat_mex(R1, R2)` produces the same answer as the answer in MATLAB.

`ans =`

```

     8     1     6
     3     5     7
     4     9     2

```

Compatibility Considerations

When the result of the concatenation is assigned to a variable that must be fixed-size, support for a variable-size, empty array in a concatenation introduces an incompatibility.

In previous releases, it is possible that a concatenation that included a variable-size array produced a fixed-size array because concatenation rules were stricter in MATLAB Coder than in MATLAB. In R2016a, a concatenation that includes a variable-size array produces a variable-size array. If the result of the concatenation is assigned to a variable that must be fixed-size, the code generation software produces a compile-time error.

Consider the function `myconcat`.

```

function Z = myconcat1(X, Y)
%#codegen
Z.f = [X Y];

```

Suppose that you generate a MEX function for `myconcat1`. Suppose that you specify these sizes for the input arguments:

- `X` has size `?:by-2`. The first dimension has a variable size with no upper bound and the second dimension has a fixed size of 2.
- `Y` has size `2-by-4`.

In the generated code, the size of the result of `[X Y]` is `2-by-:6`. The first dimension has a fixed size of 2 and the second dimension has a variable size with an upper bound of 6. This size accommodates both an empty and nonempty `X`. If you pass an empty `X` to `myconcat_mex`, the size of the result is `2-by-4`. If you pass a nonempty `X` to `myconcat_mex`, the size of the result is `2-by-6`.

Consider the function `myconcat2`.

```

function Z = myconcat2(X, Y)
%#codegen

```

```
Z.f = ones(2, 6);
myfcn(Z);
Z.f = [X Y];
```

```
function myfcn(~)
```

myconcat2 assigns a 2-by-6 value to `Z.f`. At compile time, the size of `Z.f` is fixed at 2-by-6 because `Z` is passed to `myfcn`. In the assignment `Z.f = [X Y]`, the result of the concatenation `[X Y]` is variable-size. Code generation fails because the left side of the assignment is fixed-size and the right side is variable-size.

To work around this incompatibility, you can use `coder.varsize` to declare that `Z.f` is variable-size.

```
function Z = myconcat2(X, Y)
%#codegen
coder.varsize('Z.f');
Z.f = ones(2, 6);
myfcn(Z);
Z.f = [X Y];
```

```
function myfcn(~)
```

memset Optimization for More Cases: Optimize code that assigns a constant value to consecutive array elements

To optimize generated code that assigns a literal constant to consecutive array elements, the code generation software tries to replace the code with a `memset` call. A `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

In R2016a, MATLAB Coder invokes the `memset` optimization for more cases than in previous releases.

A loop with multiple assignments.

Previous Releases	R2016a
<pre>for (i = 0; i < 100; i++) { Y1[i] = 0.0; Y2[i] = 0.0; Y3[i] = 0.0; }</pre>	<pre>memset (&Y1[0],0,100U*sizeof(double)); memset (&Y2[0],0,100U*sizeof(double)); memset (&Y3[0],0,100U*sizeof(double));</pre>

Consecutive statements that define a continuous write.

Previous Releases	R2016a
<pre>Y1[0] = 255; Y1[1] = 255; Y1[2] = 255; ... Y1[99] = 255</pre>	<pre>memset(&Y1[0], 255, 100U * sizeof(unsigned char));</pre>

A structure that contains an array.

Previous Releases	R2016a
<pre>for (i = 0; i < 100; i++) { S->f1[i] = 0.0; }</pre>	<pre>memset(&S>f1[0], 0, 100U * sizeof(double));</pre>

All fields of a structure array assigned the same constant value.

Previous Releases	R2016a
<pre>for (i = 0; i < 100; i++) { S[i].f1 = 255; S[i].f2 = 255; S[i].f3 = 255; }</pre>	<pre>memset(&S[0], 255, 100U * sizeof(struct0_T));</pre>

For information about settings that affect the `memset` optimization, see [memset Optimization](#).

Optimization for Conditional and Boolean Expressions: Generate efficient code for more cases

For certain conditional and Boolean expressions, MATLAB Coder optimizes the generated code by replacing expressions with simpler, more efficient expressions. In R2016a, MATLAB Coder uses this optimization for more cases.

Here are examples of this optimization.

Previous Releases	R2016a
<pre>if (cond) { y = true; } else { y = val; } return y;</pre>	<pre>return cond val;</pre>
<pre>y = x && !x;</pre>	<pre>y = false;</pre>

MATLAB Coder App Line Execution Count: See how well test exercises MATLAB code

When you perform the **Check for Run-Time Issues** step in the MATLAB Coder app, you must provide a test that calls your entry-point functions with representative data.

The **Check for Run-Time Issues** step generates a MEX function from your MATLAB functions and runs the test replacing calls to the MATLAB functions with calls to the MEX function. In R2016a, to help you see how well your test exercises your MATLAB code, the app collects and displays line execution counts. When the app runs the MEX function, the app counts executions of the MEX code that corresponds to a line of MATLAB code.

To see the line execution counts, after you check for run-time issues, click **View MATLAB line execution counts**.

Check for Run-Time Issues SETTINGS CHECK FOR ISSUES ▾

This step creates a MEX function from your MATLAB function(s), invokes the MEX function, and reports issues that may be hard to diagnose in the generated C code. [Learn more](#)

Enter code or select a script that exercises **myfunction**:

>> myfunction_test ▾ ↵ ...

Collect MATLAB line execution counts Check for Issues

✔ No issues detected. [View MATLAB line execution counts](#)

✔
Generating trial code
✔
Building MEX
✔
Running test file with MEX

The app displays your MATLAB code in the app editor. The app displays a color-coded coverage bar to the left of the code. This table describes the color coding.

Color	Indicates
Green	<p>One of the following situations:</p> <ul style="list-style-type: none"> The entry-point function executes multiple times and the code executes more than one time. The entry-point function executes one time and the code executes one time. <p>Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.</p>
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you position your cursor over the coverage bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that the section executes.

The screenshot shows the MATLAB Code Editor with the following code and execution counts:

```

1 function [y, z] = myfunction(x)           2 calls
2
3 persistent p;
4
5 if isempty(p)
6     p = 1;                               1 calls
7 end                                       2 calls
8 y = zeros(10, 10);
9 if x > 0
10    for i = 1:10                          20 calls
11        for j = 1:10                      200 calls
12            if i == 1 && j > 1
13                y(i, j) = 2 * x;          18 calls
14            elseif j == 1 && i > 1
15                y(i, j) = x;             18 calls
16            else
17                y(i, j) = 0;              164 calls
18            end
19        end                               200 calls
20    end                                    20 calls
21 else
22     p = p + 1;                             0 calls
23 end
24



```

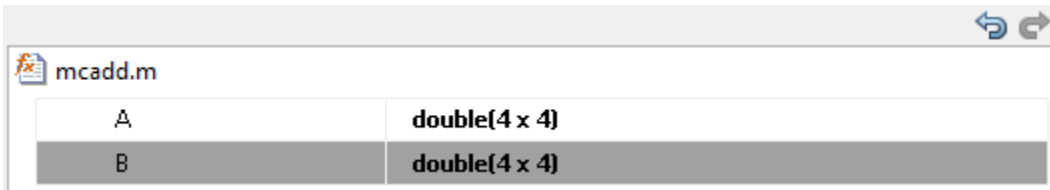
Line execution count collection is enabled by default. To disable the collection, clear the **Collect MATLAB line execution counts** check box. If line execution collection slows the run-time issue checking, consider disabling it.



See [Collect and View Line Execution Counts for Your MATLAB Code](#).

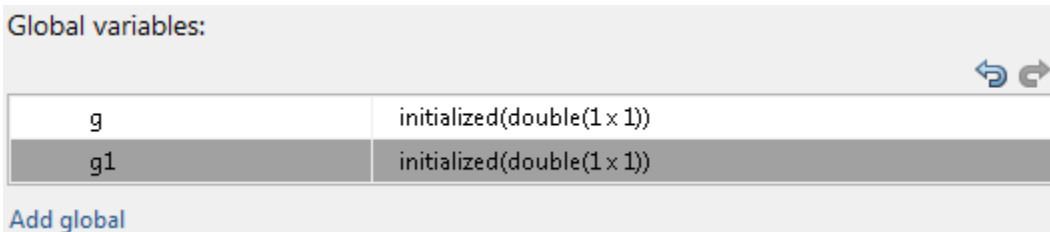
MATLAB Coder App Undo and Redo: Easily revert changes to type definitions

In R2016a, you can revert and restore changes to type definitions in the **Define Input Types** step of the MATLAB Coder app. Revert and restore changes in the input arguments table or the global variables table.

To revert or restore changes to input argument type definitions, above the input arguments table, click  or .



To revert or restore changes to global variable type definitions, above the global variables table, click  or .



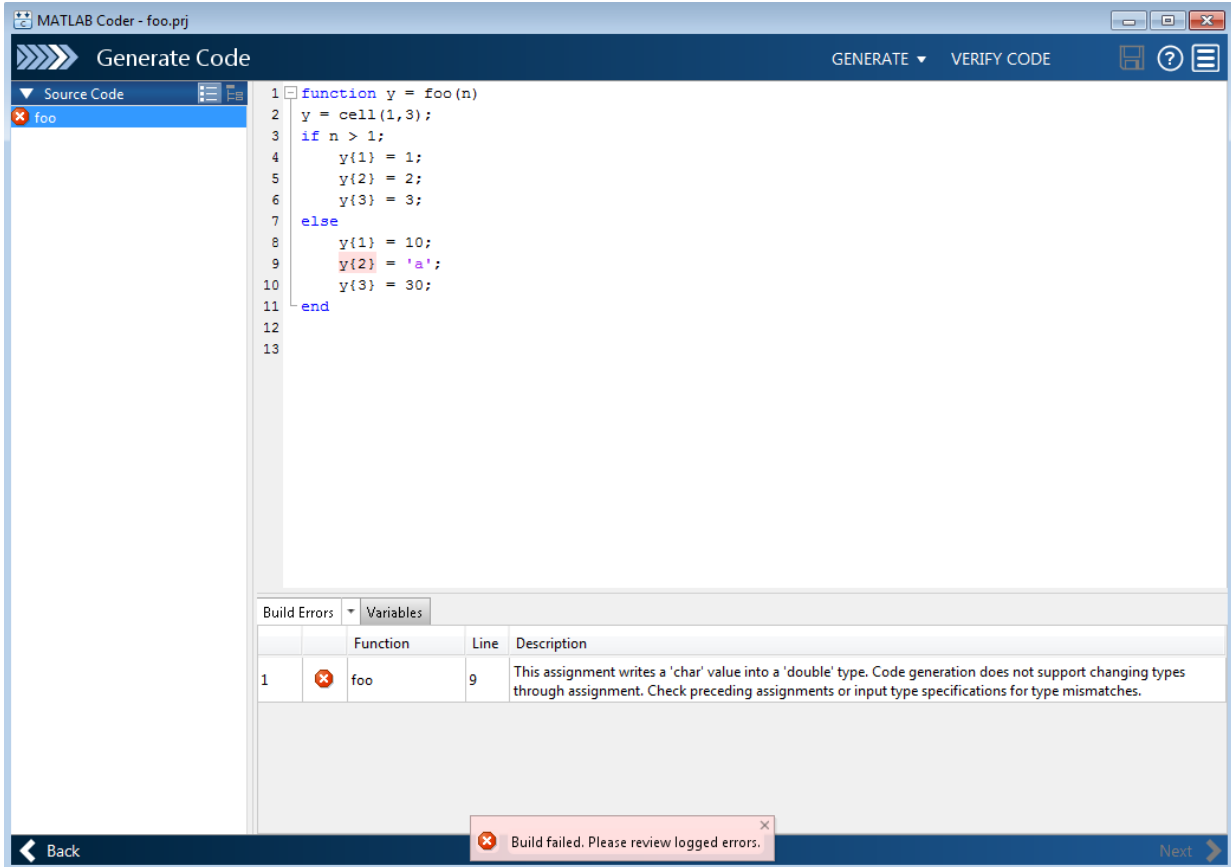
Alternatively, use the keyboard shortcuts for Undo and Redo. The keyboard shortcuts apply to the table that is selected. The shortcuts are defined in your MATLAB preferences. On a Windows® platform, the default keyboard shortcuts for Undo and Redo are **Ctrl+Z** and **Ctrl+Y**.

Each undo operation reverts the last change. Each redo operation restores the last change.

See [Define Keyboard Shortcuts](#).

MATLAB Coder App Error Table: View complete error message

In previous releases, the MATLAB Coder app truncated a message that did not fit on one line of the error message table on the **Build Errors** tab in the **Check for Run-Time Issues** or **Generate Code** steps. In R2016a, the app displays the entire message.



In previous releases, if a message included a link, the app excluded the link from the error in the error message table on the **Build Errors** tab. In R2016a, the app includes the link.

Changes to Fixed-Point Conversion Code Coverage

If you use the MATLAB Coder 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>	

See Code Coverage in Automated Fixed-Point Conversion.

Fixed-point conversion requires the Fixed-Point Designer™ software.

More Keyboard Shortcuts in Code Generation Report: Navigate the report more easily

In R2016a, you can 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
Open selected MATLAB code	Ctrl+D
Step backward through files that you opened in the code pane	Alt+Left
Step forward through files that you opened in the code pane	Alt+Right
Refresh	F5
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.

Zoom In	Ctrl+Plus
Zoom Out	Ctrl+Minus
Evaluate Selection	F9
Help on Selection	F1
Open Selection	Ctrl+D
Back	Alt+Left
Forward	Alt+Right
Refresh	F5
Find...	Ctrl+F
Page Source	

See Define Keyboard Shortcuts and Code Generation Reports.

xcorr Code Generation: Generate faster code for xcorr with long input vectors

For long input vectors, code generation for `xcorr` now uses a frequency-domain calculation instead of a time-domain calculation. The resulting code can be faster than in previous releases.

To use the `xcorr` function, you must have the Signal Processing Toolbox™ software.

Code generation for additional MATLAB functions

Specialized Math in MATLAB

- `airy`
- `besseli`
- `besselj`

Trigonometry in MATLAB

- `deg2rad`
- `rad2deg`

Interpolation and Computational Geometry in MATLAB

- `interp`

Changes to code generation support for MATLAB functions

- Code generation now supports the `nanflag` option for `sum`, `mean`, `median`, `min`, `max`, `cov`, `var`, and `std`.
- Code generation for `ismember` no longer requires that the second input be sorted.

Code generation for Audio System Toolbox functions and System objects

See Audio System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for additional Communications System Toolbox functions

- `convenc`
- `dpskdemod`
- `dpskmod`
- `qammod`
- `qamdemod`
- `vitdec`

See Communications System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for additional DSP System Toolbox

- `audioDeviceWriter`
- `dsp.Differentiator`
- `designMultirateFIR`
- `dsp.SubbandAnalysisFilter`
- `dsp.SubbandSynthesisFilter`

See DSP System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for additional Phased Array System Toolbox functions

- `fogpl`
- `gaspl`
- `rainpl`
- `phased.BackscatterRadarTarget`
- `phased.LOSChannel`
- `phased.WidebandLOSChannel`

See Phased Array System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for additional Robotics System Toolbox functions

- `robotics.ParticleFilter`

See Robotics System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for WLAN System Toolbox functions and System objects

See WLAN System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2015aSP1

Version: 2.8.1

Bug Fixes

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2015b

Version: 3.0

New Features

Bug Fixes

Compatibility Considerations

Cell Array Support: Generate C code from MATLAB code that uses cell arrays

In R2015b, you can generate code from MATLAB code that uses cell arrays.

The code generation software classifies a cell array as homogeneous or heterogeneous. This classification determines how a cell array is represented in the generated C/C++ code. It also determines how you can use the cell array in MATLAB code from which you generate C/C++ code. See [Homogeneous vs. Heterogeneous Cell Arrays](#).

As long as you do not specify conflicting requirements, you can control whether a cell array is homogeneous or heterogeneous. See [Control Whether a Cell Array is Homogeneous or Heterogeneous](#).

When you use cell arrays in MATLAB code from which you generate C/C++ code, you must follow certain restrictions. See [Cell Array Requirements and Limitations for Code Generation](#).

Faster MEX Functions for Linear Algebra: Generate MEX functions that take advantage of LAPACK

To improve the speed of the MEX generated for algorithms that call linear algebra functions, the generated MEX can now call LAPACK functions. If the input arrays for the linear algebra functions meet certain criteria, MATLAB Coder generates calls to relevant LAPACK functions.

LAPACK is a software library for numerical linear algebra. MATLAB uses this library in some linear algebra functions such as `eig` and `svd`. MATLAB Coder uses the LAPACK library that is included with MATLAB.

For information about the open source reference version, see [LAPACK — Linear Algebra PACKage](#).

Double-Precision to Single-Precision Conversion: Convert double-precision MATLAB code to single-precision C code

In R2015b, if you have a Fixed-Point Designer license, you can convert double-precision MATLAB code to single-precision MATLAB code or single-precision C code.

You can develop code for embedded hardware that requires single-precision code without changing your original MATLAB algorithm. You can verify the single-precision code using the same test files that you use for your original algorithm. When a double-precision operation cannot be removed, the code generation report highlights the MATLAB expression that results in that operation.

You can generate single-precision code in the following ways:

- Generate single-precision C code by using the MATLAB Coder app. See [Generate Single-Precision C Code Using the MATLAB Coder App](#).
- Generate single-precision C code by using `codegen` with the `-singleC` option. See [Generate Single-Precision C Code at the Command Line](#).
- Generate single-precision MATLAB code by using `codegen` with a `coder.SingleConfig` object. Optionally, you can generate single-precision C code from the single-precision MATLAB code. See [Generate Single-Precision MATLAB Code](#).

Run-Time Checks in Standalone C Code: Detect and report run-time errors while testing generated standalone libraries and executables

In R2015b, generated standalone libraries and executables can detect and report run-time errors such as out-of-bounds array indexing. In previous releases, only generated MEX detected and reported run-time errors.

By default, run-time error detection is enabled for MEX. By default, run-time error detection is disabled for standalone libraries and executables.

To enable run-time error detection for standalone libraries and executables:

- At the command line, use the code configuration property `RuntimeChecks`.

```
cfg = coder.config('lib'); % or 'dll' or 'exe'  
cfg.RuntimeChecks = true;  
codegen -config cfg myfunction
```

- Using the MATLAB Coder app, in the project build settings, on the **Debugging** tab, select the **Generate run-time error checks** check box.

The generated libraries and executables use `fprintf` to write error messages to `stderr` and `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. Error messages are in English.

See [Run-Time Error Detection and Reporting in Standalone C/C++ Code and Generate Standalone Code That Detects and Reports Run-Time Errors](#).

Multicore Capable Functions: Generate OpenMP-enabled C code from more than twenty MATLAB mathematics functions

For code generation, some MATLAB mathematics functions now use `parfor` to create loops that run in parallel on shared-memory multicore platforms. Loops that run in parallel can be faster than loops that run on a single thread.

Some functions use `parfor` when the number of elements warrants parallelism. These functions include `interp1`, `interp2`, `interp3`, and most functions in Specialized Math in MATLAB. Some functions use `parfor` when they operate on columns and when the number of columns to process warrants parallelism. These functions include `filter`, `median`, `mode`, `sort`, `std`, and `var`.

If your compiler does not support the Open Multiprocessing (OpenMP) application interface, MATLAB Coder treats the `parfor`-loops as `for`-loops. In the generated code, the loop iterations run on a single thread. See http://www.mathworks.com/support/compilers/current_release/.

Image Processing Toolbox and Computer Vision System Toolbox Code Generation: Generate code for additional functions in these toolboxes

Image Processing Toolbox

<code>bwareaopen</code>	<code>houghpeaks</code>	<code>immse</code>	<code>integralBoxFilter</code>
<code>grayconnected</code>	<code>imabsdiff</code>	<code>imresize</code>	<code>psnr</code>
<code>hough</code>	<code>imcrop</code>	<code>imrotate</code>	
<code>houghlines</code>	<code>imgaborfilt</code>	<code>imtranslate</code>	

See [Image Processing Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List](#).

Computer Vision System Toolbox

- `cameraPose`

- `detectCheckerboardPoints`
- `extractLBPFeatures`
- `generateCheckerboardPoints`
- `insertText`
- `opticalFlowFarneback`

See Computer Vision System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Statistics and Machine Learning Toolbox Code Generation: Generate code for kmeans and randsample

- `kmeans`
- `randsample`

See Statistics and Machine Learning Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Simplified hardware specification in the MATLAB Coder app

In R2015b, redesigned dialog boxes simplify the way that you specify hardware settings on the **Generate Code** page and on the project build settings **Hardware** tab. The redesign consolidates hardware settings, supports use of installed hardware support packages for processor-in-the-loop (PIL) execution, and hides hardware implementation details until you want to see them. Use of hardware support packages and PIL execution with MATLAB Coder requires an Embedded Coder license.

Here is the redesigned **Generate Code** page.

Build type:

Language C C++

Hardware Board

Device Generic MATLAB Host Computer
Device vendor Device type

Toolchain

Here is the redesigned project build settings **Hardware** tab.

Hardware

Hardware Board

Device: Generic MATLAB Host Computer
Device vendor Device type

[Customize hardware implementation](#)

Build Process

Toolchain:
 Microsoft Visual C++ 2012 v11.0 | nmake (64-bit Windows)

Build Configuration: [Show settings](#)
 Minimize compilation and linking time

The changes include:

- Toolchain settings on the **Generate Code** page and on the project build settings **Hardware** tab replace the **Toolchain** tab.

- The **Standard math library** and **Code replacement library**, formerly on the **Hardware** tab, are now on the **Custom Code** tab.
- You can specify the **Hardware board** instead of the **Device vendor** and **Device type**. The app populates **Device vendor** and **Device type** based on the hardware board. To specify the hardware on which MATLAB is running, select `MATLAB Host Computer`. To specify the device vendor and type, select `None – Select device below`.

If you have an Embedded Coder license, you can select a board for an installed hardware support package. For R2015b, the hardware support packages are:

- Embedded Coder Support Package for BeagleBone Black Hardware
- Embedded Coder Support Package for ARM® Cortex®-A Processors

For information about using hardware support packages with MATLAB Coder, see the Embedded Coder release notes.

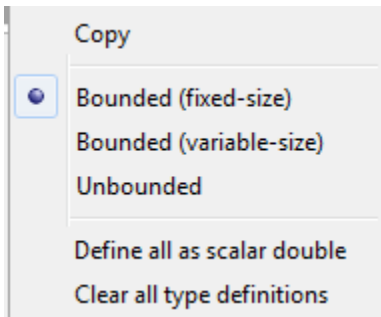
- On the **Hardware** tab, the app hides the hardware implementation details. To see or modify the hardware implementation details, click **Customize hardware implementation**. By default, the test and production hardware implementation settings are the same. The app shows only one set of settings. To display or modify the test and production hardware implementation settings separately, on the **All Settings** tab, under **Hardware**, set **Test hardware is the same as production hardware** to `No`.

MATLAB Coder app user interface improvements

Improvements for manual type definition

Improvements for manual type definition include:

- Context menu options to specify array size.



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

x	struct(1 x 2)	<i>myname</i> +
field1	double(1 x 1)	

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

x	fi(1 x 1)	<i>numerictype(1, 16, 15)</i>
---	------------------	-------------------------------

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

Tab completion for specifying files

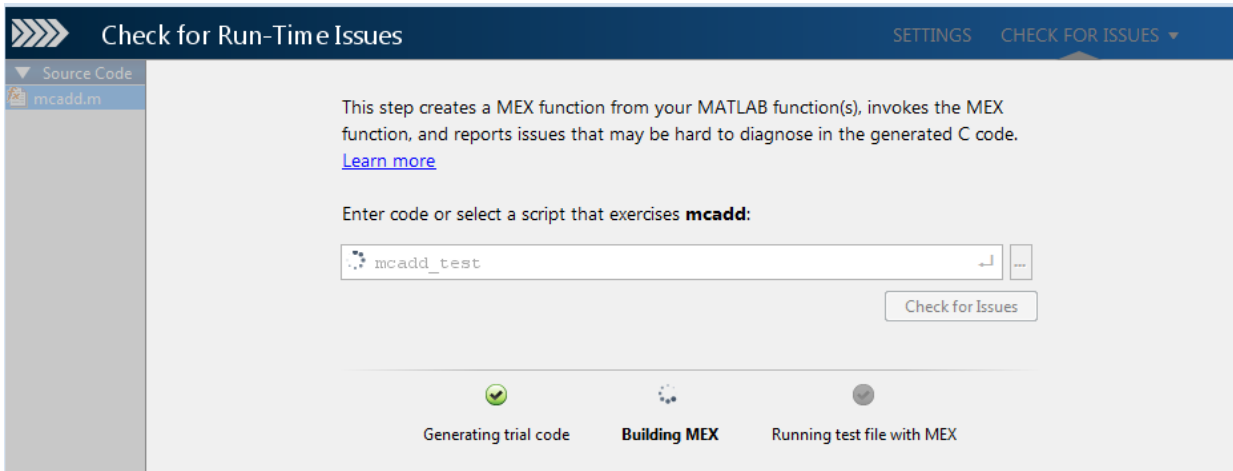
You can use tab completion to specify entry-point functions and test files.

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.

Progress indicators for the Check for Run-Time Issues step

When you perform the **Check for Run-Time Issues** step, you can see progress indicators.




Saving and restoring of workflow state between MATLAB Coder app sessions

In R2015b, when you complete the **Check for Run-Time Issues** or **Generate Code** steps and close the project, the MATLAB Coder app saves the step results. When you reopen the project, you do not have to repeat the step. You can continue from where you left off.

Project reuse between MATLAB Coder and HDL Coder



In R2015b, you can open a MATLAB Coder project in the HDL Coder™ app. You can open an HDL Coder project in the MATLAB Coder app. You must have an HDL Coder license to use the HDL Coder app. When you move between apps, the project settings for both apps are saved. For example, when you open a MATLAB Coder project in the HDL Coder app, the app uses the settings that are common to both apps. It saves the settings that it does not use so that if you open the project in the MATLAB Coder app, those settings are available.

To open a MATLAB Coder project as an HDL Coder project:

- In the MATLAB Coder app, click  and select **Reopen project as HDL Coder**.

- In the HDL Coder app, click the **Open** tab and specify the project.

To open an HDL Coder project as a MATLAB Coder project:

- In the HDL Coder app, click  and select Reopen in MATLAB Coder.
- In the MATLAB Coder app, click  and select Open existing project.

Code generation using freely available MinGW-w64 compiler

In R2015b, you can use the MinGW-w64 compiler from TDM-GCC to generate C/C++ MEX, libraries, and executables on a 64-bit Windows host. For installation instructions, see [Install MinGW-w64 Compiler](#).

When you generate code for C/C++ libraries and executables, you can specify a MinGW compiler toolchain. If you use the command-line workflow, set the `Toolchain` property in a code configuration object for a library or executable:

```
cfg = coder.config('lib')
cfg.Toolchain = 'MinGW64 v4.x | gmake (64-bit Windows)'
```

If you use the MATLAB Coder app, in the project build settings, on the **Hardware** tab, set **Toolchain** to `MinGW64 v4.x | gmake (64-bit Windows)`.

codegen debug option for libraries and executables

In R2015b, for `lib`, `dll`, and `exe` targets, you can use the `-g` option of the `codegen` command to enable the compiler debug mode. In previous releases, the `-g` option enabled the compiler debug mode for MEX targets only.

If you enable debug mode, the C compiler disables some optimizations. The compilation is faster, but the execution is slower.

Compatibility Considerations

In R2015b, for `lib`, `dll`, and `exe` targets, the `-g` option enables the compiler debug mode. In previous releases, for `lib`, `dll`, and `exe` targets, `codegen` ignored the `-g` option. The compiler generated the same code as when you omitted the `-g` option.

Code generation for additional MATLAB functions

Data Types in MATLAB

- `cell`
- `fieldnames`
- `struct2cell`

See Data Types in MATLAB in Functions and Objects Supported for C and C++ Code Generation — Category List.

String Functions in MATLAB

- `iscellstr`
- `strjoin`

See String Functions in MATLAB in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for additional Communications System Toolbox, DSP System Toolbox, and Phased Array System Toolbox System objects

Communications System Toolbox

`comm.CoarseFrequencyCompensator`

See Communications System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

DSP System Toolbox

- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.AllpassFilter`

See DSP System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Phased Array System Toolbox

- `phased.TwoRayChannel`
- `phased.GCCEstimator`
- `phased.WidebandRadiator`
- `phased.SubbandMVDRBeamformer`
- `phased.WidebandFreeSpace`
- `gccphat`

See Phased Array System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for Robotics System Toolbox functions and System objects

See Robotics System Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Code generation for System Identification Toolbox functions and System objects

See System Identification Toolbox in Functions and Objects Supported for C and C++ Code Generation — Category List.

Fixed-Point Conversion Enhancements

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

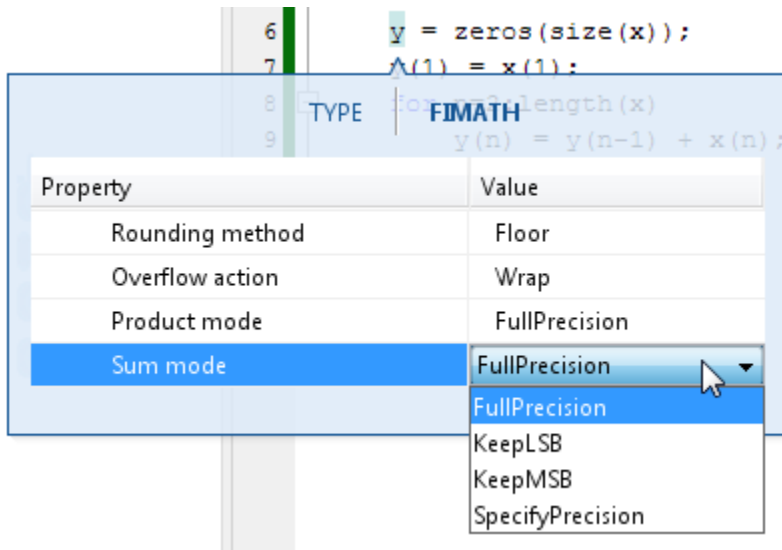
If you close a project before completing the fixed-point conversion process, the app saves your work. When you reopen the project, the app restores the state. You do not have to repeat the fixed-point conversion steps that you completed in a previous session. For example, suppose that you close the project 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.


Reuse of MEX files during fixed-point conversion using the app

During fixed-point conversion, the app minimizes the number of times that it regenerates MEX files. The app rebuilds the MEX files only when required by changes in your code.

Specification of additional fimath properties in app editor

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

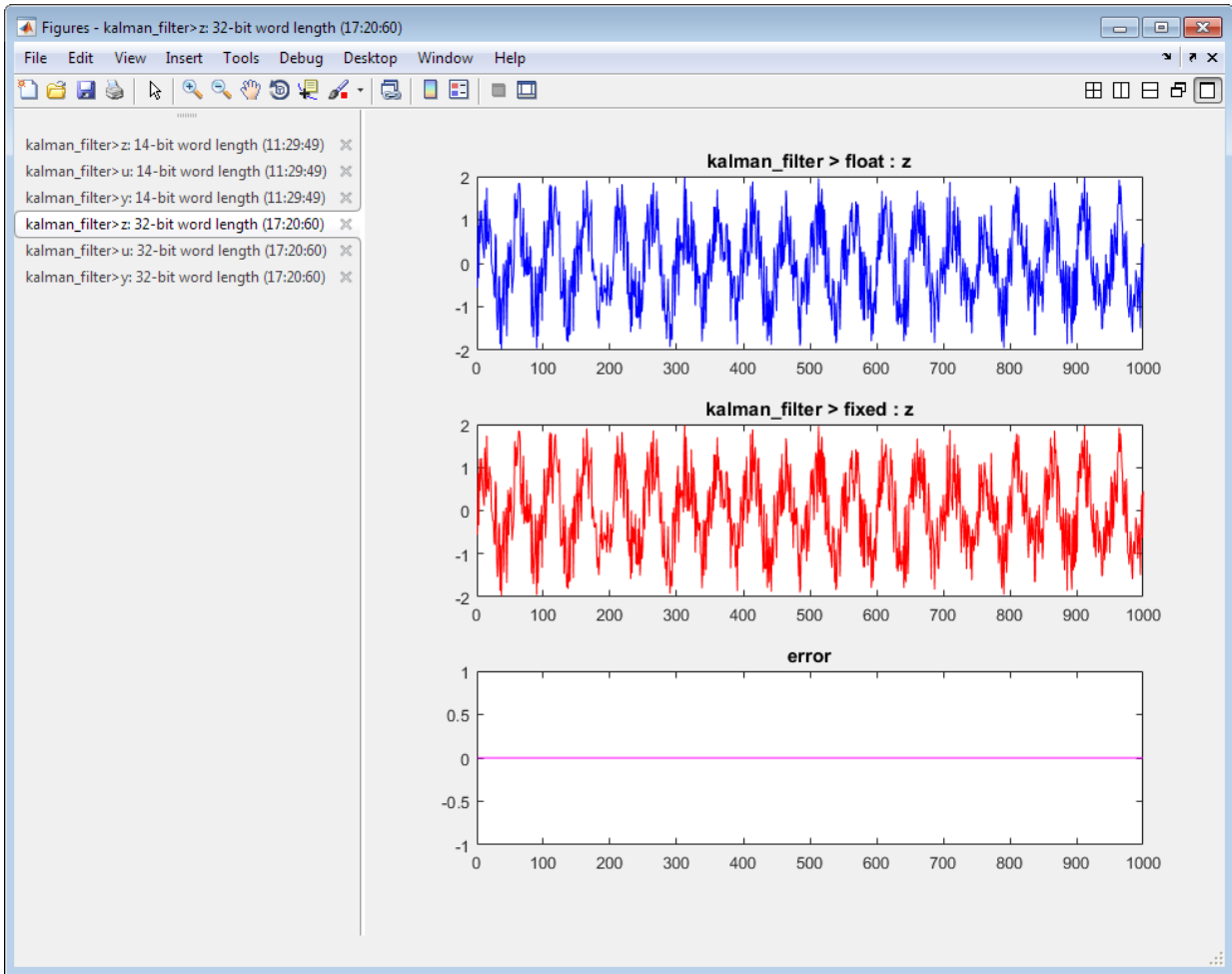


You can also modify these properties from the fixed-point conversion settings dialog box. To open the settings dialog box, on the **Convert to Fixed Point** page, click the **Settings** arrow .

Improved management of comparison plots

During fixed-point conversion, the app docks plots that are 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 subplots. 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 of the app, in the **Variables** table, you can view variable specializations.

The screenshot shows the 'Fixed-Point Converter - foo.pj' window. The main area displays the source code for a MATLAB function named 'foo'. The code is as follows:

```

1 function [y1, y2] = foo(u)
2
3     x = u;
4     y1 = x + 1;
5
6     x = int8(u) + int8(u);
7     y2 = x + int8(1);
8
9 end
10
11

```

Below the code editor, there is a table with the following columns: Variable, Type, Sim Min, Sim Max, Static Min, Static Max, Whole Nu..., and Proposed Type. The table is organized into sections: Input, Output, and Local.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
u	1x 201 double	-100	100			Yes	numerictype(1, 8, 0)
Output							
y1	1x 201 double	-99	101			Yes	numerictype(1, 8, 0)
y2	1x 201 int8	-127	127			Yes	numerictype(1, 8, 0)
Local							
x > 1	1x 201 double	-100	100			Yes	numerictype(1, 8, 0)
x > 2	1x 201 int8	-128	127			Yes	numerictype(1, 8, 0)

At the bottom of the window, there are 'Back' and 'Next' navigation buttons.

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. The expensive fixed-point operations check now highlights expressions in your MATLAB code that can result in multiword operations in generated code.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2015a

Version: 2.8

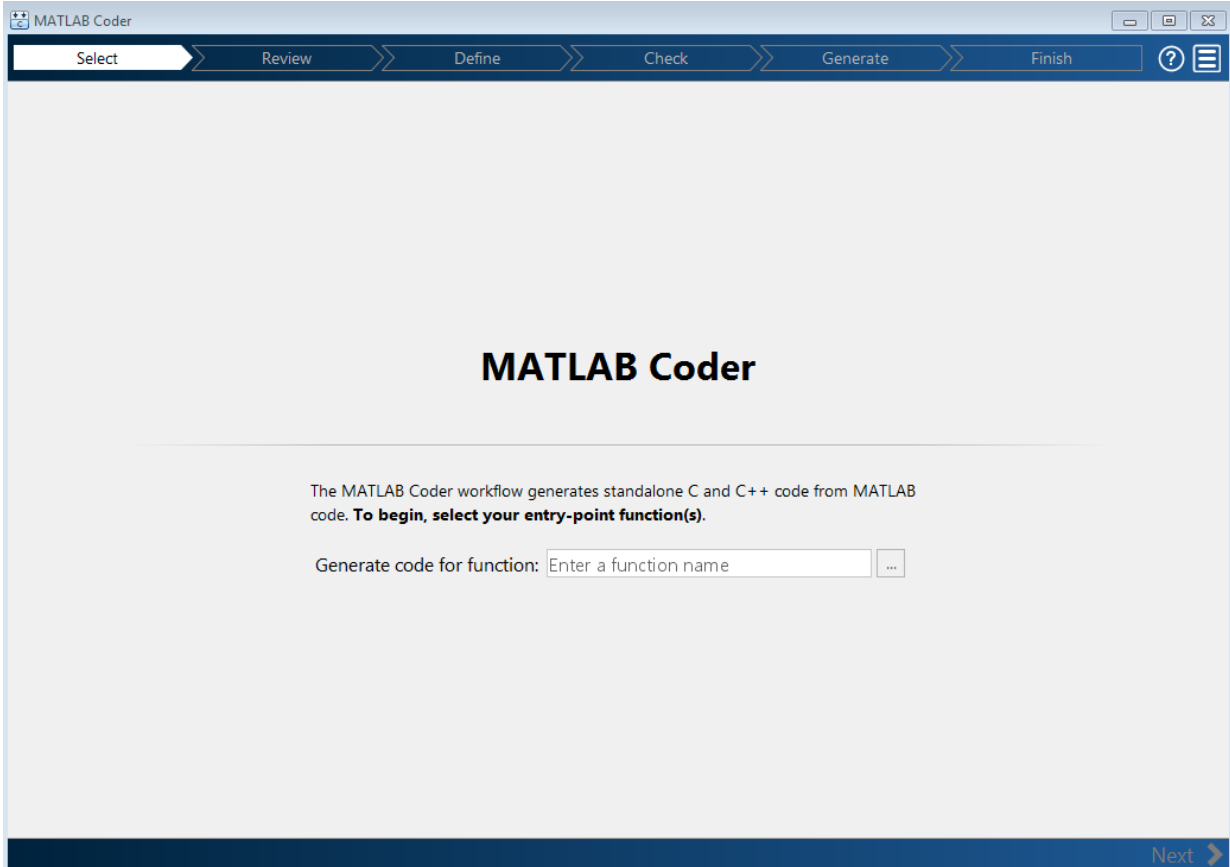
New Features

Bug Fixes

Compatibility Considerations

Improved MATLAB Coder app with integrated editor and simplified workflow

In R2015a, the MATLAB Coder app has a new user interface for the code generation workflow.



The improved app includes:

- Automatic checks for code generation readiness and run-time issues. The code generation readiness checks include identification of unsupported functions.
- An integrated editor to fix issues in your MATLAB code without leaving the app.
- A project summary and access to generated files.

- Export of project settings in the form of a MATLAB script.
- Help for each step and links to documentation for more information.

See C Code Generation Using the MATLAB Coder App.


Generation of example C/C++ main for integration of generated code into an application

In R2015a, you can generate an example C/C++ main function when generating source code, a static library, a dynamic library, or an executable. You modify the example main function to meet the requirements of your application.

An example main function provides a template that helps you incorporate generated code into your application. The template shows how to initialize function input arguments to zero and call entry-point functions. Generating an example main function is especially useful when the code uses dynamic memory allocation for data. See [Use an Example C Main in an Application](#).

By default, MATLAB Coder generates an example main function when generating source code, a static library, a dynamic library, or an executable.

To control generation of an example main function using the MATLAB Coder app:

- 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, 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 **All Settings** tab, under **Advanced**, set **Generate example main** to one of the following:
 - Do not generate an example main function
 - Generate, but do not compile, an example main function (default)

- Generate and compile an example main function

To control generation of an example main function using the command-line interface:

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

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

- 2 Set the `GenerateExampleMain` property to one of the following:

- 'DoNotGenerate'
- 'GenerateCodeOnly' (default)
- 'GenerateCodeAndCompile'

For example:

```
cfg.GenerateExampleMain = 'GenerateCodeOnly';
```

Better preservation of MATLAB variable names in generated code

To reduce memory usage, when possible, variables share names and memory in the generated code. In previous releases, this variable reuse optimization reused your variable names for other variables or replaced your variable names with the names of other variables. In R2015a, by default, this optimization preserves your variable names—it does not replace or reuse them. Other optimizations, however, can remove your variable names from the generated code. See [Variable Reuse in Generated Code](#).

Compatibility Considerations

If your MATLAB code uses large arrays or structures, in some cases, the extra memory to preserve your variable names can affect performance. To reduce memory usage, specify that the variable reuse optimization does not have to preserve variable names:

- Using a project, in the Project Settings dialog box, on the **All Settings** tab, set **Preserve variable names** to `None`.
- Using the command-line interface, set the configuration object property `PreserveVariableNames` to `None`.

See [Reuse Large Arrays and Structures](#).

More efficient generated code for logical indexing

Code generated for logical array indexing is faster and uses less memory than in previous releases. For example, the generated code for the following function is more efficient than in previous releases.

```
function x = foo(x,N)
assert(all(size(x) == [1 100]))
x(x>N) = N;
```

In R2015a, you do not have to replace `x(x>N) = N` with a `for`-loop to improve performance.

Code generation for additional Image Processing Toolbox and Computer Vision System Toolbox functions

Image Processing Toolbox

- `bweuler`
- `bwlabel`
- `bwperim`
- `regionprops`
- `watershed`

See Image Processing in MATLAB.

Computer Vision System Toolbox

- `cameraMatrix`
- `cameraParameters`
- `extrinsics`
- `opticalFlow`
- `opticalFlowHS`
- `opticalFlowLK`
- `opticalFlowLKDoG`
- `reconstructScene`

- `rectifyStereoImages`
- `stereoParameters`
- `triangulate`
- `undistortImage`
- `vision.DeployableVideoPlayer` on Mac platform.

In previous releases, `vision.DeployableVideoPlayer` supported code generation on Linux® and Windows platforms. In R2015a, `vision.DeployableVideoPlayer` also supports code generation on a Mac platform.

See Computer Vision System Toolbox.

Code generation for additional Communications System Toolbox, DSP System Toolbox, and Phased Array System Toolbox System objects

Communications System Toolbox

- `comm.CarrierSynchronizer`
- `comm.FMBroadcastDemodulator`
- `comm.FMBroadcastModulator`
- `comm.FMDemodulator`
- `comm.FMModulator`
- `comm.SymbolSynchronizer`

See Communications System Toolbox.

DSP System Toolbox

- `iirparameq`
- `dsp.HighpassFilter`
- `dsp.LowpassFilter`

See DSP System Toolbox.

Phased Array System Toolbox

- `pilotcalib`

- `phased.UCA`
- `phased.MFSKWaveform`

See Phased Array System Toolbox

Code generation for additional Statistics and Machine Learning Toolbox functions

- `betafit`
- `betalike`
- `pca`
- `pearsrnd`

See Statistics and Machine Learning Toolbox.

Code generation for additional MATLAB functions

Linear Algebra

- `bandwidth`
- `isbanded`
- `isdiag`
- `istril`
- `istriu`
- `lsqnonneg`

See Linear Algebra in MATLAB.

Statistics in MATLAB

- `cummin`
- `cummax`

See Statistics in MATLAB

Code generation for additional MATLAB function options

- `dimension` option for `cumsum` and `cumprod`

See [Functions and Objects Supported for C and C++ Code Generation — Alphabetical List](#).

Conversion from project to MATLAB script using MATLAB Coder app

In previous releases, to convert a project to a MATLAB script, you used the `-tocode` option of the `coder` command. In R2015a, you can also use the MATLAB Coder app to convert a project to a script. Before you convert a project to a script, complete the **Define Input Types** step.

To convert a project to a script using the MATLAB Coder app, on the workflow bar, click



, and then select **Convert to script**.

See [Convert MATLAB Coder Project to MATLAB Script](#).

Improved recognition of compile-time constants

In previous releases, the code generation software recognized that structure fields or array elements were constant only when all fields or elements were constant. In R2015a, in some cases, the software can recognize constant fields or constant elements even when some structure fields or array elements are not constant.

For example, consider the following code. Field `s.a` is constant and field `s.b` is not constant:

```
function y = create_array(x)
s.a = 10;
s.b = x;
y = zeros(1, s.a);
```

In previous releases, the software did not recognize that field `s.a` was constant. In the generated code, if variable-sizing was enabled, `y` was a variable-size array. If variable-sizing was disabled, the code generation software reported an error. In R2015a, the software recognizes that `s.a` is a constant. `y` is a static row vector with 10 elements.

As a result of this improvement, you can use individual assignments to assign constant values to structure fields. For example:

```
function y = mystruct(x)
s.a = 3;
s.b = 4;
y = zeros(s.a,s.b);
```

In previous releases, the software recognized the constants only if you defined the complete structure using the `struct` function: For example:

```
function y = mystruct(x)
s = struct('a', 3, 'b', 4);
y = zeros(s.a,s.b);
```

In some cases, the code generation software cannot recognize constant structure fields or array elements. See [Code Generation for Constants in Structures and Arrays](#).

Compatibility Considerations

The improved recognition of constant fields and elements can cause the following differences between code generated in R2015a and code generated in previous releases:

- A function output can be more specific in R2015a than it was in previous releases. An output that was complex in previous releases can be real in R2015a. An array output that was variable-size in previous releases can be fixed-size in R2015a.
- Some branches of code that are present in code generated using previous releases are eliminated from the generated code in R2015a.

Improved `emxArray` interface function generation

When you generate code that uses variable-size data, MATLAB Coder exports functions that you can use to create and interact with `emxArrays` in your generated code. R2015a includes the following improvements to `emxArray` interface functions:

`emxArray` interface functions for variable-size arrays that external C/C++ functions use

When you use `coder.ceval` to call an external C/C++ function, MATLAB Coder generates `emxArray` interface functions for the variable-size arrays that the external function uses.

Functions to initialize output emxArrays and emxArrays in structure outputs

MATLAB Coder generates functions to initialize emxArrays that are outputs or emxArrays that are in structure outputs.

A function that creates an empty emxArray on the heap has a name of the form:

```
emxInitArray_<baseType>
```

<baseType> is the type of the elements of the emxArray. The inputs to this function are a pointer to an emxArray pointer and the number of dimensions. For example:

```
void emxInitArray_real_T(emxArray_real_T **pEmxArray, int numDimensions);
```

A function that creates empty emxArrays in a structure has a name of the form:

```
void emxInitArray_<structType>
```

<structType> is the type of the structure. The input to this function is a pointer to the structure that contains the emxArrays. For example:

```
void emxInitArray_cstruct0_T(cstruct0_T *structure);
```

MATLAB Coder also generates functions that free the dynamic memory that the functions that create the emxArrays allocate. For example, the function that frees dynamic memory that emxInitArray_real_T allocates is:

```
void emxDestroyArray_real_T(emxArray_real_T *emxArray)
```

The function that frees dynamic memory that emxInitArray_cstruct0_T allocates is:

```
void emxDestroyArray_struct0_T(struct0_T *structure)
```

See C Code Interface for Arrays.

External definition of a structure that contains emxArrays

In previous releases, MATLAB Coder did not allow external definition of a structure that contained emxArrays. If you defined the structure in C code and declared it in an external header file, MATLAB Coder reported an error.

In R2015a, MATLAB Coder allows external definition of a structure that contains `emxArrays`. However, do not define the type of the `emxArray` in the external C code. MATLAB Coder defines the types of the `emxArrays` that a structure contains.

Code generation for casts to and from types of variables declared using `coder.opaque`

For code generation, you can use the MATLAB `cast` function to cast a variable to or from a variable that is declared using `coder.opaque`. Use `cast` with `coder.opaque` only for numeric types.

To cast a variable declared by `coder.opaque` to a MATLAB type, you can use the `B = cast(A, type)` syntax. For example:

```
x = coder.opaque('size_t', '0');
x1 = cast(x, 'int32');
```

You can also use the `B = cast(A, 'like', p)` syntax. For example:

```
x = coder.opaque('size_t', '0');
x1 = cast(x, 'like', int32(0));
```

To cast a MATLAB variable to the type of a variable declared by `coder.opaque`, you must use the `B = cast(A, 'like', p)` syntax. For example:

```
x = int32(12);
x1 = coder.opaque('size_t', '0');
x2 = cast(x, 'like', x1);
```

Use `cast` with `coder.opaque` to generate the correct data types for:

- Inputs to C/C++ functions that you call using `coder.ceval`.
- Variables that you assign to outputs from C/C++ functions that you call using `coder.ceval`.

Without this casting, it is possible to receive compiler warnings during code generation.

Consider this MATLAB code:

```
yt = coder.opaque('size_t', '42');
yt = coder.ceval('foo');
y = cast(yt, 'int32');
```

- `coder.opaque` declares that `yt` has C type `size_t`.
- `y = cast(yt, 'int32')` converts `yt` to `int32` and assigns the result to `y`.

Because `y` is a MATLAB numeric type, you can use `y` as you would normally use a variable in your MATLAB code.

The generated code looks like:

```
size_t yt= 42;
int32_T y;
y = (int32_T)yt;
```

It is possible that the explicit cast in the generated code prevents a compiler warning.

Generation of reentrant code without an Embedded Coder license

In previous releases, generation of reentrant code required an Embedded Coder license. In R2015a, you can generate reentrant code using MATLAB Coder without an Embedded Coder license.

See Reentrant Code.

Code generation for parfor-loops with stack overflow

In previous releases, you could not generate code for `parfor`-loops that contained variables that did not fit on the stack. In R2015a, you can generate code for these `parfor`-loops. See Algorithm Acceleration Using Parallel for-Loops (`parfor`).

Change in default value of the PassStructByReference code configuration object property

The `PassStructByReference` code configuration object property controls whether the `codegen` command generates pass by reference or pass by value structures for entry-point input and output structures.

In previous releases, the default value of `PassStructByReference` was `false`. By default, `codegen` generated pass by value structures. This default behavior differed from the MATLAB Coder app default behavior. The app generated pass by reference structures.

In R2015a, the value of `PassStructByReference` is `true`. By default, `codegen` generates pass by reference structures. The default behavior now matches the default behavior of the MATLAB Coder app.

See [Pass Structure Arguments by Reference or by Value](#).

Compatibility Considerations

For an entry-point function with structure arguments, if the `PassStructByReference` property has the default value, `codegen` generates a different function signature in R2015a than in previous releases.

Here is an example of a function signature generated in R2015a using the `codegen` command with the `PassStructByReference` property set to the default value, `true`:

```
void my_struct_in(const struct0_T *s, double y[4])
```

`my_struct_in` passes the input structure `s` by reference.

The signature for the same function generated in previous releases, using the `codegen` command with the `PassStructByReference` property set to the default value, `false` is:

```
void my_struct_in(const struct0_T s, double y[4])
```

`my_struct_in` passes the input structure `s` by value.

To control whether `codegen` generates pass by reference or pass by value structures, set the `PassStructByReference` code configuration object property. For example, to generate pass by value structures:

```
cfg = coder.config('lib');  
cfg.PassStructByReference = false;
```

Change in GLOBALS variable in scripts generated from a project

A script generated from a MATLAB Coder project that uses global variables creates the variable `GLOBALS`. In previous releases, `GLOBALS` stored the types of global variables. The initial values of the global variables were specified directly in the `codegen` command. In R2015a, `GLOBALS` stores both the types and the initial values of global variables. The `codegen` command obtains the initial values from `GLOBALS`.

See [Convert MATLAB Coder Project to MATLAB Script](#).

Target build log display for command-line code generation when hyperlinks disabled

In previous releases, if hyperlinks were disabled, you could not access the code generation report to view compiler or linker messages in the target build log. In R2015a, when hyperlinks are disabled, you see the target build log in the Command Window.

If you use the `-nojvm` startup option when you start MATLAB, hyperlinks are disabled. See [Commonly Used Startup Options](#).

For more information about the target build log, see [View Target Build Information](#).

Removal of instrumented MEX output type

You can no longer specify the output type `Instrumented MEX`.

Compatibility Considerations

For manual fixed-point conversion, use the command-line workflow. This workflow uses the Fixed-Point Designer functions `buildInstrumentedMex` and `showInstrumentationResults`. See [Manually Convert a Floating-Point MATLAB Algorithm to Fixed Point in the Fixed-Point Designer documentation](#).

Truncation of long enumerated type value names that include the class name prefix

In previous releases, when the code generation software determined the length or uniqueness of a generated enumerated type value name, it ignored the class name prefix. If you specified that a generated enumerated type value name included the class name prefix, it is possible that the generated type value name:

- Exceeded the maximum identifier length that you specified.
- Was the same as another identifier.

In R2015a, if you specify that a generated enumerated type value name includes the class name prefix, the generated type value name:

- Does not exceed the maximum identifier length.
- Is unique.

Compatibility Considerations

For a long type value name that includes the class name prefix, the name generated in previous releases can be different from the name generated in R2015a. For example, consider the enumerated type:

```
classdef Colors < int32
    enumeration
        Red (1)
        Green678911234567892123456789312 (2)
    end
    methods (Static)
        function p = addClassNameToEnumNames()
            p = true;
        end
    end
end
```

Suppose that the maximum identifier length is the default value, 31. In previous releases, the generated name for the enumerated value `Green678911234567892123456789312` was `Colors_Green678911234567892123456789312`. The length of the name exceeded 31 characters. In R2015a, the truncated name is 31 characters. Assuming that the generated name does not clash with another name, the name in R2015a is `Colors_Green6789112345678921234`. External code that uses the long name generated in the previous release cannot interface with the code generated in R2015a.

To resolve this issue, if possible, increase the maximum identifier length:

- At the command line, set `MaxIdLength`.
- In the MATLAB Coder app, in the project build settings, on the **Code Appearance** tab, set **Maximum identifier length**.

Fixed-point conversion enhancements

Support for multiple entry-point functions

Fixed-point conversion now supports multiple entry-point functions. You can generate C/C++ library functions to integrate with larger applications.

Support for global variables

You can now convert MATLAB algorithms that contain global variables to fixed-point code without modifying your MATLAB code.

Code coverage-based translation

During fixed-point conversion, MATLAB Coder now detects dead and constant folded code. It warns you if any parts of your code do not execute during the simulation of your test file. This detection can help you verify if your test file is testing the algorithm over the intended operating range. The software uses this code coverage information during the translation of your code from floating-point MATLAB code to fixed-point MATLAB code. The software inserts inline comments in the fixed-point code to mark the dead and untranslated regions. It includes the code coverage information in the generated fixed-point conversion HTML report.

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.

Automated fixed-point conversion of additional DSP System Toolbox objects

If you have a DSP System Toolbox™ license, you can now convert the following DSP System Toolbox System objects to fixed-point:

- `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. Using the MATLAB Coder app, 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.

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 prelookup step and reducing the use of multipliers in the data path. This interpolation method is available from the command-line workflow, and in the **Function Replacements** tab of the Fixed-Point Conversion step.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2014b

Version: 2.7

New Features

Bug Fixes

Compatibility Considerations

Code generation for additional Image Processing Toolbox and Computer Vision System Toolbox functions

Image Processing Toolbox

<code>bwdist</code>	<code>imadjust</code>	<code>intlut</code>	<code>ordfilt2</code>
<code>bwtraceboundary</code>	<code>imclearborder</code>	<code>iptcheckmap</code>	<code>rgb2ycbcr</code>
<code>fitgeotrans</code>	<code>imlincomb</code>	<code>medfilt2</code>	<code>stretchlim</code>
<code>histeq</code>	<code>imquantize</code>	<code>multithresh</code>	<code>ycbcr2rgb</code>

For the list of Image Processing Toolbox functions supported for code generation, see Image Processing Toolbox.

Computer Vision System Toolbox

- `bboxOverlapRatio`
- `selectStrongestBbox`
- `vision.DeployableVideoPlayer` on Linux

For the list of Computer Vision System Toolbox functions supported for code generation, see Computer Vision System Toolbox.

Code generation for additional Communications System Toolbox and DSP System Toolbox functions and System objects

Communications System Toolbox

- `iqcoef2imbal`
- `iqimbal2coef`
- `comm.IQImbalanceCompensator`

For the list of Communications System Toolbox™ functions supported for code generation, see Communications System Toolbox.

DSP System Toolbox

- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`

- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`

You cannot generate code directly from this System object. You can use the `generateFilteringCode` method to generate a MATLAB function. You can generate C/C++ code from this MATLAB function.

- `dsp.FIRDecimator` for transposed structure
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.PeakToPeak`
- `dsp.PeakToRMS`
- `dsp.PhaseExtractor`
- `dsp.SampleRateConverter`
- `dsp.StateLevels`

For the list of DSP System Toolbox functions and System objects supported for code generation, see DSP System Toolbox.

Code generation for enumerated types based on built-in MATLAB integer types

In previous releases, enumeration types were based on `int32`. In R2014b, you can base an enumerated type on one of the following built-in MATLAB integer data types:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`

You can use the base type to control the size of the enumerated type in the generated code. You can choose a base type to:

- Represent an enumerated type as a fixed-size integer that is portable to different targets.

- Reduce memory usage.
- Interface to legacy code.
- Match company standards.

The base type determines the representation of the enumerated types in the generated C and C++ code. For the base type `int32`, the code generation software generates a C enumeration type. For example:

```
enum LEDcolor
{
    GREEN = 1,
    RED
};

typedef enum LEDcolor LEDcolor;
```

For the other base types, the code generation software generates a `typedef` statement for the enumerated type and `#define` statements for the enumerated values. For example:

```
typedef short LEDColor;
#define GREEN ((LEDColor)1)
#define RED((LEDColor)2)
```

See Enumerated Types Supported for Code Generation.

Code generation for function handles in structures

You can now generate code for structures containing fields that are function handles. See Function Handle Definition for Code Generation.

Change in enumerated type value names in generated code

In previous releases, by default, the enumerated type value name in the generated code included a class name prefix, for example, `LEDcolor_GREEN`. In R2014b, by default, the generated enumerated type value name does not include the class name prefix. To generate enumerated type value names that include the class name prefix, in the enumerated type definition, modify the `addClassNameToEnumNames` method to return `true` instead of `false`:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
```

```
        GREEN(1),  
        RED(2)  
    end  
  
    methods(Static)  
        function y = addClassNameToEnumNames()  
            y = true;  
        end  
    end  
end
```

See Control Names of Enumerated Type Values in Generated Code.

Compatibility Considerations

The name of an enumerated type value in code generated using previous releases differs from the name generated using R2014b. If you have code that uses one of these names, modify the code to use the R2014b name or generate the name so that it matches the name from a previous release. If you want an enumerated type value name generated in R2014b to match the name from a previous release, in the enumerated types definition, modify the `addClassNameToEnumNames` method to return `true` instead of `false`.

Code generation for ode23 and ode45 ordinary differential equation solvers

- `ode23`
- `ode45`
- `odeget`
- `odeset`

See Numerical Integration and Differentiation in MATLAB.

Code generation for additional MATLAB functions

Data and File Management in MATLAB

- `feof`
- `frewind`

See Data and File Management in MATLAB.

Linear Algebra in MATLAB

- `ishermitian`
- `issymmetric`

See Linear Algebra in MATLAB.

String Functions in MATLAB

`str2double`

See String Functions in MATLAB.

Code generation for additional MATLAB function options

- `'vector'` and `'matrix'` eigenvalue options for `eig`
- All output class options for `sum` and `prod`
- All output class options for `mean` except `'native'` for integer types
- Multidimensional array support for `flipud`, `fliplr`, and `rot90`
- Dimension to operate along option for `circshift`

See Functions and Objects Supported for C and C++ Code Generation — Alphabetical List.

Collapsed list for inherited properties in code generation report

The code generation report displays inherited object properties on the **Variables** tab. In R2014b, the list of inherited properties is collapsed by default.

Change in length of exported identifiers

In previous releases, the code generation software limited exported identifiers, such as entry-point function names or `emxArray` utility function names, to a maximum length defined by the maximum identifier length setting. If the truncation of identifiers resulted in different functions having identical names, the code generation failed. In R2014b, for exported identifiers, the code generation software uses the entire generated identifier,

even if its length exceeds the maximum identifier length setting. If, however, the target C compiler has a maximum identifier length that is less than the length of the generated identifier, the target C compiler truncates the identifier.

Compatibility Considerations

Unless the target C compiler has a maximum identifier length that equals the length of a truncated exported identifier from a previous release, the identifier from the previous release does not match the identifier that R2014b generates. For example, suppose the maximum identifier length setting has the default value 31 and the target C compiler has a maximum identifier length of 255. Suppose that in R2014b, the code generation software generates the function `emxCreateWrapperND_StructType_123` for an unbounded variable-size structure array named `StructType_123`. In previous releases, the same function had the truncated name `emxCreateWrapperND_StructType_1`. In this example, code that previously called `emxCreateWrapperND_StructType_1` must now call `emxCreateWrapperND_StructType_123`.

Intel Performance Primitives (IPP) platform-specific code replacement libraries for cross-platform code generation

In R2014b, you can select an Intel® Performance Primitive (IPP) code replacement library for a specific platform. You can generate code for a platform that is different from the host platform that you use for code generation. The new code replacement libraries are:

- Intel IPP for x86-64 (Windows)
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)
- Intel IPP for x86/Pentium (Windows)
- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)
- Intel IPP for x86-64 (Linux)
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)

In a MATLAB Coder project that you create in R2014b, you can no longer select these libraries:

- Intel IPP
- Intel IPP/SSE with GNU99 extensions

If, however, you open a project from a previous release that specifies Intel IPP or Intel IPP/SSE with GNU99 extensions, the library selection is preserved and that library appears in the selection list.

See [Choose a Code Replacement Library](#).

Fixed-point conversion enhancements

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

For a MATLAB Coder project that includes automated fixed-point conversion, you can use the `-tocode` option of the `coder` command to create a pair of scripts for fixed-point conversion and fixed-point code generation. You can use the scripts to repeat the project workflow in a command-line workflow. Before you convert the project to the scripts, you must complete the **Test Numerics** step of the fixed-point conversion process.

For example:

```
coder -tocode my_fixpt_proj -script myscript.m
```

This command generates two scripts:

- `myscript.m` contains the MATLAB commands to create a code configuration object and generate fixed-point C code from fixed-point MATLAB code. The code configuration object has the same settings as the project.
- `myscriptsuffix.m` contains the MATLAB commands to create a floating-point to fixed-point configuration object and generate fixed-point MATLAB code from the entry-point function. The floating-point to fixed-point configuration object has the same fixed-point conversion settings as the project. `suffix` is the generated fixed-point file name suffix specified by the project file.

If you do not specify the `-script` option, `coder` writes the scripts to the Command Window.

See [Convert Fixed-Point Conversion Project to MATLAB Scripts](#).

Lookup table approximations for unsupported functions

The Fixed-Point Conversion tool now provides an option to generate lookup table approximations for continuous and stateless functions in your original MATLAB code.

This capability is useful for handling functions that are not supported for fixed point. To replace a function with a generated lookup table, specify the function that you want to replace on the **Function Replacements** tab.

In the command-line workflow, use `coder.approximation` and the `coder.FixptConfig` configuration object `addApproximation` method.

See [Replacing Functions Using Lookup Table Approximations](#).

Enhanced plotting capabilities

The Fixed-Point Conversion tool now provides additional plotting capabilities. You can use these plotting capabilities during the testing phase to compare the generated fixed-point versions of your algorithms to the original floating-point versions.

Default plots

The default comparison plots now plot vector and matrix data in addition to scalar data.

Custom plotting functions

You can now specify your own custom plotting function. The Fixed-Point Conversion tool calls the function and, for each variable, passes in the name of the variable and the function that uses it, and the results of the floating-point and fixed-point simulations. Your function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

For example, function `customComparisonPlot(varInfo, floatVarVals, fixedPtVarVals)`.

To use a custom plot function, in the Fixed-Point Conversion tool, select **Advanced**, and then set **Custom plot function** to the name of your plot function.

In the command-line workflow, set the `coder.FixptConfig` configuration object `PlotFunction` property to the name of your plot function.

See [Custom Plot Functions](#).

Integration with Simulation Data Inspector

You can now use the Simulation Data Inspector for comparison plots. The Simulation Data Inspector provides the capability to inspect and compare logged simulation data for multiple runs. You can import and export logged data, customize the organization of your logged data, and create reports.

In the Fixed-Point Conversion tool, select **Advanced** and then set **Plot with Simulation Data Inspector** to **Yes**. See [Enable Plotting Using the Simulation Data Inspector](#).

When generating fixed-point code in the command-line workflow, set the `coder.FixptConfig` configuration object `PlotWithSimulationDataInspector` property to `true`.

Custom plotting functions take precedence over the Simulation Data Inspector. See [Enable Plotting Using the Simulation Data Inspector](#).

Automated fixed-point conversion for commonly used System objects in MATLAB including Biquad Filter, FIR Filter, and Rate converter

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

- `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.

Additional fixed-point conversion command-line options

You can now use the `codegen` function with the `-float2fixed` option to convert floating point to fixed point based on derived ranges as well as simulation ranges. For more information, see `coder.FixptConfig`.

Type proposal report

After running the Test Numerics step to verify the data type proposals, the tool provides a link to a type proposal report that shows the instrumentation results for the fixed-point simulation. This report includes:

- The fixed-point code generated for each function in your original MATLAB algorithm
- Fixed-point instrumentation results for each variable in these functions:
 - Simulation minimum value
 - Simulation maximum value
 - Proposed data type

Generated fixed-point code enhancements

The generated fixed-point code now:

- Avoids loss of range or precision in unsigned subtraction operations. When the result of the subtraction is negative, the conversion process promotes the left operand to a signed type.
- Handles multiplication of fixed-point variables by non fixed-point variables. In previous releases, the variable that did not have a fixed-point type had to be a constant.
- Avoids overflows when adding and subtracting non fixed-point variables and fixed-point variables.
- Avoids loss of range when concatenating arrays of fixed-point numbers using `vertcat` and `horzcat`.

If you concatenate matrices, the conversion tool uses the largest `numericType` among the expressions of a row and casts the leftmost element to that type. This type is then used for the concatenated matrix to avoid loss of range.

- If the function that you are converting has a scalar input, and the `mpower` exponent input is not constant, the conversion tool sets `fimath ProductMode` to

Specify `Precision` in the generated code. With this setting, the output data type can be determined at compile time.

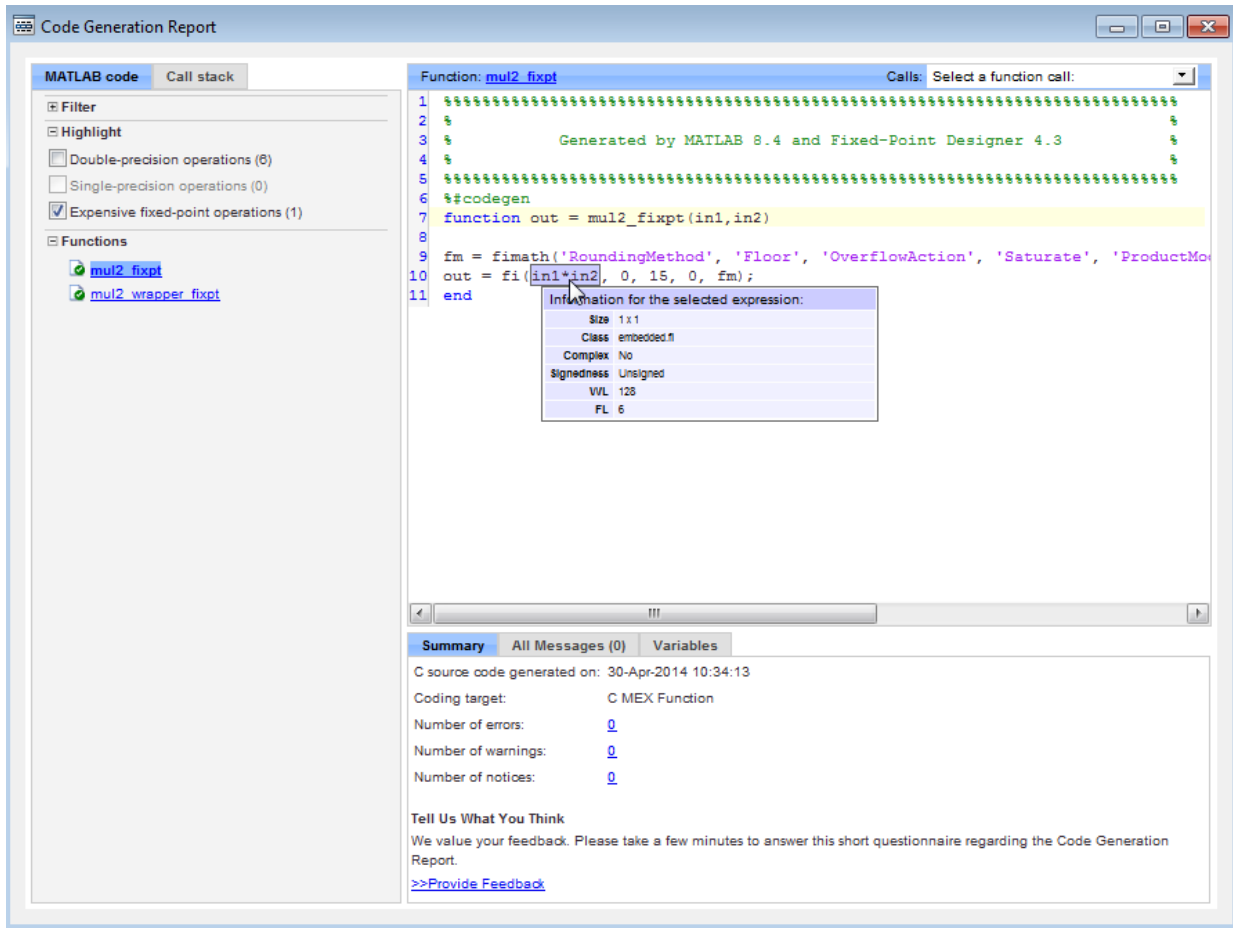
- Supports the following functions:
 - `true(m,n)`
 - `false(m,n)`
 - `sub2ind`
 - `mode`
 - `rem`
- Uses enhanced division replacement.

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

The tool now numbers function specializations sequentially in the **Function** list. In the generated fixed-point code, the number of each fixed-point specialization matches the number in the **Function** list which makes it easy to trace between the floating-point and fixed-point versions of your code. For example, the generated fixed-point function for the specialization of function `foo` named `foo > 1` is named `foo_s1`. For more information, see [Specializations](#).

Highlighting of potential data type issues in generated HTML report

You now have the option to 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 for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, or expensive rounding. The following example report highlights MATLAB code that requires expensive fixed-point operations.



The checks for the data type issues are disabled by default.

To enable the checks in a project:

- 1 In the Fixed-Point Conversion Tool, click **Advanced** to view the advanced settings.
- 2 Set **Highlight potential data type issues** to **Yes**.

To enable the checks at the command-line interface:

- 1 Create a floating-point to fixed-point conversion configuration object:

```
fxptcfg = coder.config('fixpt');
```

- 2** Set the `HighlightPotentialDataTypeIssues` property to `true`:

```
fxptcfg.HighlightPotentialDataTypeIssues = true;
```

See [Data Type Issues in Generated Code](#).

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2014a

Version: 2.6

New Features

Bug Fixes

Compatibility Considerations

Code generation for additional Image Processing Toolbox and Computer Vision System Toolbox functions

Image Processing Toolbox

<code>affine2d</code>	<code>im2uint16</code>	<code>imhist</code>
<code>bwpack</code>	<code>im2uint8</code>	<code>imopen</code>
<code>bwselect</code>	<code>imbothat</code>	<code>imref2d</code>
<code>bwunpack</code>	<code>imclose</code>	<code>imref3d</code>
<code>edge</code>	<code>imdilate</code>	<code>imtophat</code>
<code>getrangefromclass</code>	<code>imerode</code>	<code>imwarp</code>
<code>im2double</code>	<code>imextendedmax</code>	<code>mean2</code>
<code>im2int16</code>	<code>imextendedmin</code>	<code>projective2d</code>
<code>im2single</code>	<code>imfilter</code>	<code>strel</code>

See Image Processing Toolbox.

Computer Vision System Toolbox

- `detectHarrisFeatures`
- `detectMinEigenFeatures`
- `estimateGeometricTransform`

See Computer Vision System Toolbox.

Code generation for additional Signal Processing Toolbox, Communications System Toolbox, and DSP System Toolbox functions and System objects

Signal Processing Toolbox

- `findpeaks`
- `db2pow`
- `pow2db`

See Signal Processing Toolbox.

Communications System Toolbox

- `comm.OFDMModulator`
- `comm.OFDMDemodulator`

See Communications System Toolbox.

DSP System Toolbox

<code>ca2tf</code>	<code>firhalfband</code>	<code>ifir</code>	<code>iirnotch</code>
<code>cl2tf</code>	<code>firlpnorm</code>	<code>iircomb</code>	<code>iirpeak</code>
<code>firceqrip</code>	<code>firminphase</code>	<code>iirgrpdelay</code>	<code>tf2ca</code>
<code>fireqint</code>	<code>firnyquist</code>	<code>iirlpnorm</code>	<code>tf2cl</code>
<code>firgr</code>	<code>firpr2chfb</code>	<code>iirlpnormc</code>	<code>dsp.DCBlocker</code>

See DSP System Toolbox.

Code generation for fminsearch optimization function and additional interpolation functions in MATLAB**Optimization Functions in MATLAB**

- `fminsearch`
- `optimget`
- `optimset`

See Optimization Functions in MATLAB.

Interpolation and Computational Geometry in MATLAB

- `interp3`
- `mkpp`
- `pchip`
- `ppval`
- `spline`
- `unmkpp`
- `'spline'` and `'v5cubic'` interpolation methods for `interp1`

- 'spline' and 'cubic' interpolation methods for `interp2`

See [Interpolation and Computational Geometry in MATLAB](#).

Conversion from project to MATLAB script for command-line code generation

Using the `-tocode` option of the `coder` command, you can convert a MATLAB Coder project to the equivalent MATLAB code in a MATLAB script. The script reproduces the project in a configuration object and runs the `codegen` command. With this capability, you can:

- Move from a project workflow to a command-line workflow.
- Save the project as a text file that you can share.

The following command converts the project named `myproject` to the script named `myscript.m`:

```
coder -tocode myproject -script myscript.m
```

If you omit the `-script` option, the `coder` command writes the script to the Command Window.

See [Convert MATLAB Coder Project to MATLAB Script](#).

Code generation for `fread` function

In R2014a, you can generate code for the `fread` function.

See [Data and File Management in MATLAB](#).

Automatic C/C++ compiler setup

Previously, you used `mex -setup` to set up a compiler for C/C++ code generation. In R2014a, the code generation software locates and uses a supported installed compiler. You can use `mex -setup` to change the default compiler. See [Changing Default Compiler](#).

Compile-time declaration of constant global variables

You can specify that a global variable is a compile-time constant. Use a constant global variable to:

- Generate optimized code.
- Define the value of a constant without changing source code.

To declare a constant global variable in a MATLAB Coder project:

- 1 On the **Overview** tab, click **Add global**. Enter a name for the global variable.
- 2 Click the field to the right of the global variable name.
- 3 Select `Define Constant Value`.
- 4 Enter the value in the field to the right of the global variable name.

To declare a constant global variable at the command-line interface, use the `-globals` option along with the `coder.Constant` function.

In the following code, `gConstant` is a global variable with constant value 42.

```
cfg = coder.config('mex');
globals = {'gConstant', coder.Constant(42)};
codegen -config cfg myfunction -globals globals
```

See [Define Constant Global Data](#).

Enhanced code generation support for switch statements

Code generation now supports:

- Switch expressions and case expressions that are noninteger numbers, nonconstant strings, variable-size strings, or empty matrices
- Case expressions with mixed types and sizes

If all case expressions are scalar integer values, the code generation software generates a C `switch` statement. If at run time, the switch value is not an integer, the code generation software generates an error.

When the case expressions contain noninteger or nonscalar values, the code generation software generates `C if` statements in place of a `C switch` statement.

Code generation support for value classes with `set.prop` methods

In R2014a, you can generate code for value classes that have `set.prop` methods.

Code generation error for property that uses `AbortSet` attribute

Previously, when the current and new property values were equal, the generated code set the property value and called the `set` property method regardless of the value of the `AbortSet` attribute. When the `AbortSet` attribute was true, the generated code behavior differed from the MATLAB behavior.

In R2014a, if your code has properties that use the `AbortSet` attribute, the code generation software generates an error.

Compatibility Considerations

Previously, for code using the `AbortSet` attribute, code generation succeeded, but the behavior of the generated code was incorrect. Now, for the same code, code generation ends with an error. Remove the `AbortSet` attribute from your code and rewrite the code to explicitly compare the current and new property value.

Independent configuration selections for standard math and code replacement libraries

In R2014a, you can independently select and configure standard math and code replacement libraries for C and C++ code generation.

- The language selection (C or C++) determines the available standard math libraries.
 - In a project, the **Language** setting on the **All Settings** tab determines options that are available for a new **Standard math library** setting on the **Hardware** tab.
 - In a code configuration object, the `TargetLang` parameter determines options that are available for a new `TargetLangStandard` parameter.

- Depending on the your language selection, the following options are available for the **Standard math library** setting in a project and for the `TargetLangStandard` parameter in a configuration object.

Language	Standard Math Libraries (<code>TargetLangStandard</code>)
C	C89/C90 (ANSI) – default
	C99 (ISO)
C++	C89/C90 (ANSI) – default
	C99 (ISO)
	C++03 (ISO)

- The language selection and the standard math library selection determine the available code replacement libraries.
 - In a project, the **Code replacement library** setting on the **Hardware** tab lists available code replacement libraries. The MATLAB Coder software filters the list based on compatibility with the **Language** and **Standard math library** settings and the product licensing. For example, Embedded Coder offers more libraries and the ability to create and use custom code replacement libraries.
 - In a configuration object, the valid values for the `CodeReplacementLibrary` parameter depend on the values of the `TargetLang` and `TargetLangStandard` parameters and the product licensing.

Compatibility Considerations

In R2014a, code replacement libraries provided by MathWorks® no longer include standard math libraries.

- When you open a project that was saved with an earlier version:
 - The **Code replacement library** setting remains the same unless previously set to C89/C90 (ANSI), C99 (ISO), C++ (ISO), Intel IPP (ANSI), or Intel IPP (ISO). In these cases, MATLAB Coder software sets **Code replacement library** to None or Intel IPP.
 - MATLAB Coder software sets the new **Standard math library** setting to a value based on the previous **Code replacement library** setting.

If Code replacement library was set to:	Standard Math Library is set to:
C89/C90 (ANSI), C99 (ISO), or C++ (ISO)	C89/C90 (ANSI), C99 (ISO), C++03 (ISO), respectively
GNU99 (GNU), Intel IPP (ISO), Intel IPP (GNU), ADI TigerSHARC (Embedded Coder only), or MULTI BF53x (Embedded Coder only)	C99 (ISO)
A custom library (Embedded Coder), and the corresponding registration file has been loaded in memory	A value based on the BaseTfl property setting
Any other value	The default standard math library, C89/C90 (ANSI)

- When you load a configuration object from a MAT file that was saved in an earlier version:
 - The CodeReplacementLibrary setting remains the same unless previously set to Intel IPP (ANSI) or Intel IPP (ISO). In these cases, MATLAB Coder software sets CodeReplacementLibrary to Intel IPP.
 - MATLAB Coder software sets the new TargetLangStandard setting to a value based on the previous CodeReplacementLibrary setting.

If CodeReplacementLibrary was set to:	TargetLangStandard is set to:
Intel IPP (ANSI)	C89/C90 ANSI
Intel IPP (ISO)	C99 (ISO)
Any other value	The default standard math library, C89/C90 (ANSI)

- The generated code can differ from earlier versions if you use the default standard math library, C89/C90 (ANSI), with one of these code replacement libraries:
 - GNU99 (GNU)
 - Intel IPP (GNU)
 - ADI TigerSHARC (Embedded Coder only)
 - MULTI BF53x (Embedded Coder only)

To generate the same code as in earlier versions, change TargetLangStandard to C99 (ISO).

- After you open a project, if you select a code replacement library provided by MathWorks, the code generation software can produce different code than in previous versions, depending on the **Standard math library** setting. Verify generated code.
- If a script that you used in a previous version sets the configuration object `CodeReplacementLibrary` parameter, modify the script to use both the `CodeReplacementLibrary` and the `TargetLangStandard` parameters.

Restrictions on bit length for integer types in a `coder.HardwareImplementation` object

In R2014a, the code generation software imposes restrictions on the bit length of integer types in a `coder.HardwareImplementation` object. For example, the value of `ProdBitPerChar` must be between 8 and 32 and less than or equal to `ProdBitPerShort`. If you set the bit length to an invalid value, the code generation software reports an error.

See `coder.HardwareImplementation`.

Change in location of interface files in code generation report

The code generation software creates and uses interface files prefixed with `_coder`. For MEX code generation, these files appear in the code generation report. Previously, these files appeared in the **Target Source Files** pane of the **C code** tab of the code generation report. They now appear in the **Interface Source Files** pane of the **C code** tab. The report is now consistent with the folder structure for generated files. Since R2013b, the interface files are in a subfolder named **interface**.

Compiler warnings in code generation report

For MEX code generation, the code generation report now includes C and C++ compiler warning messages. If the code generation software detects compiler warnings, it generates a warning message in the **All Messages** tab. Compiler error and warning messages are highlighted in red on the **Target Build Log** tab.

See **View Errors and Warnings in a Report**.

Removal of date and time comment from generated code files

Previously, generated code files contained a comment with the string `C source code generated on` followed by a date and time stamp. This comment no longer appears in the generated code files. If you have an Embedded Coder license, you can include the date and time stamp in custom file banners by using code generation template (CGT) files.

Removal of two's complement guard from `rtwtypes.h`

`rtwtypes.h` no longer contains the following code:

```
#if ((SCHAR_MIN + 1) != -SCHAR_MAX)
#error "This code must be compiled using a 2's complement representation for signed integer values"
#endif
```

You must compile the code that is generated by the MATLAB Coder software on a target that uses a two's complement representation for signed integer values. The generated code does not verify that the target uses a two's complement representation for signed integer values.

Removal of `TRUE` and `FALSE` from `rtwtypes.h`

When the target language is C, `rtwtypes.h` defines `true` and `false`. It no longer defines `TRUE` and `FALSE`.

Compatibility Considerations

If you integrate code generated in R2014a with custom code that references `TRUE` or `FALSE`, modify your custom code in one of these ways:

- Define `TRUE` or `FALSE` in your custom code.
- Change `TRUE` and `FALSE` to `true` and `false`, respectively.
- Change `TRUE` and `FALSE` to `1U` and `0U`, respectively.

Change to default names for structure types generated from entry-point function inputs and outputs

In previous releases, the code generation software used the same default naming convention for structure types generated from local variables and from entry-point

function inputs and outputs. The software used `struct_T` for the first generated structure type name, `a_struct_T` for the next name, `b_struct_T` for the next name, and so on.

In R2014a, the code generation software uses a different default naming convention for structure types generated from entry-point function inputs and outputs. The software uses `struct0_T` for the first generated structure type name, `struct1_T` for the next name, `struct2_T` for the next name, and so on. With this new naming convention, you can more easily predict the structure type name in the generated code.

Compatibility Considerations

If you have C or C++ code that uses default structure type names generated from an entry-point function in a previous release, and you generate the entry-point function in R2014a, you must rewrite the code to use the new structure type names. However, subsequent changes to your MATLAB code, such as adding a variable with a structure type, can change the default structure type names in the generated code. To avoid compatibility issues caused by changes to default names for structure types in generated code, specify structure type names using `coder.cstructname`.

Toolbox functions supported for code generation

See [Functions and Objects Supported for C and C++ Code Generation — Alphabetical List](#) and [Functions and Objects Supported for C and C++ Code Generation — Categorical List](#).

Communications System Toolbox

- `comm.OFDMModulator`
- `comm.OFDMDemodulator`

Computer Vision System Toolbox

- `detectHarrisFeatures`
- `detectMinEigenFeatures`
- `estimateGeometricTransform`

Data and File Management in MATLAB

`fread`

DSP System Toolbox

ca2tf	firhalfband	ifir	iirnotch
cl2tf	firlpnorm	iircomb	iirpeak
firceqrip	firminphase	iirgrpdelay	tf2ca
fireqint	firnyquist	iirlpnorm	tf2cl
firgr	firpr2chfb	iirlpnormc	dsp.DCBlocker

Image Processing Toolbox

affine2d	im2uint16	imhist
bwpack	im2uint8	imopen
bwselect	imbothat	imref2d
bwunpack	imclose	imref3d
edge	imdilate	imtophat
getrangefromclass	imerode	imwarp
im2double	imextendedmax	mean2
im2int16	imextendedmin	projective2d
im2single	imfilter	strel

Interpolation and Computational Geometry in MATLAB

- interp2
- interp3
- mkpp
- pchip
- ppval
- polyarea
- rectint
- spline
- unmkpp

Matrices and Arrays in MATLAB

flip

Optimization Functions in MATLAB

- `fminsearch`
- `optimget`
- `optimset`

Polynomials in MATLAB

- `polyder`
- `polyint`
- `polyvalm`

Signal Processing Toolbox

- `findpeaks`
- `db2pow`
- `pow2db`

Fixed-point conversion enhancements

These capabilities require a Fixed-Point Designer license.

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](#).

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 for float-to-fixed conversion

In R2014a, when you convert floating-point MATLAB code to fixed-point C or 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`.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2013b

Version: 2.5

New Features

Bug Fixes

Compatibility Considerations

Code generation for Statistics Toolbox and Phased Array System Toolbox

Code generation now supports more than 100 Statistics Toolbox™ functions. For implementation details, see Statistics Toolbox Functions.

Code generation now supports most of the Phased Array System Toolbox™ functions and System objects. For implementation details, see Phased Array System Toolbox Functions and Phased Array System Toolbox System Objects.

Toolbox functions supported for code generation

For implementation details, see Functions Supported for C/C++ Code Generation — Alphabetical List.

Data Type Functions

- `narginchk`

Programming Utilities

- `mfilename`

Specialized Math

- `psi`

Computer Vision System Toolbox Classes and Functions

- `extractFeatures`
- `detectSURFFeatures`
- `disparity`
- `detectMSERFeatures`
- `detectFASTFeatures`
- `vision.CascadeObjectDetector`
- `vision.PointTracker`
- `vision.PeopleDetector`
- `cornerPoints`

- `MSEERegions`
- `SURFPoints`

parfor function for standalone code generation, enabling execution on multiple cores

You can use MATLAB Coder software to generate standalone C/C++ code from MATLAB code that contains `parfor`-loops. The code generation software uses the Open Multi-Processing (OpenMP) application interface to generate C/C++ code that runs in parallel on multiple cores on the target hardware.

For more information, see `parfor` and Accelerate MATLAB Algorithms That Use Parallel for-loops (`parfor`).

Persistent variables in parfor-loops

You can now generate code from parallel algorithms that use persistent variables.

For more information, see `parfor`.

Random number generator functions in parfor-loops

You can now generate code from parallel algorithms that use the random number generators `rand`, `randn`, `randi`, `randperm`, and `rng`.

For more information, see `parfor`.

External code integration using `coder.ExternalDependency`

You can define the interface to external code using the new `coder.ExternalDependency` class. Methods of this class update the compile and build information required to integrate the external code with MATLAB code. In your MATLAB code, you can call the external code without needing to update build information. See `coder.ExternalDependency`.

Updating build information using `coder.updateBuildInfo`

You can use the new function `coder.updateBuildInfo` to update build information. For example:

```
coder.updateBuildInfo('addLinkFlags','/STACK:1000000');
```

adds a stack size option to the linker command line. See `coder.updateBuildInfo`.

Generation of simplified code using built-in C types

By default, MATLAB Coder now uses built-in C types in the generated code. You have the option to use predefined types from `rtwtypes.h`. To control the data type in the generated code:

- In a project, on the Project Settings dialog box **Code Appearance** tab, use the **Data Type Replacement** setting.
- At the command line, use the configuration object parameter `DataTypeReplacement`.

The built-in C type that the code generation software uses depends on the target hardware.

For more information, see [Specify Data Type Used in Generated Code](#).

Compatibility Considerations

If you use the default configuration or project settings, the generated code has built-in C types such as `double` or `char`. Code generated prior to R2013b has predefined types from `rtwtypes.h`, such as `real_T` or `int32_T`.

Conversion of MATLAB expressions into C constants using `coder.const`

You can use the new function `coder.const` to convert expressions and function calls to constants at compile time. See `coder.const` and [Constant Folding](#).

Highlighting of constant function arguments in the compilation report

The compilation report now highlights constant function arguments and displays them in a distinct color. You can display the constant argument data type and value by placing

the cursor over the highlighted argument. You can export the constant argument value to the base workspace where you can display detailed information about the argument.

For more information, see [Viewing Variables in Your MATLAB Code](#).

Code Generation Support for `int64`, `uint64` data types

You can now use `int64` and `uint64` data types for code generation.

C99 long long integer data type for code generation

If your target hardware and 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 and multiword helper functions. To specify the long long data type for code generation:

- In a project, on the Project Settings dialog box **Hardware** tab, use the following production and test hardware settings:
 - **Enable long long:** Specify that your C compiler supports the long long data type. Set to `Yes` to enable **Sizes: long long**.
 - **Sizes: long long:** Describe length in bits of the C long long data type supported by the hardware.
- At the command line, use the following hardware implementation configuration object parameters:
 - `ProdLongLongMode`: Specify that your C compiler supports the long long data type. Set to `true` to enable `ProdBitPerLongLong`.
 - `ProdBitPerLongLong`: Describes the length in bits of the C long long data type supported by the production hardware.
 - `TargetLongLongMode`: Specifies whether your C compiler supports the long long data type. Set to `true` to enable `TargetBitPerLongLong`.
 - `TargetBitPerLongLong`: Describes the length in bits of the C long long data type supported by the test hardware.

For more information, see the class reference information for `coder.HardwareImplementation`.

Change to passing structures by reference

In R2013b, the option to pass structures by reference to entry-point functions in the generated code applies to function outputs and function inputs. In R2013a, this option applied only to inputs to entry-point functions.

Compatibility Considerations

If you select the pass structures by reference option, and a MATLAB entry-point function has a single output that is a structure, the generated C function signature in R2013b differs from the signature in R2013a. In R2013a, the generated C function returns the output structure. In R2013b, the output structure is a pass by reference function parameter.

If you have code that calls one of these functions generated in R2013a, and then you generate the function in R2013b, you must change the call to the function. For example, suppose *S* is a structure in the following MATLAB function `foo`.

```
function S = foo()
```

If you generate this function in R2013a, you call the function this way:

```
S = foo();
```

If you generate this function in R2013b, you call the function this way:

```
foo(&S);
```

`coder.runTest` new syntax

Use the syntax `coder.runTest(test_fcn, MEX_name_ext)` to run `test_fcn` replacing calls to entry-point functions with calls to the corresponding MEX functions in the MEX file named `MEX_name_ext`. `MEX_name_ext` includes the platform-specific file extension. See `coder.runTest`.

`coder.target` syntax change

The new syntax for `coder.target` is:

```
tf = coder.target('target')
```


For example, `coder.target('MATLAB')` returns true when code is running in MATLAB. See `coder.target`.

You can use the old syntax, but consider changing to the new syntax. The old syntax will be removed in a future release.

Changes for complex values with imaginary part equal to zero

In R2013b, complex values with an imaginary part equal to zero become real when:

- They are returned by a MEX function.
- They are passed to an extrinsic function.

See Expressions With Complex Operands Yield Complex Results.

Compatibility Considerations

MEX functions generated in R2013b return a real value when a complex result has an imaginary part equal to zero. MEX functions generated prior to R2013b return a complex value when a complex result has an imaginary part equal to zero.

In R2013b, complex values with imaginary part equal to zero become real when passed to an extrinsic function. In previous releases, they remain complex.

Subfolder for code generation interface files

Previously, interface files for MEX code generation appeared in the code generation output folder. In R2013b, these interface files have the prefix `_coder`, appear in a subfolder named `interface`, and appear for all code generation output types.

Support for LCC compiler on Windows 64-bit machines

The LCC-win64 compiler is shipping with MATLAB Coder for Microsoft® Windows 64-bit machines. For Windows 64-bit machines that do not have a third-party compiler installed, MEX code generation uses LCC by default.

You cannot use LCC for code generation of C/C++ static libraries, C/C++ dynamic libraries, or C/C++ executables. For these output types, you must install a compiler. See http://www.mathworks.com/support/compilers/current_release/.

Fixed-Point conversion enhancements

These capabilities require a Fixed-Point Designer license.

Fixed-Point conversion option for `codegen`

You can now convert floating-point MATLAB code to fixed-point code, and then generate C/C++ code at the command line using the option `-float2fixed` with the `codegen` command. See `codegen` and Convert Floating-Point MATLAB Code to Fixed-Point C Code Using `codegen`.

Fixed-point conversion using derived ranges on Mac platforms

You can now convert floating-point MATLAB code to fixed-point C code using the Fixed-Point Conversion tool in MATLAB Coder projects on Mac platforms.

For more information, see Automated Fixed-Point Conversion and Propose Fixed-Point Data Types Based on Derived Ranges.

Derived ranges for complex variables in MATLAB Coder projects

Using the Fixed-Point Conversion tool in MATLAB Coder projects, you can derive ranges for complex variables. For more information, see Propose Fixed-Point Data Types Based on Derived Ranges

Fixed-point conversion workflow supports designs that use enumerated types

Using the Fixed-Point Conversion tool in MATLAB Coder projects, you can 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.

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](#).

Fixed-point conversion test file coverage results

The 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.

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

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2013a

Version: 2.4

New Features

Bug Fixes

Compatibility Considerations

Automatic fixed-point conversion during code generation (with Fixed-Point Designer)

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 Fixed-Point Designer 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](#).

File I/O function support

The following file I/O functions are now supported for code generation:

- `fclose`
- `fopen`
- `fprintf`

To view implementation details, see [Functions Supported for Code Generation — Alphabetical List](#).

Support for nonpersistent handle objects

You can now generate code for local variables that contain references to handle objects or System objects. In previous releases, generating code for these objects was limited to objects assigned to persistent variables.

Structures passed by reference to entry-point functions

You can now specify to pass structures by reference to entry-point functions in the generated code. This optimization is available for standalone code generation only; it is not available for MEX functions. Passing structures by reference reduces the number of copies at entry-point function boundaries in your generated code. It does not affect how structures are passed to functions other than entry-point functions.

To pass structures by reference:

- In a project, on the Project Settings dialog box **All Settings** tab, under **Advanced**, set **Pass structures by reference to entry-point functions** to Yes.
- At the command line, create a code generation configuration object and set the `PassStructByReference` parameter to `true`. For example:

```
cfg = coder.config('lib');  
cfg.PassStructByReference=true;
```

Include custom C header files from MATLAB code

The `coder.cinclude` function allows you to specify in your MATLAB code which custom C header files to include in the generated C code. Each header file that you specify using `coder.cinclude` is included in every C/C++ file generated from your MATLAB code. You can specify whether the `#include` statement uses double quotes for application header files or angle brackets for system header files in the generated code.

For example, the following code for function `foo` specifies to include the application header file `mystruct.h` in the generated code using double quotes.

```
function y = foo(x1, x2)  
%#codegen  
coder.cinclude('mystruct.h');
```

...

For more information, see `coder.cinclude`.

Load from MAT-files

MATLAB Coder now supports a subset of the `load` function for loading run-time values from a MAT-file while running a MEX function. It also provides a new function, `coder.load`, for loading compile-time constants when generating MEX or standalone code. 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 Generation — Alphabetical List](#).

For more information, see `coder.load`.

`coder.opaque` function enhancements

When you use `coder.opaque` to declare a variable in the generated C code, you can now also specify the header file that defines the type of the variable. Specifying the location of the header file helps to avoid compilation errors because the MATLAB Coder software can find the type definition more easily.

You can now compare `coder.opaque` variables of the same type. This capability helps you verify, for example, whether an `fopen` command succeeded.

```
null = coder.opaque('FILE*', 'NULL', 'HeaderFile', 'stdio.h');
ftmp = null;
ftmp = coder.ceval('fopen', fname, permission);
if ftmp == null
    % Error - file open failed
end
```

For more information, see `coder.opaque`.

Automatic regeneration of MEX functions in projects

When you run a test file from a MATLAB Coder project to verify the behavior of the generated MEX function, the project now detects when to rebuild the MEX function. MATLAB Coder rebuilds the MEX function only if you have modified the original

MATLAB algorithm since the previous build, saving you time during the verification phase.

MEX function signatures include constant inputs

When you generate a MEX function for a MATLAB function that takes constant inputs, by default, the MEX function signature now contains the constant inputs. If you are verifying your MEX function in a project, this behavior allows you to use the same test file to run the original MATLAB algorithm and the MEX function.

Compatibility Considerations

In previous releases, MATLAB Coder removed the constants from the MEX function signature. To use these existing scripts with MEX functions generated using R2013a software, do one of the following:

- Update the scripts so that they no longer remove the constants.
- Configure MATLAB Coder to remove the constant values from the MEX function signature.

To configure MATLAB Coder to remove the constant values:

- In a project, on the Project Settings dialog box **All Settings** tab, under **Advanced**, set **Constant Inputs** to Remove from MEX signature.
- At the command line, create a code generation configuration object, and, set the `ConstantInputs` parameter to 'Remove'. For example:

```
cfg = coder.config;  
cfg.ConstantInputs='Remove';
```

Custom toolchain registration

MATLAB Coder software enables you to register third-party software build tools for creating executables and libraries.

- The software automatically detects supported tool chains on your system.
- You can manage and customize multiple tool chain definitions.
- Before generating code, you can select any one of the definitions using a drop-down list.

- The software generates simplified makefiles for improved readability.

For more information:

- See Custom Toolchain Registration.
- See the Adding a Custom Toolchain example.

Compatibility Considerations

If you open a MATLAB Coder project or use a code generation configuration object from R2012b, the current version of MATLAB Coder software automatically tries to use the toolchain approach. If an existing project or configuration object does not use default target makefile settings, MATLAB Coder might not be able to upgrade to use a toolchain approach and will emit a warning. For more information, see [Project or Configuration is Using the Template Makefile](#).

Complex trigonometric functions

Code generation support has been added for complex `acosD`, `acotD`, `acscD`, `asecD`, `asinD`, `atanD`, `cosD`, `cscD`, `cotD`, `secD`, `sinD`, and `tanD` functions.

parfor function reduction improvements and C support

When generating MEX functions for `parfor`-loops, you can now use `intersect` and `union` as reduction functions, and the following reductions are now supported:

- Concatenations
- Arrays
- Function handles

By default, when MATLAB Coder generates a MEX function for MATLAB code that contains a `parfor`-loop, MATLAB Coder no longer requires C++ and now honors the target language setting.

Support for integers in number theory functions

Code generation supports integer inputs for the following number theory functions:

- `cumprod`
- `cumsum`
- `factor`
- `factorial`
- `gcd`
- `isprime`
- `lcm`
- `median`
- `mode`
- `nchoosek`
- `nextpow2`
- `primes`
- `prod`

To view implementation details, see [Functions Supported for Code Generation — Alphabetical List](#).

Enhanced support for class property initial values

If you initialize a class property, you can now assign a different type to the property when you use the class. For example, class `foo` has a property `prop1` of type `double`.

```
classdef foo %#codegen
    properties
        prop1= 0;
    end
    methods
        ...
    end
end
```

Function `bar` assigns a different type to `prop1`.

```
function bar %#codegen
    f=foo;
    f.prop1=single(0);
    ...
end
```

In R2013a, MATLAB Coder ignores the initial property definition and uses the reassigned type. In previous releases, MATLAB Coder did not support this reassignment and code generation failed.

Compatibility Considerations

In previous releases, if the reassigned property had the same type as the initial value but a different size, the property became variable-size in the generated code. In R2013a, MATLAB Coder uses the size of the reassigned property, and the size is fixed. If you have existing MATLAB code that relies on the property being variable-size, you cannot generate code for this code in R2013a. To fix this issue, do not initialize the property in the property definition block.

For example, you can no longer generate code for the following function `bar`.

Class `foo` has a property `prop1` which is a scalar double.

```
classdef foo %#codegen
    properties
        prop1= 0;
    end
    methods
        ...
    end
end
```

Function `bar` changes the size of `prop1`.

```
function bar %#codegen
    f=foo;
    f.prop1=[1 2 3];
    % Use f
    disp(f.prop1);
    f.prop1=[1 2 3 4 5 6];
```

Optimized generated code for `x=[x c]` when `x` is a vector

MATLAB Coder now generates more optimized code for the expression `x=[x c]`, if:

- `x` is a row or column vector.
- `x` is not in `c`.

- `x` is not aliased.
- There are no function calls in `c`.

In previous releases, the generated code contained multiple copies of `x`. In R2013a, it does not contain multiple copies of `x`.

This enhancement reduces code size and execution time. It also improves code readability.

Default use of Basic Linear Algebra Subprograms (BLAS) libraries

MATLAB Coder now uses BLAS libraries whenever they are available. There is no longer an option to turn off the use of these libraries.

Compatibility Considerations

If existing configuration settings disable BLAS, MATLAB Coder now ignores these settings.

Changes to compiler support

MATLAB Coder supports these new compilers.

- On Microsoft Windows platforms, Visual C++® 11.
- On Mac OS X platforms, Apple Xcode 4.2 with Clang.

MATLAB Coder no longer supports the `gcc` compiler on Mac OS X platforms.

MATLAB Coder no longer supports Watcom for standalone code generation. Watcom is still supported for building MEX functions.

Compatibility Considerations

- Because Clang is the only compiler supported on Mac OS X platforms, and Clang does not support Open MP, `parfor` is no longer supported on Mac OS X platforms.
- MATLAB Coder no longer supports Watcom for standalone code generation. Use Watcom only for building MEX functions. Use an alternative compiler for standalone

code generation. For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

New toolbox functions supported for code generation

To view implementation details, see [Functions Supported for Code Generation — Alphabetical List](#).

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`

Functions being removed

These functions have been removed from MATLAB Coder software.

Function Name	What Happens When You Use This Function?
<code>emlc</code>	Errors in R2013a.
<code>emlmex</code>	Errors in R2013a.

Compatibility Considerations

`emlc` and `emlmex` have been removed. Use `codegen` instead. If you have existing code that calls `emlc` or `emlmex`, use `coder.upgrade` to help convert your code to the new syntax.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2012b

Version: 2.3

New Features

Bug Fixes

parfor function support for MEX code generation, enabling execution on multiple cores

You can use MATLAB Coder software to generate MEX functions from MATLAB code that contains `parfor`-loops. The generated MEX functions can run on multiple cores on a desktop. For more information, see `parfor` and Acceleration of MATLAB Algorithms Using Parallel for-loops (`parfor`).

Code generation readiness tool

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions and an indication of how much work is needed to make the MATLAB code suitable for code generation.

For more information, see `coder.screener` and Code Generation Readiness Tool.

Reduced data copies and lightweight run-time checks for generated MEX functions

MATLAB Coder now eliminates data copies for built-in, non-complex data types. It also performs faster bounds checks. These enhancements result in faster generated MEX functions.

Additional string function support for code generation

The following string functions are now supported for code generation. To view implementation details, see Functions Supported for Code Generation — Alphabetical List.

- `deblank`
- `hex2num`
- `isletter`
- `isspace`
- `isstrprop`
- `lower`

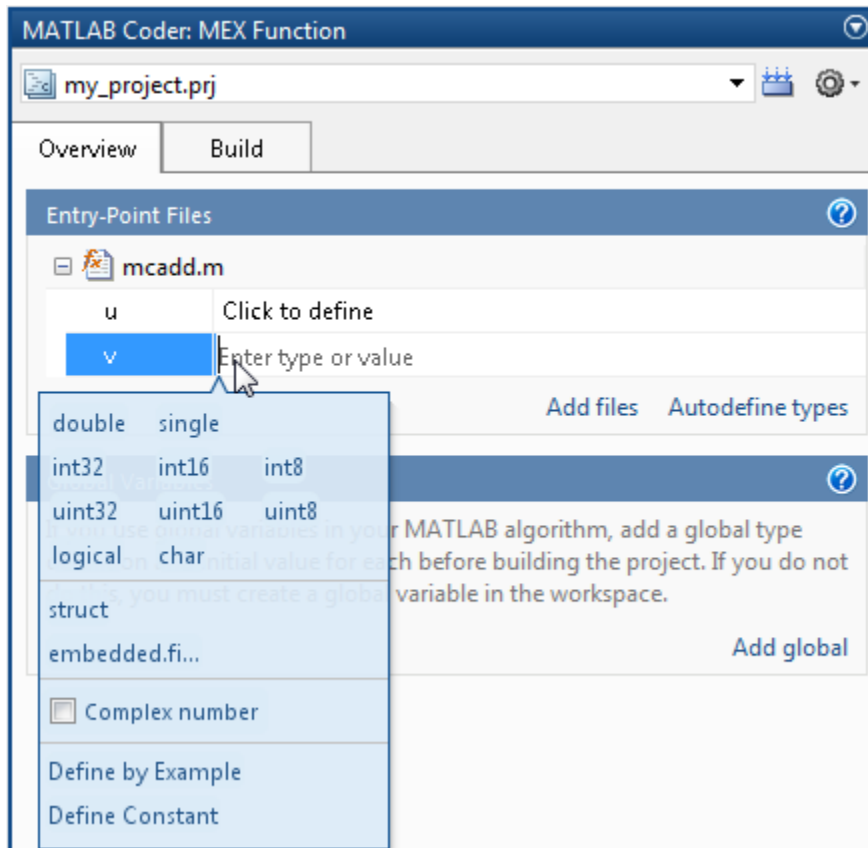
- `num2hex`
- `strcmpi`
- `strjust`
- `strncmp`
- `strncmpi`
- `strtok`
- `strtrim`
- `upper`

Visualization functions in generated MEX functions

The MATLAB Coder software now detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. For MEX code generation, MATLAB Coder automatically calls out to MATLAB for these functions. For standalone code generation, MATLAB Coder does not generate code for these visualization functions. This capability reduces the amount of time that you spend making your code suitable for code generation. It also removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

Input parameter type specification enhancements

The updated project user interface facilitates input parameter type specification.



Project import and export capability

You can now export project settings to a configuration object stored as a variable in the base workspace. You can then use the configuration object to import the settings into a different project or to generate code at the command line with the `codegen` function.

This capability allows you to:

- Share settings between the project and command-line workflow
- Share settings between multiple projects
- Standardize on settings for code generation projects

For more information, see [Share Build Configuration Settings](#).

Package generated code in zip file for relocation

The `packNGo` function packages generated code files into a compressed zip file so that you can relocate, unpack, and rebuild them in another development environment. This capability is useful if you want to relocate files so that you can recompile them for a specific target environment or rebuild them in a development environment in which MATLAB is not installed.

For more information, see [Package Code For Use in Another Development Environment](#).

Fixed-point instrumentation and data type proposals

MATLAB Coder projects provide the following fixed-point conversion support:

- Option to generate instrumented MEX functions
- Use of instrumented MEX functions to provide simulation minimum and maximum results
- Fixed-point data type proposals based on simulation minimum and maximum values
- Option to propose fraction lengths or word lengths

You can use these proposed fixed-point data types to create a fixed-point version of your original MATLAB entry-point function.

Note Requires a Fixed-Point Toolbox™ license.

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

New toolbox functions supported for code generation

To view implementation details, see [Functions Supported for Code Generation — Alphabetical List](#).

Computer Vision System Toolbox

- `integralImage`

Image Processing Toolbox

- `bwlookup`
- `bwmorph`

Interpolation and Computational Geometry

- `interp2`

Trigonometric Functions

- `atan2d`

New System objects supported for code generation

The following System objects are now supported for code generation. To see the list of System objects supported for code generation, see [System Objects Supported for Code Generation](#).

Communications System Toolbox

- `comm.ACPR`
- `comm.BCHDecoder`
- `comm.CCDF`
- `comm.CPMCarrierPhaseSynchronizer`
- `comm.GoldSequence`
- `comm.LDPCDecoder`
- `comm.LDPCEncoder`
- `comm.LTEMIMOChannel`
- `comm.MemorylessNonlinearity`
- `comm.MIMOChannel`
- `comm.PhaseNoise`
- `comm.PSKCarrierPhaseSynchronizer`
- `comm.RSDecoder`

DSP System Toolbox

- `dsp.AllpoleFilter`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.IIRFilter`
- `dsp.SignalSource`

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2012a

Version: 2.2

New Features

Compatibility Considerations

Code Generation for MATLAB Classes

In R2012a, there is preliminary support for code generation for MATLAB classes targeted at supporting System objects defined by users. For more information about generating code for MATLAB classes, see [Code Generation for MATLAB Classes](#). For more information about generating code for System objects, see the [DSP System Toolbox](#), [Computer Vision System Toolbox](#) or the [Communications System Toolbox](#) documentation.

Dynamic Memory Allocation Based on Size

By default, dynamic memory allocation is now enabled for variable-size arrays whose size exceeds a configurable threshold. This behavior allows for finer control over stack memory usage. Also, you can generate code automatically for more MATLAB algorithms without modifying the original MATLAB code.

Compatibility Considerations

If you use scripts to generate code and you do not want to use dynamic memory allocation, you must disable it. For more information, see [Controlling Dynamic Memory Allocation](#).

C/C++ Dynamic Library Generation

You can now use MATLAB Coder to build a dynamically linked library (DLL) from the generated C code. These libraries are useful for integrating into existing software solutions that expect dynamically linked libraries.

For more information, see [Generating C/C++ Dynamically Linked Libraries from MATLAB Code](#).

Automatic Definition of Input Parameter Types

MATLAB Coder software can now automatically define input parameter types by inferring these types from test files that you supply. This capability facilitates input type definition and reduces the risk of introducing errors when defining types manually.

To learn more about automatically defining types:

- In MATLAB Coder projects, see Autodefining Input Types.
- At the command line, see the `coder.getArgTypes` function reference page.

Verification of MEX Functions

MATLAB Coder now provides support for test files to verify the operation of generated MEX functions. This capability enables you to verify that the MEX function is functionally equivalent to your original MATLAB code and to check for run-time errors.

To learn more about verifying MEX function behavior:

- In MATLAB Coder projects, see How to Verify MEX Functions in a Project.
- At the command line, see the `coder.runTest` function reference page.

Enhanced Project Settings Dialog Box

The **Project Settings** dialog box now groups configuration parameters so that you can easily identify the parameters associated with code generation objectives such as speed, memory, and code appearance. The dialog boxes for code generation configuration objects, `coder.MexCodeConfig`, `coder.CodeConfig`, and `coder.EmbeddedCodeConfig`, also use the same new groupings.

To view the updated **Project Settings** dialog box:

- 1 In a project, click the **Build** tab.
- 2 On the **Build** tab, click the More settings link to open the **Project Settings** dialog box.

For information about the parameters on each tab, click the **Help** button.

To view the updated dialog boxes for the code generation configuration objects:

- 1 At the MATLAB command line, create a configuration object. For example, create a configuration object for MEX code generation.

```
mex_cfg = coder.config;
```

- 2 Open the dialog box for this object.

```
open mex_cfg
```

For information about the parameters on each tab, click the **Help** button.

Projects Infer Input Types from assert Statements in Source Code

MATLAB Coder projects can now infer input data types from `assert` statements that define the properties of function inputs in your MATLAB entry-point files. For more information, see [Defining Inputs Programmatically in the MATLAB File](#).

Code Generation from MATLAB

For details about new toolbox functions and System objects supported for code generation, see the [Code Generation from MATLAB Release Notes](#).

New Demo

The following demo has been added:

Demo...	Shows How You Can...
<code>coderdemo_reverb</code>	Generate a MEX function for an algorithm that uses MATLAB classes.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2011b

Version: 2.1

New Features

Support for Deletion of Rows and Columns from Matrices

You can now generate C/C++ code from MATLAB code that deletes rows or columns from matrices. For example, the following code deletes the second column of matrix `X`:

```
X(:,2) = [];
```

For more information, see [Diminishing the Size of a Matrix](#) in the MATLAB documentation.

Code Generation from MATLAB

For details of new toolbox functions and System objects supported for code generation, see [Code Generation from MATLAB Release Notes](#).

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2011a

Version: 2.0

New Features

Compatibility Considerations

New User Interface for Managing Projects

The new MATLAB Coder user interface simplifies the MATLAB to C/C++ code generation process. Using this user interface, you can:

- Specify the MATLAB files from which you want to generate code
- Specify the data types for the inputs to these MATLAB files
- Select an output type:
 - MEX function
 - C/C++ Static Library
 - C/C++ Executable
- Configure build settings to customize your environment for code generation
- Open the code generation report to view build status, generated code, and compile-time information for the variables and expressions in your MATLAB code

To Get Started

You launch a MATLAB Coder project by doing one of the following:

- From the MATLAB main menu, select **File > New > Code Generation Project**
- Enter `coder` at the MATLAB command line

To learn more about working with MATLAB Coder, see [Generating C Code from MATLAB Code Using the MATLAB Coder Project Interface](#).

Migrating from Real-Time Workshop emlc Function

In MATLAB Coder, the `codegen` function replaces `emlc` with the following differences:

New codegen Options

Old emlc Option	New codegen Option
<code>-eg</code>	<code>-args</code>
<code>emlcoder.egc</code>	<code>coder.Constant</code>

Old emlc Option	New codegen Option
emlcoder.egs	<p><code>coder.typeof(a,b,1)</code> specifies a variable-size input with the same class and complexity as <code>a</code> and same size and upper bounds as the size vector <code>b</code>.</p> <p>Creates <code>coder.Type</code> objects for use with the <code>codegen -args</code> option. For more information, see <code>coder.typeof</code>.</p>
-F	<p><code>Nocodegen</code> option available. Instead, use the default <code>fimath</code>. For more information, see the Fixed-Point Toolbox documentation.</p>
-global	<p><code>-globals</code></p> <hr/> <p>Note <code>-global</code> continues to work with <code>codegen</code></p>
-N	<p>This option is no longer supported. Instead, set up <code>numericType</code> in MATLAB.</p>
-s	<p><code>-config</code></p> <p>Use with the new configuration objects, see “New Code Generation Configuration Objects” on page 15-4.</p>
-T rtw:exe	<p><code>-config:exe</code></p> <p>Use this option to generate a C/C++ executable using default build options. Otherwise, use <code>-config</code> with a <code>coder.CodeConfig</code> or <code>coder.EmbeddedCodeConfig</code> configuration object.</p>
-T mex	<p><code>-config:mex</code></p> <p>Use this option to generate a MEX function using default build options. Otherwise, use <code>-config</code> with a <code>coder.MexCodeConfig</code> configuration object.</p>

Old emlc Option	New codegen Option
-T rtw -T rtw:lib	-config:lib Use either of these options to generate a C/C++ library using default build options. Otherwise, use -config with a coder.CodeConfig or coder.EmbeddedCodeConfig configuration object.

New Code Generation Configuration Objects

The codegen function uses new configuration objects that replace the old emlc objects with the following differences:

Old emlc Configuration Object	New codegen Configuration Object
emlcoder.MEXConfig	coder.MexCodeConfig
emlcoder.RTWConfig emlcoder.RTWConfig('grt')	coder.CodeConfig The SupportNonFinite property is now available without an Embedded Coder license. The following property names have changed: <ul style="list-style-type: none"> • RTWCompilerOptimization is now CCompilerOptimization • RTWCustomCompilerOptimization is now CCustomCompilerOptimization • RTWVerbose is now Verbose
emlcoder.RTWConfig('ert')	coder.EmbeddedCodeConfig The following property names have changed: <ul style="list-style-type: none"> • MultiInstanceERTCode is now MultiInstanceCode • RTWCompilerOptimization is now CCompilerOptimization • RTWCustomCompilerOptimization is now CCustomCompilerOptimization • RTWVerbose is now Verbose

Old emlc Configuration Object	New codegen Configuration Object
emlcoder. HardwareImplementation	coder.HardwareImplementation

The codegen Function Has No Default Primary Function Input Type

In previous releases, if you used the `emlc` function to generate code for a MATLAB function with input parameters, and you did not specify the types of these inputs, by default, `emlc` assumed that these inputs were real, scalar, doubles. In R2011a, the `codegen` function does not assume a default type. You must specify at least the class of each primary function input. For more information, see [Specifying Properties of Primary Function Inputs in a Project](#).

Compatibility Considerations

If your existing script calls `emlc` to generate code for a MATLAB function that has inputs and does not specify the input types, and you migrate this script to use `codegen`, you must modify the script to specify inputs.

The codegen Function Processes Compilation Options in a Different Order

In previous releases, the `emlc` function resolved compilation options from left to right so that the right-most option prevailed. In R2011a, the `codegen` function gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the right-most option prevails.

Compatibility Considerations

If your existing script calls `emlc` specifying a configuration object as well as other command-line options, and you migrate this script to use `codegen`, `codegen` might not use the same configuration parameter values as `emlc`.

New coder.Type Classes

MATLAB Coder includes the following new classes to specify input parameter definitions:

- `coder.ArrayType`

- `coder.Constant`
- `coder.EnumType`
- `coder.FiType`
- `coder.PrimitiveType`
- `coder.StructType`
- `coder.Type`

New coder Package Functions

The following new package functions let you work with objects and types for C/C++ code generation:

Function	Purpose
<code>coder.config</code>	Create MATLAB Coder code generation configuration objects
<code>coder.newtype</code>	Create a new <code>coder.Type</code> object
<code>coder.resize</code>	Resize a <code>coder.Type</code> object
<code>coder.typeof</code>	Convert a MATLAB value into its canonical type

Script to Upgrade MATLAB Code to Use MATLAB Coder Syntax

The `coder.upgrade` script helps you upgrade to MATLAB Coder by searching your MATLAB code for old commands and options and replacing them with their new equivalents. For more information, at the MATLAB command prompt, enter `help coder.upgrade`.

Embedded MATLAB Now Called Code Generation from MATLAB

MathWorks is no longer using the term *Embedded MATLAB* to refer to the language subset that supports code generation from MATLAB algorithms. This nomenclature incorrectly implies that the generated code is used in embedded systems only. The new term is code generation from MATLAB. This terminology better reflects the full extent of the capability for translating MATLAB algorithms into readable, efficient, and compact MEX and C/C++ code for deployment to both desktop and embedded systems.

MATLAB Coder Uses `rtwTargetInfo.m` to Register Target Function Libraries

In previous releases, the `emlc` function also recognized the customization file, `sl_customization.m`. In R2011a, the MATLAB Coder software does not recognize this customization file, you must use `rtwTargetInfo.m` to register a Target Function Library (TFL). To register a TFL, you must have Embedded Coder software. For more information, see [Use the `rtwTargetInfo` API to Register a CRL with MATLAB Coder Software](#) in the Embedded Coder documentation.

New Getting Started Tutorial Video

To learn how to generate C code from MATLAB code, see the “Generating C Code from MATLAB Code” video in the MATLAB Coder Getting Started demos.

New Demos

The following demos have been added:

Demo...	Shows How You Can...
Hello World	Generate and run a MEX function from a simple MATLAB program
Working with Persistent Variables	Compute the average for a set of values by using persistent variables
Working with Structure Arrays	Shows how to build a scalar template before growing it into a structure array, a requirement for code generation from MATLAB.
Balls Simulation	Simulates bouncing balls and shows that you should specify only the entry function when you compile the application into a MEX function.
General Relativity with MATLAB Coder	Uses Einstein's theory of general relativity to calculate geodesics in curved space-time.
Averaging Filter	Generate a standalone C library from MATLAB code using <code>codegen</code>
Edge Detection on Images	Generate a standalone C library from MATLAB code that implements a Sobel filter

Demo...	Shows How You Can...
Read Text File	Generate a standalone C library from MATLAB code that uses the <code>coder.ceval</code> , <code>coder.extrinsic</code> and <code>coder.opaque</code> functions.
“Atoms” Simulation	Generate a standalone C library and executable from MATLAB code using a code generation configuration object to enable dynamic memory allocation
Replacing Math Functions and Operators	Use target function libraries (TFLs) to replace operators and functions in the generated code
	Note To run this demo, you need Embedded Coder software.
Kalman Filter	<ul style="list-style-type: none"> • Generate a standalone C library from a MATLAB version of a Kalman filter • Accelerate the Kalman filter algorithm by generating a MEX function

Functionality Being Removed in a Future Version

This function will be removed in a future version of MATLAB Coder software.

Function Name	What Happens When You Use This Function?	Compatibility Considerations
<code>emlc</code>	Still runs in R2011a	None

Function Elements Being Removed in a Future Release

Function or Element Name	What Happens When You Use the Function or Element?	Use This Element Instead
<code>%#eml</code>	Still runs	<code>%#codegen</code>
<code>eml.allowpcode</code>	Still runs	<code>coder.allowpcode</code>
<code>eml.ceval</code>	Still runs	<code>coder.ceval</code>
<code>eml.cstructname</code>	Still runs	<code>coder.cstructname</code>

Function or Element Name	What Happens When You Use the Function or Element?	Use This Element Instead
<code>eml.extrinsic</code>	Still runs	<code>coder.extrinsic</code>
<code>eml.inline</code>	Still runs	<code>coder.inline</code>
<code>eml.nullcopy</code>	Still runs	<code>coder.nullcopy</code>
<code>eml.opaque</code>	Still runs	<code>coder.opaque</code>
<code>eml.ref</code>	Still runs	<code>coder.ref</code>
<code>eml.rref</code>	Still runs	<code>coder.rref</code>
<code>eml.target</code>	Still runs	<code>coder.target</code>
<code>eml.unroll</code>	Still runs	<code>coder.unroll</code>
<code>eml.varsize</code>	Still runs	<code>coder.varsize</code>
<code>eml.wref</code>	Still runs	<code>coder.wref</code>

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.