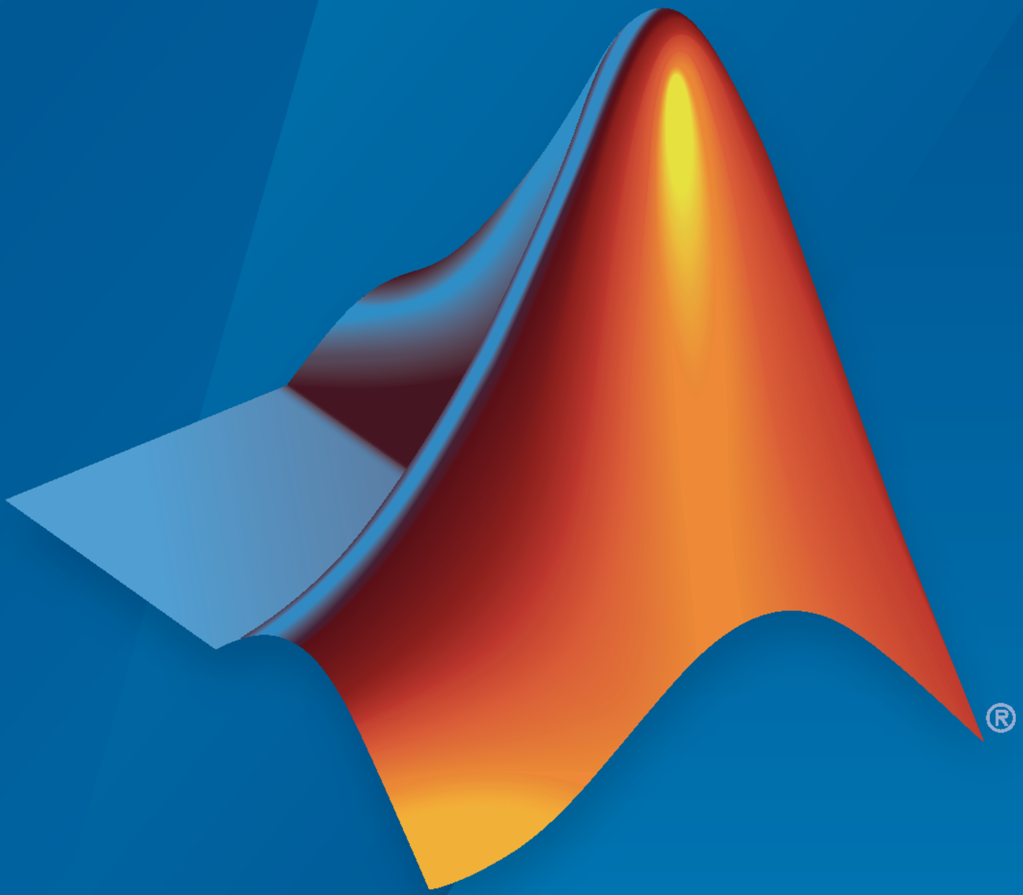
MATLAB® Coder™ Release Notes



MATLAB®

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

MATLAB® Coder™ 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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

Contents

R2023a

R2022a

R2021a

R2014b

R2023a

MATLAB Programming for Code Generation

Name-Value Argument Validation: Generate code for arguments blocks in MATLAB functions

In R2023a, you can generate code for `arguments` blocks that validate name-value arguments in your MATLAB function. You declare name-value arguments in an `arguments` block using dot notation to define the fields of a structure. See “Validate Name-Value Arguments”.

In this example code snippet, the structure named `NameValueArgs` defines two name-value arguments, `Name1` and `Name2`. You can use any valid MATLAB identifier for the structure name and the field names.

```
function result = myFunction(NameValueArgs)
    arguments
        NameValueArgs.Name1
        NameValueArgs.Name2
    end
    ...
end
```

Code generation supports most features of `arguments` blocks for name-value arguments, including size and class validation, validation functions, and default values. Code generation also supports the `namedargs2cell` function.

Code generation does not support these features of name-value argument validation:

- Name-value input arguments at entry-point functions
- Name-value arguments from class properties using the `structName.?ClassName` syntax

See “Generate Code for arguments Block That Validates Input and Output Arguments”.

Output Argument Validation: Generate code for arguments(Output) blocks in MATLAB functions

In R2023a, you can generate code for `arguments` blocks that perform output argument validation in your MATLAB function. Output argument validation declares specific restrictions on function output arguments. Using argument validation, you can constrain the class, size, and other aspects of function output values without writing code in the body of the function to perform these tests.

Code generation supports most features of `arguments` blocks for output variables, including size and class validation, and validation functions. For repeating output arguments, code generation does not support size validation, class validation, and validation functions.

See “Generate Code for arguments Block That Validates Input and Output Arguments”.

Input Argument Validation: Use any name for repeating input arguments

In R2023a, you can use any valid MATLAB identifier for the name of a repeating input argument inside an `arguments` block. Code generation only supports a single repeating argument for a function.

In previous releases, code generation supported only `varargin` as a repeating input argument.

Generate code for growing arrays with (end + 1) indexing

In R2023a, you can generate code for MATLAB code that uses the `(end + 1)` indexing syntax to grow the size of arrays. For example, you can generate code for this code snippet:

```
...
a = [1 2 3 4 5 6];
a(end + 1) = 7;

b = [1 2];
for i = 3:10
    b(end + 1) = i;
end
...
```

To use this functionality, make sure that the code generation configuration property `EnableVariableSizing` or the corresponding setting **Enable variable-sizing** in the MATLAB Coder app is enabled. See “Generate Code for Growing Arrays and Cell Arrays with end + 1 Indexing”.

Generate code for uint32 enumerations

In R2023a, you can generate code for a MATLAB enumeration that derives from the base type `uint32`. For members of `uint32` enumerations, code generation supports values that are less than or equal to `intmax("int32")`. See “Code Generation for Enumerations”.

coder.read and coder.write: Read data from .coderdata file into your deployed application

In R2023a, you can use the `coder.read` function to read data from `.coderdata` files. In contrast with MAT-files that can be read only inside the MATLAB environment, you can read `.coderdata` files on any deployment platform that supports a file system. In addition, the `.coderdata` format supports most primitive and aggregate MATLAB data types, including arrays, structures, and cell arrays. So, the C/C++ code generated for `coder.read` can be used to read complex aggregate data from `.coderdata` files into your deployed application.

To export MATLAB data to `.coderdata` files, use the `coder.write` function. This function is not supported for code generation.

The code generated for `coder.read` has two distinct advantages over the code generated for the `coder.load` function:

- You can update the data stored in `.coderdata` files without having to regenerate code, as long as the type and size of the new data matches those of the old data.
- The data is not hard-coded in the generated code, thereby improving the readability of the generated code.

This is an example workflow that uses the `coder.read` and `coder.write` functions:

1. Use the `coder.write` function at the MATLAB command line to store the data in `.coderdata` files. For example, create a file named `myfile.coderdata` by using these commands:

```
c = rand(100);
coder.write('myfile.coderdata',c);
```

2 In your MATLAB entry-point function (for which you intend to generate code), use the `coder.read` function to read data from the `.coderdata` files. For example:

```
function y = my_entry_point(x) %#codegen
dataOut = coder.read('myfile.coderdata');
y = x + mean(dataOut,"all");
end
```

3 Generate MEX or standalone C/C++ code for the entry-point function by using the `codegen` command or the MATLAB Coder app. For example, generate a MEX function `my_entry_point_mex` and then call the generated MEX by running these commands:

```
codegen my_entry_point -args {0}
my_entry_point_mex(1)

Code generation successful.

ans =

    1.4996
```

4 You can now update the data stored in `myfile.coderdata` to a different `100`-by-`100` array of double type. If you then call `my_entry_point_mex` that you already generated, the MEX now reads and uses the new data.

```
d = rand(100) - 1;
coder.write('myfile.coderdata',d);
my_entry_point_mex(1)

Wrote file 'myfile.coderdata'. You can read this file with 'coder.read'.

ans =

    0.4963
```

For more information and examples, see `coder.read`, `coder.write`, and “Data Read and Write Considerations”.

Dynamic memory allocation for fixed-size arrays

Starting in R2023a, you can dynamically allocate memory to heap for fixed-size arrays. By default, dynamic memory allocation for fixed-size arrays is disabled.

To enable dynamic memory allocation for fixed-size arrays:

- In a configuration object for code generation, set the `EnableDynamicMemoryAllocation` and `DynamicMemoryAllocationForFixedSizeArrays` parameters to `true`.
- In the MATLAB Coder app, in the **Memory** settings, select **Enable dynamic memory allocation** and **Enable dynamic memory allocation for fixed-sized arrays**.

For more information, see “Control Dynamic Memory Allocation for Fixed-Size Arrays”.

Compatibility Considerations

The `DynamicMemoryAllocation` configuration option will be removed in a future release. To dynamically allocate memory for variable-sized arrays, use the `EnableDynamicMemoryAllocation` option. To set the threshold, use the `DynamicMemoryAllocationThreshold` option.

Supported Functions

Code generation for more MATLAB functions

- `cd`
- `filesep`
- `griddedInterpolant`
- `inedges`
- `isfile`
- `isfolder`
- `ismac`
- `ispc`
- `isuniform`
- `isunix`
- `mape`
- `mustBeFile`
- `mustBeFolder`
- `mustBeInRange`
- `namedargs2cell`
- `NET.isNETSupported`
- `num2cell`
- `outedges`
- `pathsep`
- `polyshape`
- `predecessors`
- `pwd`
- `rmse`
- `strcat`
- `successors`
- `warning`

Code generation for more toolbox functions

In R2023a, you can generate code for many additional toolbox functions and objects. For a list of all functions and objects that are supported for code generation, see:

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

These are links to the release notes of some toolboxes that added code generation support in R2023a:

Computer Vision Toolbox

See “Generate C and C++ Code Using MATLAB Coder: Support for functions” (Computer Vision Toolbox).

Image Processing Toolbox

See “C Code Generation: Generate code from additional functions using MATLAB Coder” (Image Processing Toolbox).

Signal Processing Toolbox

See “C/C++ Code Generation Support: Code generation for digital filter design, multirate signal processing, and waveform generation” (Signal Processing Toolbox).

Statistics and Machine Learning Toolbox

See “Generate C/C++ code for prediction using Gaussian kernel classification and regression models (requires MATLAB Coder)” (Statistics and Machine Learning Toolbox).

Wavelet Toolbox

See “C/C++ Code Generation: Automatically generate code for wavelet functions” (Wavelet Toolbox).

Generated Code Improvements

Generate C++11 code that passes variables by reference

In R2023a, when you generate C++11 code for your MATLAB code, the code generator passes pointer arguments as references whenever it can establish that those arguments are not null. This behavior applies to functions that are not entry-point functions. Passing function arguments by reference improves MISRA™ compliance and makes the generated code more idiomatic.

Improved quality of code generated for logical indexing operations

In R2023a, the code generated for logical indexing operations in MATLAB is more readable and closer to hand-written C/C++ code for such operations. In some situations, the generated code can also have better performance compared to previous releases.

MATLAB Code	R2023a Generated Code	R2022b Generated Code
`% entry-point function`	`void foo(double A_data[], const int A_size[2])`	`void foo(double A_data[], const int A_s`
`function A = foo(A)`	`{`	`{`
`A(A < 0) = A(A < 0) * -1;`	`  int end;`	`  double b_A_data[25];`
`end`	`  int i;`	`  int end_tmp;`
	`  end = A_size[0] * A_size[1] - 1;`	`  int i;`
`% code generation`	`  for (i = 0; i <= end; i++) {`	`  int partialTrueCount;`
`codegen -config:lib foo -args {coder.typeof(1,[5 5],[1 1])}`	`    double d;`	`  int trueCount;`
	`    d = A_data[i];`	`  signed char b_tmp_data[25];`
	`    if (d < 0.0) {`	`  signed char tmp_data[25];`
	`      d = -d;`	`  end_tmp = A_size[0] * A_size[1] - 1;`
	`      A_data[i] = d;`	`  trueCount = 0;`
	`    }`	`  partialTrueCount = 0;`
	`  }`	`  for (i = 0; i <= end_tmp; i++) {`
	`}`	`    if (A_data[i] < 0.0) {`
		`      trueCount++;`
		`      tmp_data[partialTrueCount] = (sig`
		`      partialTrueCount++;`
		`    }`
		`  }`
		`  partialTrueCount = 0;`
		`  for (i = 0; i <= end_tmp; i++) {`
		`    if (A_data[i] < 0.0) {`
		`      b_tmp_data[partialTrueCount] = (s`
		`      partialTrueCount++;`
		`    }`
		`  }`
		`  for (end_tmp = 0; end_tmp < trueCount`
		`    b_A_data[end_tmp] = -A_data[tmp_dat`
		`  }`
		`  for (end_tmp = 0; end_tmp < trueCount`
		`    A_data[b_tmp_data[end_tmp] - 1] = b`
		`  }`
		`}`

Improved loop fusion for vectorized operations that use variable-size arrays

Vectorized operations involving arrays in your MATLAB code get converted to loops in the generated code. If the arrays are variable-size, the corresponding loop upper bounds are also variable-size. In R2023a, if the code generator produces multiple loops that have identical variable-size upper bounds, it often fuses them into a single loop. This optimization is likely to improve both the performance and the readability of the generated code.

MATLAB Code	R2023a Generated Code	R2022b Generated Code

`% entry-point function`
`function A = bar(B)`
`A = [B;B];`
`end`
 	
`% code generation`
`codegen -config:lib bar -args {coder.typeof(1,[10 10],[1 1])}`
	`...`
	`  for (i = 0; i < result; i++) {`
	`    for (i1 = 0; i1 < input_sizes_idx_0; i1++) {`
	`      A_data[i1 + A_size[0] * i] = B_data[i1 + input_sizes_idx_0 * i];`
	`    }`
	`    for (i1 = 0; i1 < sizes_idx_0_tmp; i1++) {`
	`      A_data[(i1 + input_sizes_idx_0) + A_size[0] * i] =`
	`          B_data[i1 + sizes_idx_0 * i];`
	`    }`
	`  }`
	`...`
		`...`
		`  for (i = 0; i < result; i++) {`
		`    for (i1 = 0; i1 < input_sizes_idx_0`
		`      A_data[i1 + A_size[0] * i] = B_da`
		`    }`
		`  }`
		`  for (i = 0; i < result; i++) {`
		`    for (i1 = 0; i1 < sizes_idx_0_tmp;`
		`      A_data[(i1 + input_sizes_idx_0) +`
		`          B_data[i1 + sizes_idx_0 * i];`
		`    }`
		`  }`
		`...`

Removed redundant operations that use identical values

In R2023a, the generated code no longer contains some redundant operations that access values that are identical at run time. By eliminating these redundant operations, the generated code shows improved execution speed.

For example, consider this sigmoid function, which uses a variable-size input.

```
function sigm = sigmoidFcn(x)
    sigm = 1 ./ (1 + exp(-x));
end
```

In R2022b, the generated code for `sigmoidFcn` contained this code, which accesses the data in the `sigm_data` array in three separate `for` loops.

```
    int32_T k;
    int32_T loop_ub;
    sigm_size[0] = x_size[0];
    sigm_size[1] = x_size[1];
    loop_ub = x_size[0] * x_size[1];
    for (k = 0; k < loop_ub; k++) {
        sigm_data[k] = -x_data[k];
    }

    loop_ub = x_size[0] * x_size[1];
    for (k = 0; k < loop_ub; k++) {
        sigm_data[k] = exp(sigm_data[k]);
    }

    for (k = 0; k < loop_ub; k++) {
        sigm_data[k] = 1.0 / (sigm_data[k] + 1.0);
    }
```

In R2023a, the generated code combines the operations and accesses the data in the `sigm_data` array in only one `for` loop.

```
    int32_T k;
    int32_T loop_ub_tmp;
    sigm_size[0] = x_size[0];
    sigm_size[1] = x_size[1];
    loop_ub_tmp = x_size[0] * x_size[1];
    for (k = 0; k < loop_ub_tmp; k++) {
        sigm_data[k] = 1.0 / (exp(-x_data[k]) + 1.0);
    }
```

The generated code produces the same output as the code from R2022b, but the R2023a code accesses the array data fewer times and shows improved execution speed.

Code Generation Workflow

Generate generic CMakeLists.txt file when you generate source code only

In R2023a, when you generate only the C/C++ source code for your MATLAB code, you can instruct the code generator to also produce a `CMakeLists.txt` file that does not depend on specific build tools. Do one of the following:

- In a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object, set the `Toolchain` property to `"CMake"`.
- In the MATLAB Coder app, in the **Generate Code** step, on the **More Settings > Hardware** tab, set **Toolchain** to `CMake`.

See “Configure CMake Build Process”.

Improved Error Recovery: Code generation produces fewer unhelpful cascading errors

In previous releases, a single unsupported construct in your MATLAB code often produced multiple cascading errors during code generation. In such situations, only the first error was helpful because fixing it also fixed the subsequent cascading errors. In R2023a, for most situations, code generation only produces the initial error and suppresses the subsequent unhelpful cascading errors.

Functionality being removed or changed

Specifying Multiple Files or Paths for a Configuration Property by Using Character Vector to be Removed

Warns

In a future release, the capability to specify multiple file names or paths for the `CustomInclude`, `CustomLibrary`, and `CustomSource` code configuration properties by using character vectors or string scalars that have delimiters will be removed. Use string arrays or cell arrays of character vectors instead. For example, to include multiple folder names, set the `CustomInclude` property by using either of these syntaxes:

- Use string array: `cfg.CustomInclude = ["C:\Project","C:\Custom Files"]`
- Use cell array of character vectors: `cfg.CustomInclude = {'C:\Project','C:\Custom Files'}`

Performance

Generate standalone code that uses built-in FFTW library

In R2023a, the required FFTW library is shipped with MATLAB and the code generation process is simpler compared to previous releases. To generate code that produces calls to this built-in FFTW library for fast Fourier transform (FFT) functions in your MATLAB code, do one of the following:

- In a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object, set the property `UseBuiltinFFTWLibrary` to `true`.
- In the MATLAB Coder app, on the **Custom Code** tab, select the **Use built-in FFTW library** option.

Prior to R2023a, to generate code that uses the FFTW library, you had to install the FFTW library, write a custom callback class to specify the FFTW library installation using `coder.fftw.StandaloneFFTW3Interface`, and then set the configuration parameter **Custom FFT library callback** (`CustomFFTCallback`) to the name of the callback class. This functionality continues to exist and is particularly useful if you want to either customize the FFTW options or use your own compiled version of the FFTW library for deployment on an embedded device. See “Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”.

Loop Optimization: Use coder.loop.Control objects to improve for loop performance in generated code

In R2023a, you can instruct the code generator to transform specific `for` loops in your MATLAB code in a variety of ways (such as parallelize or vectorize) during code generation. These transforms can often improve the run-time performance of large loops. To specify these transforms, use the following directives in your MATLAB functions for which you intend to generate code:

- `coder.loop.interchange`: Interchange nested loops to improve cache performance when accessing array elements.
- `coder.loop.parallelize`: Parallelize loop execution to improve speed by utilizing available threads.
- `coder.loop.reverse`: Reverse the execution order of loop iterations. In some situations, the reversed loop execution can be faster.
- `coder.loop.tile`: Tile loop nests to reduce memory access latency.
- `coder.loop.unrollAndJam`: Unroll and jam loops to improve cache locality.
- `coder.loop.vectorize`: Generate code that uses SIMD instructions to apply an operation simultaneously to multiple instances.

In your MATLAB code, call the directive immediately before the loop you intend to transform. You can combine multiple transforms into a single call by using the dot builder syntax shown in this code snippet. Use the loop index variable name to specify the loop to which you intend to apply a certain transform.

```
coder.loop.parallelize('loopId').interchange('loopId','loopId2');
for loopId = 1:100
    for loopId2 = 1:100
...
end
```

In some situations, you might want to combine multiple transforms in more complex ways than is possible by using this simple dot builder syntax. For example, you might want to apply one of the transforms only if a certain compile-time condition is satisfied. In such situations, use the `coder.loop.Control` objects and the associated object functions. For example:

```
...
loopControl = coder.loop.Control;
loopControl = loopControl.parallelize('loopId');

% You can apply multiple transforms to the same object
if inputVal > threshold
    loopControl = loopControl.interchange('loopId','loopId2');
end

% Call the apply method to inform the code
% generator to affect the loops in the generated code
loopControl.apply;

for loopId = 1:100
    for loopId2 = 1:100
...
end
```

See “Optimize Loops in Generated Code”.

Generate SIMD instructions in MEX code

In R2023a, you can improve the performance of generated MEX code on Intel® and AMD® platforms by including vectorized SIMD instructions in the generated code. Set the new configuration parameter **Hardware SIMD acceleration** or the command-line property `SIMDAcceleration` to one of these values:

- **Portable** (default) — Use the SSE2 instruction set
- **Full** — Use the AVX2 instruction set
- **None** — Do not use instruction sets

MATLAB Coder generates portable MEX code using the SSE2 instruction set by default. For more information, see “Generate SIMD Code for MATLAB Functions”.

Functionality being removed or changed

Code generator no longer produces calls to FFTW cleanup functions

Behavior change

In previous releases, when you used the `coder.fftw.StandaloneFFTW3Interface` callback class to generate standalone code that calls FFTW library functions, the generated terminate function included calls to one or more of the following memory cleanup functions.

- `fftw_cleanup`
- `fftwf_cleanup`
- `fftw_cleanup_threads`
- `fftwf_cleanup_threads`

These functions are called to clean up memory that the FFTW library functions use while the calling process is still running. By contrast, when the calling process terminates, FFTW library functions automatically free the memory and these cleanup functions don't need to be called.

In R2023a, these functions calls are no longer included in the generated terminate function. To clean up memory that the FFTW library functions use while the calling process is still running, you must incorporate the generated code into your own project and manually call the appropriate cleanup function after executing the generated code. Follow these rules to decide which cleanup function to use:

- If you use single-precision floating point numbers in calls to FFT functions, use the `fftwf` prefixed cleanup functions.
- If you use double-precision floating point numbers in calls to FFT functions, use the `fftw` prefixed cleanup functions.
- If your implementation of the `coder.fftw.StandaloneFFTW3Interface.getNumThreads` method returns a value that is greater than `1`, use the cleanup functions that have the `threads` suffix.

See “Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”.

Deep Learning with MATLAB Coder

Generate code for variable-size dlarray data type

In R2023a, you can generate code for MATLAB code that uses variable-size `dlarray` objects.

For example, define this MATLAB design file:

```
function out = fooAdd(in1,in2) %#codegen
dlIn1_1 = dlarray(in1);
dlIn1_2 = dlarray(in2);
out = dlIn1_1 + dlIn1_2;
end
```

Specify the two inputs `in1` and `in2` to be unbounded two-dimensional arrays of `single` type. Create the appropriate code configuration object `cfg` to generate generic C MEX code for `fooAdd`. Generate MEX code and run the generated MEX.

```
t_in1 = coder.typeof(single(1),[inf inf],[1 1]);
t_in2 = coder.typeof(single(1),[inf inf],[1 1]);

codegen fooAdd -args {t_in1,t_in2} -report

out = fooAdd_mex(single(eye(4,4)),single(ones(4,1)));
```

When generating code for variable-size `dlarray` objects, adhere to these restrictions:

- The `U` dimension of a `dlarray` object must be of fixed size.
- If the `dlarray` data format `fmt` contains only one character, the corresponding data array `X` can have only one variable-size dimension. All other dimensions of `X` must be singleton.
- For operations between a `dlarray` object and a numeric array that might implicitly expand either operands, do not combine a fixed size `U` dimension of the `dlarray` object with a variable-size dimension of the numeric array.
- For unary operations such as `max`, `min`, and `mean` on a variable-size `dlarray` object, specify the intended working dimension explicitly as a constant value. See “Automatic dimension restriction”.

See “dlarray Limitations for Code Generation”.

Generate code for dlnetwork objects that accept variable sequence length inputs

In R2023a, you can generate code for `dlnetwork` objects that accept `dlarray` inputs with a variable-size time (T) dimension. You use such `dlarray` objects to represent time series data of variable sequence length.

For more information on how to create variable-size `dlarray` objects for code generation, see “Generate code for variable-size dlarray data type” on page 1-15.

Generate code for channel-wise convolution layer

In R2023a, you can generate C or C++ code that does not depend on third-party libraries for channel-wise convolution (also known as depth-wise convolution) layer with `groupedConvolution2dLayer`.

Generate code for Pooling layers with mean padding

In R2023a, you can generate generic C/C++ code in MATLAB for the following layer using `'mean'` for `PaddingValue` property:

- `averagePooling2dLayer`

Generate code that takes advantage of learnables compression in bfloat16 format

In 2023a, you can perform learnables compression and generate C/C++ code for these layers in Brain Floating Point format, `bfloat16`:

- `fullyConnectedLayer`
- `gruLayer`
- `lstmLayer`
- `bilstmLayer`
- `lstmProjectedLayer`

`bfloat16` format keeps the same number range as 32-bit IEEE 754 single-precision floating-point format. Compressing learnables from single-precision to `bfloat16` reduces memory usage of deep learning networks with accuracy variance. This enables deployment of larger networks to devices with tight memory budget. For more information on `bfloat16` format, see “Generate bfloat16 Code for Deep Learning Networks”.

To enable learnables compression, set the `LearnablesCompression` property of the deep learning configuration object `coder.DeepLearningConfig` to `bfloat16`.

```
dlcfg = coder.DeepLearningConfig(TargetLibrary = 'none');
dlcfg.LearnablesCompression = 'bfloat16';
```

Alternatively, you can set the new **Learnables Compression** property in the **Deep Learning** setting tab of the MATLAB Coder App or the Configuration Parameters dialog box.

Deep learning configuration object name change

`coder.DeepLearningConfigBase` configuration object is now called `coder.DeepLearningCodeConfig`. The behavior remains the same.

Quantized TensorFlow Lite Models: Configure predict function to accept and return fp32 values

Quantized deep learning models use reduced-precision numbers, usually 8-bit integers (`int8` or `uint8`) instead of 32-bit floating point numbers (`fp32`), to represent the model's parameters. For inference computation with quantized TFLite models, the `predict` function accepts and returns 8-bit integer values by default. In R2023a, when performing inference computation with quantized TFLite models, you can configure the `predict` function to accept and return `fp32` values and perform the appropriate conversion at the function interface. To do this, use the `predict` function with the additional name-value arguments `QuantizeInputs` and `DequantizeOutputs`.

Use newer version of TensorFlow Lite library in simulation and code generation

In R2023a, you can perform inference with models created using TFLite version 2.8.0 in both simulation and code generation. TFLite models are forward and backward compatible. So, if your model was created using a different version of the library but contains layers that are available in version 2.8.0, you can still simulate, generate code, and deploy your model. For more information, see “Prerequisites for Deep Learning with TensorFlow Lite Models” (Deep Learning Toolbox).

Improved performance of generated generic C/C++ code

In R2023a, generated generic C/C++ code that does not depend on third-party libraries has improved performance for networks containing the following layers:

- `convolution2dLayer`
- `maxPooling2dLayer`
- `globalMaxPooling2dLayer`
- `averagePooling2dLayer`
- `globalAveragePooling2dLayer`
- `reluLayer`
- `leakyReluLayer`
- `clippedReluLayer`
- `additionLayer`
- `multiplicationLayer`
- `gruLayer`
- `lstmLayer`
- `bilstmLayer`
- `lstmProjectedLayer`
- Layer that implements ONNX identity operator

  `nnet.onnx.layer.GlobalAveragePooling2dLayer` (Deep Learning Toolbox)
- Layer that implements KERAS identity operator

  `nnet.keras.layer.GlobalAveragePooling2dLayer` (Deep Learning Toolbox)

In addition, you are likely to have further performance improvement using the SIMD intrinsics. For more information, see “Generate SIMD Code for MATLAB Functions”.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2022b

MATLAB Programming for Code Generation

Input Argument Validation: Generate code for arguments blocks in MATLAB functions

In R2022b, you can generate code for `arguments` blocks that perform input argument validation in your MATLAB function. Input argument validation declares specific restrictions on function input arguments. Using argument validation, you can constrain the class, size, and other aspects of function input values without writing code in the body of the function to perform these tests.

Code generation supports most features of `arguments` blocks, including size and class validation, validation functions, and default values.

Code generation supports only `varargin` as a repeating argument. For `varargin`, size validation, class validation, and validation functions are not supported for code generation.

Code generation does not support these features of `arguments` blocks:

- Repeating arguments other than `varargin`
- Name-value arguments
- Output argument validation

See Generate Code for `arguments` Block That Validates Input Arguments.

More MATLAB functions declared as auto-extrinsic

In R2022b, code generation automatically treats several additional MATLAB functions as extrinsic. You do not need to explicitly specify that these functions are extrinsic by using the `coder.extrinsic` construct.

These functions include:

- `contour`
- `histogram`
- `imshow`
- `legend`
- `patch`
- `scatter`
- `stairs`
- `trimesh`
- `voronoi`

For more information, see `coder.extrinsic` and Use MATLAB Engine to Execute a Function Call in Generated Code.

Supported Functions

Code generation for more MATLAB functions

- `allfinite`
- `anymissing`
- `anynan`
- `groupfilter`
- `grouptransform`
- `griddedInterpolant`
- `integral2`
- `integral3`
- `makima`
- `nearest`
- `ode15s`
- `pagetranspose`
- `pagectranspose`
- `shortestpath`

Code generation for more toolbox functions

In R2022b, you can generate code for many additional toolbox functions and objects. For a list of all functions and objects that are supported for code generation, see:

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

Code Generation Workflow

Improved representation for string type objects

Starting in R2022b, the representation of the string data type as coder type objects that you create by using `coder.typeof()` and `coder.newtype()` is easier to use, more succinct, and excludes internal state values. Pass a `coder.StringType` object as an input to the `codegen -args` option to specify inputs to the generated code as a string.

The following code snippet shows the representation of a string input type that is created by using the `coder.typeof()` function.

```
t = coder.typeof("Hello")
```

```
t =

coder.StringType
   1×1 string
       StringLength: 5
       VariableStringLength: false
```

To change the string length of the string type object, set the `StringLength` property to the required value. To make the string length variable size, set `VariableStringLength` to `true`. Setting `StringLength` to `Inf` automatically sets `VariableStringLength` to `true`. Consider the following example:

```
t = coder.typeof("world");

% To specify t string length as upperbound at 10
t.StringLength = 10;
t.VariableStringLength = true;

% To specify t string length as variable-size without upper bound
t.StringLength = Inf;
```

Compatibility Considerations

In prior releases, to change the string length of the type object, the `Value` property must be set accordingly. Similarly, to make the string length variable size, the `Value` property must be made variable-size. The legacy interface is accessible in this release. Consider the following example:

```
t = coder.typeof("hello");

% To specify t string length as upperbound character vector
t.Properties.Value = coder.typeof('a',[1 10],[0 1]);

% To specify t string length as variable-size without upper bound
t.Properties.Value = coder.typeof('a',[1 Inf]);
```

Specify code generation target language by using coder.target

Starting in R2022b, code generator allows you to specialize the MATLAB code for specific target language.

You can use `coder.target` in the MATLAB code for which you are generating code as the following.

```
coder.target('CUDA');
```

Supported target languages for code generation are:

- C
- C++
- CUDA
- OpenCL
- SystemC
- SystemVerilog
- Verilog
- VHDL

Build generated code with CMake

To build code generated from MATLAB code, R2022b provides CMake toolchain definitions for:

- Microsoft® Visual C++® and MinGW® on Windows®, GCC on Linux®, and Xcode on Mac computers, using Ninja and makefile generators.
- Microsoft Visual Studio® and Xcode project builds.

CMake and the associated `CMakeLists.txt` file are widely used for building C++ code and can be directly leveraged by command-line tools and IDEs like Microsoft Visual Studio, Microsoft Visual Studio Code, Xcode, and CLion.

If a supported toolchain is installed on your development computer, you can specify the corresponding CMake toolchain definition during code generation. When you run `codegen` at the command line or click the **Generate Code** button in the MATLAB Coder app, CMake:

1. Uses configuration (`CMakeLists.txt`) files to generate standard build files.
2. Runs the compiler and other build tools to create executable code.

For more information, see Configure CMake Build Process and https://www.mathworks.com/support/requirements/supported-compilers.html.

Creation of custom CMake toolchain definitions

Using the `target` package, create custom CMake toolchain definitions for building code generated from MATLAB code. You can:

- Specify CMake parameters, for example, `Generator` and `Toolchain file`.
- Associate the toolchain with operating systems of your development computers.
- Associate the toolchain with your target hardware.
- Add the toolchain definition to an internal database, which enables you to use the toolchain in subsequent MATLAB sessions.

For more information, see Create Custom CMake Toolchain Definition and https://
www.mathworks.com/support/requirements/supported-compilers.html.

Performance

Improved cache efficiency of generated code containing loop distribution, interchange, and reversal

In R2022b, the code generator can optimize the generated code by applying loop interchange and distribution. These loop transformations increase the number of cache hits and improve the code execution time. The optimizations apply to code generation targets for which the cache information is available to the code generator. To increase the availability of cache information to the code generation target, specify the target hardware information by using the `coder.HardwareImplementation` object `ProdHWDeviceType`.

This code performs operations on the elements of the two input matrices of dimension [180x80] by using for loops.

```
function out = MatLabFun(A, B)

sizeRow=90;
sizeCol=80;

for i = 2 : sizeRow
    for j = 2 : sizeCol
        B(i*2,j) =  B((i*2)-1,j)+i*j;
        for k = 2 : sizeCol
            A(i*2,k) =  A(i-1,k)+i+j;
        end
    end
end

out = [A;B];
end
```

In R2022a, the generated code contains one loop nest that evaluates the loop with iteration variable `B_tmp` at the innermost position.

```
void MatLabFun(double A[14400], double B[14400], double out[28800])
{
  int A_tmp;
  int B_tmp;
  int B_tmp_tmp;
  int i;
  int j;
  for (i = 0; i < 89; i++) {
    B_tmp_tmp = (i + 2) << 1;
    for (j = 0; j < 79; j++) {
      B_tmp = B_tmp_tmp + 180 * (j + 1);
      B[B_tmp - 1] = B[B_tmp - 2] + (double)((i + 2) * (j + 2));
      for (B_tmp = 0; B_tmp < 79; B_tmp++) {
        A_tmp = 180 * (B_tmp + 1);
        A[(B_tmp_tmp + A_tmp) - 1] =
            (A[i + A_tmp] + ((double)i + 2.0)) + ((double)j + 2.0);
      }
    }
  }
  for (B_tmp = 0; B_tmp < 80; B_tmp++) {
    for (A_tmp = 0; A_tmp < 180; A_tmp++) {
      i = A_tmp + 180 * B_tmp;
      B_tmp_tmp = A_tmp + 360 * B_tmp;
      out[B_tmp_tmp] = A[i];
      out[B_tmp_tmp + 180] = B[i];
    }
  }
}
```

In R2022b, the loop in the generated code is distributed to two loop nests. The loop nests are interchanged to evaluate the loop with iteration variable `j` at the innermost position.

```
void MatLabFun(double A[14400], double B[14400], double out[28800])
{
  int A_tmp;
  int B_tmp;
  int i;
  int j;
  for (j = 0; j < 79; j++) {
    for (i = 0; i < 89; i++) {
      B_tmp = ((i + 2) << 1) + 180 * (j + 1);
      B[B_tmp - 1] = B[B_tmp - 2] + (double)((i + 2) * (j + 2));
    }
  }
  for (B_tmp = 0; B_tmp < 79; B_tmp++) {
    A_tmp = 180 * (B_tmp + 1);
    for (i = 0; i < 89; i++) {
      for (j = 0; j < 79; j++) {
        A[(((i + 2) << 1) + A_tmp) - 1] =
            (A[i + A_tmp] + ((double)i + 2.0)) + ((double)j + 2.0);
      }
    }
  }
  for (B_tmp = 0; B_tmp < 80; B_tmp++) {
    for (A_tmp = 0; A_tmp < 180; A_tmp++) {
      j = A_tmp + 180 * B_tmp;
      i = A_tmp + 360 * B_tmp;
      out[i] = A[j];
      out[i + 180] = B[j];
    }
  }
}
```

This interchange improves the locality of reference for the loop nest and improves cache performance.

Improved performance of generated MEX files

In R2022b, performance improvements to generated MEX files include:

- Optimization of the run-time library that is used by the generated MEX files.
- Reduction of the initialization overhead of the generated MEX file.

SIMD code for bitwise and shift operations

In R2022b, you can generate SIMD code for bitwise operations and shift operations. When you select an instruction set by using the **Leverage target hardware instruction set extensions** parameter, the generated code includes the associated instructions for these bitwise operations and shift operations:

- `bitand`
- `bitor`
- `bitxor`
- `bitshift`

For more information, see Generate SIMD Code for MATLAB Functions.

Deep Learning with MATLAB Coder

Deep Learning: Analyze and find issues in the network for code generation

You can analyze code generation compatibility of deep learning networks by using the `analyzeNetworkForCodegen` function. Use the network code generation analyzer to validate a `SeriesNetwork`, `DAGNetwork`, and `dlnetwork` for non-library and library targets and detect problems before code generation. Supported library targets include `MKL-DNN`, `ARM Compute`, and `CMSIS-NN`. Problems that `analyzeNetworkForCodegen` detects include unsupported layers for code generation, network issues, built-in layer specific issues, and issues with custom layers.

The `analyzeNetworkForCodegen` function requires the MATLAB Coder Interface for Deep Learning and GPU Coder™ Interface for Deep Learning support packages. To download and install support package, use the Add-On Explorer. You can also download the support packages from MathWorks GPU Coder Team and MathWorks MATLAB Coder Team. For more information, see Analyze Network for Code Generation.

TensorFlow Lite: Generate C++ code for pretrained models and deploy on Windows platforms

Use the `loadTFLiteModel` (Deep Learning Toolbox) function to load a pretrained TensorFlow™ Lite model into a `TFLiteModel` (Deep Learning Toolbox) object. Use this object with the `predict` (Deep Learning Toolbox) function in your MATLAB code to perform inference in MATLAB execution, code generation, or inside MATLAB Function blocks in Simulink® models.

To use this functionality, you must install the Deep Learning Toolbox™ Interface for TensorFlow Lite. For more information, see Prerequisites for Deep Learning with TensorFlow Lite Models (Deep Learning Toolbox). For examples, see:

- Deploy Super Resolution Application That Uses TensorFlow Lite (TFLite) Model on Host and Raspberry Pi (Deep Learning Toolbox)
- Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi (Deep Learning Toolbox)

Generate code for dlnetwork objects that do not have input layers

In R2022b, you can generate code for `dlnetwork` (Deep Learning Toolbox) objects that do not contain input layers. This enables you to generate code for `dlnetwork` objects that do not represent entire models but are used as intermediate building blocks that you connect together to create complex networks. The generated code can take advantage of either the Intel MKL-DNN or the ARM® Compute library. You can also generate generic C/C++ code that does not depend on any third-party libraries.

Deep Learning Arrays: Generate code for more functions that use dlarray

In R2022b, you can generate code for additional MATLAB functions that use `dlarray` (Deep Learning Toolbox) inputs. You can now generate code for these functions:

- Size Manipulation functions — Use `repelem` to repeat copies of array elements.
- Size Manipulation functions — Use `repmat` to repeat copies of array.
- Error function — Use `erf` to compute the error function for each element of input.

To learn more about generating code from MATLAB functions when using `dlarray` (Deep Learning Toolbox), see Code Generation for dlarray.

Deep Learning Networks: Generate code for additional networks

In R2022b, you can generate generic C/C++ code for these additional networks:

- `yolov3ObjectDetector` (Computer Vision Toolbox)– YOLO v3 object detector. This feature requires the functions in the Computer Vision Toolbox™ Model for YOLO v3 Object Detection support package.
- `yolov4ObjectDetector` (Computer Vision Toolbox) – YOLO v4 object detector. This feature requires the functions in the Computer Vision Toolbox Model for YOLO v4 Object Detection support package.
- `ssdObjectDetector` (Computer Vision Toolbox) – SSD-based object detector.

Deep Learning: Generate code for additional layers

In R2022b, you can generate C or C++ code that does not depend on third-party libraries for these additional layers:

- `globalMaxPooling2dLayer` (Deep Learning Toolbox)
- `globalAveragePooling2dLayer` (Deep Learning Toolbox)
- `averagePooling2dLayer` (Deep Learning Toolbox)
- `depthConcatenationLayer` (Deep Learning Toolbox)
- `flattenLayer` (Deep Learning Toolbox)
- `focalLossLayer` (Computer Vision Toolbox)
- `anchorBoxLayer` (Computer Vision Toolbox)
- `rcnnBoxRegressionLayer` (Computer Vision Toolbox)
- `ssdMergeLayer` (Computer Vision Toolbox)

See Networks and Layers Supported for Code Generation.

In addition, Spatio-Temporal data propagation support is added for generic C/C++ code generation. You can now pass 2D image sequences with both spatial and time dimensions to these layers and generate generic C/C++ code:

- `convolution2dLayer` (Deep Learning Toolbox)
- `maxPooling2dLayer` (Deep Learning Toolbox)
- `averagePooling2dLayer` (Deep Learning Toolbox)
- `globalMaxPooling2dLayer` (Deep Learning Toolbox)
- `globalAveragePooling2dLayer` (Deep Learning Toolbox)

Improved performance of generated generic C/C++ code

In R2022b, the generated generic C/C++ code (that does not depend on third-party libraries) for the following layers has improved performance:

- `convolution2dLayer` (Deep Learning Toolbox)
- `fullyConnectedLayer` (Deep Learning Toolbox)

In addition, generated code for certain network that contains convolutional layers followed by ReLU or Leaky ReLU layer is likely to have improved performance.

Functionality being removed or changed

coder.getDeepLearningLayers function is not recommended
Still runs

`coder.getDeepLearningLayers` is not recommended. Use `analyzeNetworkForCodegen` instead.

For more information, see `analyzeNetworkForCodegen`.

Code generation behavior change for dlarray inputs and outputs
Behavior change

In R2022b, the generated code creates structures for the `dlarray` inputs and outputs of entry-point functions. `Data` is a public field that you can directly access it.

In previous releases, the generated code uses class to represent the `dlarray` inputs and outputs of entry-point functions. In these releases, you use the initializing function `init` to access the `Data` field. This example shows the difference in the generated code between the two releases:

MATLAB Code	R2022a Generated Code	R2022b Generated Code
`% entry-point function`	`// File: dlarray.h (generated)`	`// File: foo_types.h (generated)`
`function out = foo(a)`	`namespace coder {`	`namespace coder {`
`    out = dims(a);`	`class FOO_DLL_EXPORT dlarray {`	`struct dlarray {`
`end`	`public:`	`  double Data[20];`
	`  void init(const double b_Data[20]);`	`};`
`% code generation`	`  dlarray();`	`}`
`cfg = coder.config('dll');`	`  ~dlarray();`	
`cfg.TargetLang = 'C++';`		
`codegen -config cfg  foo -args dlarray(ones(5,4), 'SC')`	`private:`	`// File: main.cpp (generated)`
	`  double Data[20];`	`static void argInit_dlarray(coder::dlar`
	`};`	`{`
	`}`	
		`  argInit_5x4_real_T(result->Data);`
		`}`
	`// File: main.cpp (generated)`	
	`static void argInit_dlarray(coder::dlarray *result)`	
	`{`	
	`  double dv[20];`	
	`  argInit_5x4_real_T(dv);`	
	`  result->init(dv);`	
	`}`	

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2022a

Supported Functions

Code generation for more MATLAB functions

- `integral`
- `interpft`
- `ldl`
- `mustBeFloat`
- `mustBeNonmissing`
- `mustBeScalarOrEmpty`
- `mustBeText`
- `mustBeTextScalar`
- `mustBeVector`
- `groupcounts`
- `groupsummary`

Code generation for more toolbox functions

In R2022a, you can generate code for many additional toolbox functions and objects. For a list of all functions and objects that are supported for code generation, see:

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

Generated Code Improvements

Generate C++11 enumeration classes for MATLAB enumerations

In R2022a, when you generate C++11 MEX or standalone code for your MATLAB enumerations, the code generator produces C++11 enumeration classes by default. Using enumeration classes makes the generated C++11 code more idiomatic. They also improve code quality because:

- The enumerators are contained in the scope of the enumeration and thus avoid name clashes.
- The enumerators do not implicitly convert to the integer type that might cause unexpected behavior.

To instruct the code generator to produce ordinary C enumeration for a particular MATLAB enumeration class, include the static method `generateEnumClass` that returns `false` in the implementation of that MATLAB enumeration class. See the example below.

MATLAB Code	R2021b Generated Code	R2022a Generated Code
`classdef MyEnumClass < int32`	`enum MyEnumClass : int`	`enum class MyEnumClass : int`
`    enumeration`	`{`	`{`
`        Red(0),`	`  Red = 0, // Default value`	`  Red = 0, // Default value`
`        Blue(1),`	`  Blue,`	`  Blue,`
`        Green(2)`	`  Green`	`  Green`
`    end`	`};`	`};`
`end`		
	`enum MyEnumClass16 : short`	`enum MyEnumClass16 : short`
`classdef MyEnumClass16 < int16`	`{`	`{`
`    enumeration`	`  Orange = 0, // Default value`	`  Orange = 0, // Default value`
`        Orange(0),`	`  Yellow,`	`  Yellow,`
`        Yellow(1),`	`  Pink`	`  Pink`
`        Pink(2)`	`};`	`};`
`    end`		
	`void xEnumIsCppEnumClass(MyEnumClass *out1,`	`void xEnumIsCppEnumClass(MyEnumClass *o`
`    % particular enum opting out`	`  MyEnumClass16 *out2)`	`    MyEnumClass16 *out2)`
`    methods(Static)`	`{`	`{`
`        function y = generateEnumClass()`	`  *out1 = Green;`	`  *out1 = MyEnumClass::Green;`
`            y = false;`	`  *out2 = Pink;`	`  *out2 = Pink;`
`        end`	`}`	`}`
`    end`		
`end`		
`function [out1, out2] = xEnumIsCppEnumClass`		
`out1 = MyEnumClass.Green;`		
`out2 = MyEnumClass16.Pink;`		
`end`		

See Code Generation for Enumerations and Customize Enumerated Types in Generated Code.

Compatibility Considerations

You can change the default behavior of the code generator to produce ordinary C enumerations for all MATLAB enumerations in your code, similar to previous releases. Do one of the following:

- In the code generation configuration object, set the `CppGenerateEnumClass` property to `false`.

- In the MATLAB Coder app, in the **Generate** step, on the **Code Appearance** tab, clear the **Generate C++ enum class from MATLAB enumeration** check box.

Improvement to generated C++ code that uses externally specified enumerations

The code generator allows you to provide your own C++ implementation of a specific MATLAB enumeration in a header file. In R2022a, if you place such a MATLAB enumeration `myEnum` inside the package `pkg`, code generation preserves the name of this enumeration and places it inside the namespace `pkg` in the generated C++ code. Therefore, in the header file that you provide, you must define this enumeration inside the namespace `pkg`.

In previous releases, the generated code named the enumeration as `pkg_enum` and did not place it inside a namespace. With this new behavior, the C++ code generated for externally specified enumerations matches with the C++ code generated for enumerations in other situations. In addition, the current behavior produces code that is better organized and easier to read and use.

MATLAB Code	R2021b Generated C++11 Code	R2022a Generated C++11 Code
`% File: +pkg1\MyEnum.m`	`// File: Header.h (you provide)`	`// File: Header.h (you provide)`
`classdef(Enumeration) MyEnum < int32`	`#pragma once`	`#pragma once`
`    enumeration`	`typedef enum pkg1_MyImportedEnum {`	`namespace pkg1 {`
`        Red(0),`	`    Red = 0,`	`enum class MyEnum : int`
`        Blue(1),`	`    Blue = 1,`	`{`
`        Green(2)`	`    Green = 2`	`  Red = 0, // Default value`
`    end`	`}  MyImportedEnum ;`	`  Blue,`
`    methods(Static)`		`  Green`
`        function y = getHeaderFile()`	`// File: foo.cpp (generated)`	`};`
`            y = 'Header.h';`	`pkg1_MyEnum foo()`	`}`
`        end`	`{`	
`    end`	`  return Red;`	`// File: foo.cpp (generated)`
`end`	`}`	`pkg1::MyEnum foo()`
		`{`
`% File: foo.m`		`  return pkg1::MyEnum::Red;`
`function out = foo`		`}`
`  out = pkg1.MyEnum.Red;`		
`end`		

See Customize Enumerated Types in Generated Code and Code Generation for Enumerations.

Compatibility Considerations

In previous releases, the generated code did not place an externally specified enumeration in a namespace. Instead, the name of the MATLAB package was prefixed to the name of the enumeration in the generated code. As a result, you had to implement the C++ enumeration in the header file differently compared to the current release, as illustrated in the above example. To use your legacy header files with the code generated in R2022a, modify them based on this example.

Additional improvements to generated C++11 code

In R2022a, in most situations, the code generator produces more concise and idiomatic C++11 code that uses these language constructs:

- Empty constructors and destructors are denoted by using the `= default` syntax.
- Type aliases are created by using the `using` keyword instead of the `typedef` keyword. For declaring aliases for anonymous structures, the generated code now uses the `struct` keyword instead of `typedef`.

Examples:

R2021b Generated Code	R2022a Generated Code

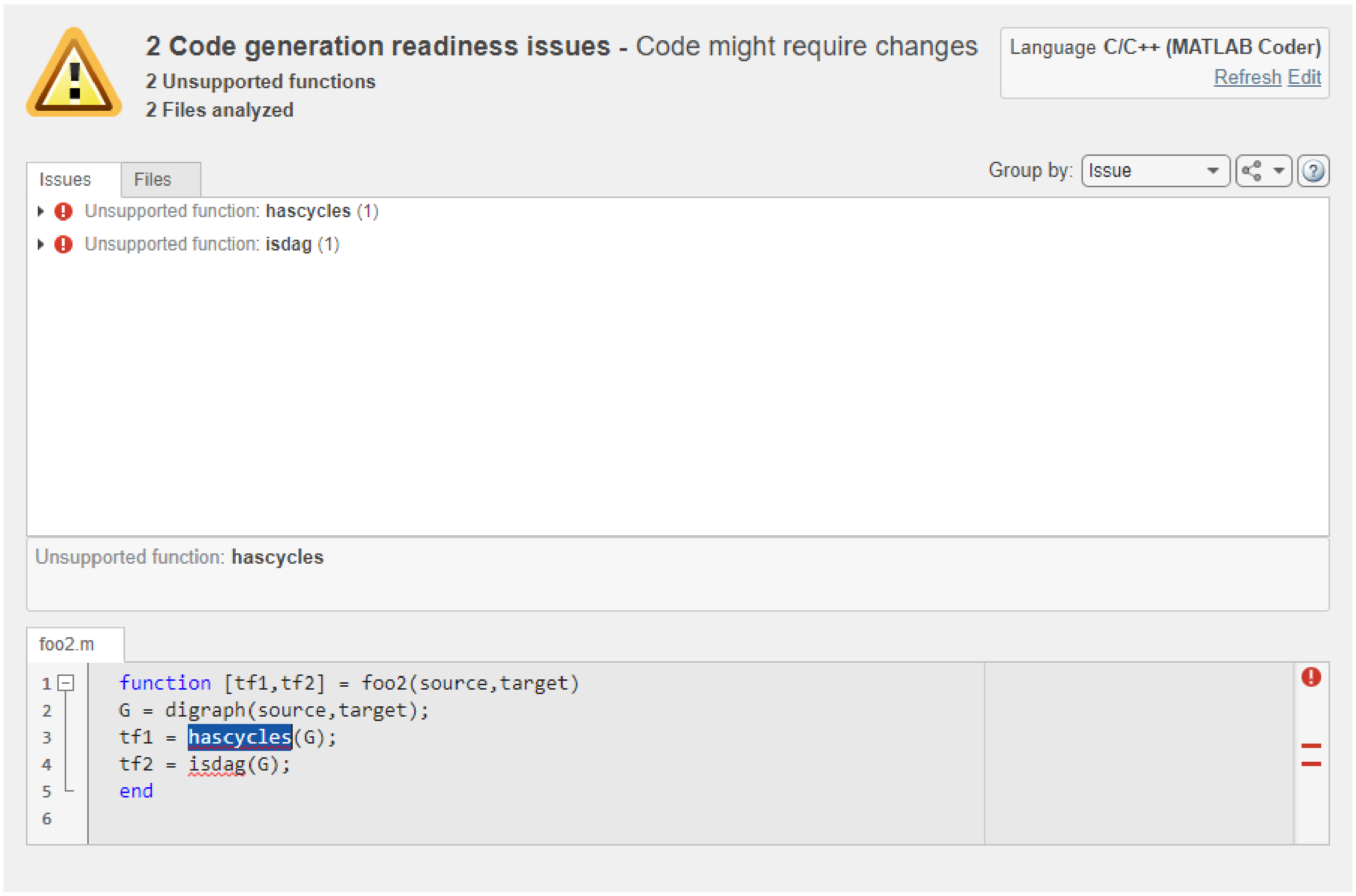
`class foo {`
`   foo();`
`}`
`foo::foo() { }`
	`class foo {`
	`   foo() = default;`
	`}`
	
	Or
	
	`class foo {`
	`   foo();`
	`}`
	`foo::foo() = default;`

`typedef int32 myInt;`
`typedef aStruct myStruct1;`
`typedef struct { ... } myStruct2;`
	`using myInt = int32;`
	`using myStruct1 = aStruct;`
	`struct myStruct2 { ... };`

Code Generation Workflow

New Code Generation Readiness Tool: View more information and navigate through readiness results more easily

In R2022a, the Code Generation Readiness Tool has a new user interface, more information, additional functionality, and improved navigation. In addition, you can now use the Code Generation Readiness Tool in MATLAB Online™.


2 Code generation readiness issues - Code might require changes
2 Unsupported functions
2 Files analyzed
Language C/C++ (MATLAB Coder)
Refresh Edit
Issues
Files
Group by: Issue
Unsupported function: hascycles (1)
Unsupported function: isdag (1)
Unsupported function: hascycles
foo2.m

```
function [tf1,tf2] = foo2(source,target)
G = digraph(source,target);
tf1 = hascycles(G);
tf2 = isdag(G);
end
```

In addition to the existing functionalities, you can now:

- View your MATLAB code inside the Code Generation Readiness Tool. When you select an issue, the part of your MATLAB code that caused this issue gets highlighted.
- Group the readiness results either by issue or by file.
- Select the language that the code generation readiness analysis uses.
- Refresh the code generation readiness analysis if you updated your MATLAB code.
- Export the analysis report either as plain text file or as a `coder.ScreenerInfo` object in the base workspace.

See:

- Check Code by Using the Code Generation Readiness Tool
- `coder.screener` and `coder.ScreenerInfo Properties`

coder.ScreenerInfo object: Access code generation readiness information programmatically

In R2022a, you can export the code generation readiness information about your MATLAB code to a variable in your base workspace. This variable contains a `coder.ScreenerInfo` object whose properties contain information about:

- MATLAB files analyzed by the Code Generation Readiness Tool
- Code generation readiness messages
- Calls to functions not supported for code generation

To export code generation readiness information about your the files `foo1.m`, `foo2.m`, and `foo3.mlx` to the variable `info` in your base workspace, execute this function call:

```
info = coder.screener('foo1.m','foo2.m','foo3.mlx')
```

You can also export the entire report to a MATLAB string by executing the object function `textReport`:

```
reportString = textReport(info)
```

See:

- Reference pages: `coder.ScreenerInfo Properties` and `coder.screener`
- Example: Access Code Generation Readiness Results Programmatically

MATLAB Coder Interface for Visual Studio Code Debugging

If you install the support package MATLAB Coder Interface for Visual Studio Code Debugging, you can use Visual Studio Code as the graphical user interface for these debuggers:

- MinGW GDB on Windows
- GDB on Linux
- LLDB on macOS

For information about installing the support package, in MATLAB Central™ File Exchange, search for `MATLAB Coder Interface for Visual Studio Code Debugging`.

For information about debugger support, see Debug Generated Code During SIL Execution (Embedded Coder).

Generated MEX: UTF-8 system encoding on Windows platform

In R2022a, MATLAB uses UTF-8 as its system encoding on Windows platform. As a result, system calls made from within a generated MEX function now accept and return UTF-8 encoded strings. By contrast, the code generated by MATLAB Coder encodes text data by using the encoding specified by the Windows locale. So, if your MATLAB entry-point function uses `coder.ceval` to call external C/C ++ functions that assume a different system encoding, then the generated MEX function might produce garbled text. If this happens, you must update the external C/C++ functions to handle this situation.

See “MEX Functions: UTF-8 system encoding on Windows platforms”.

Performance

SIMD code for reduction operations

In R2022a, you can generate SIMD code for reduction operations by using the new parameter **Optimize reductions**. The generated code uses the reduction operations from the instruction set that you specify by using the **Instruction set extensions** parameter.

You can generate SIMD code for these operations:

- Sum
- Product
- Minimum
- Maximum
- Handwritten loops for the previous operations

For more information, see Generate SIMD Code for MATLAB Functions.

Parallelization of for-loops performing reduction operations

In R2022a, you can parallelize `for`-loops performing reduction operations by using the configuration parameter **Optimize reductions**.

During a reduction operation, the current iteration value depends on the previous iteration value of the same variable in the `for`-loop. For example, consider the MATLAB function `addition`, which computes the sum of first `n` numbers.

```
function y = addition(n)
    y = 0;
    for i = 1:n
        y = y + i; % for-loop performing reduction operation
    end
end
```

The configuration property parallelizes only arithmetic reduction operations, such as addition (+), subtraction (-), and product (*), in a `for`-loop.

To enable automatic parallelization of reduction operations, use either of these settings:

- Set the `EnableAutoParallelization` property and `OptimizeReductions` property to `true`.

  ```
  cfg = coder.config('lib');
  cfg.EnableAutoParallelization = true;
  cfg.OptimizeReductions = true;
  ```
- Open the MATLAB Coder app. On the **Speed** tab, select the **Enable automatic parallelization** option, and then select **Optimize reductions** option.

For more information, see Classification of Variables in `parfor`-Loops and Automatically Parallelize `for` Loops in Generated Code

Minimized variable scope for C99 (ISO) code generation

In R2022a, variables and functions are declared closer to their usage in the generated code when the target language is specified as C99 (ISO) to improve code readability.

This table shows the declaration of variables in R2021b generated code and R2022a generated code with the target language as C99 (ISO). In the R2022a generated code, the scope of array `zee` is minimized because it is placed inside the `if` block.

MATLAB Function	R2021b Generated Code	R2022a Generated Code
`function y = C99example(n) %#codegen`	`double C99example(int n)`	`double C99example(int n)`
`    y = zeros(1,n);`	`{`	`{`
`    for i = 1:n`	**`  double zee[8];`**	`  double u;`
`        y(i) = 42;`	`  double u;`	`  int i;`
`    end`	`  int i;`	`  if (n < 40) {`
`end`	`  if (n < 40) {`	**`    double zee[8];`**
	`    for (i = 0; i < 8; i++) {`	`    for (i = 0; i < 8; i++) {`
	`      zee[i] = 1.0;`	`      zee[i] = 1.0;`
	`    }`	`    }`
	`    for (i = 0; i < n; i++) {`	`    for (i = 0; i < n; i++) {`
	`      zee[n - 1] += zee[0];`	`      zee[n - 1] += zee[0];`
	`    }`	`    }`
	`    u = zee[0] + zee[n - 1];`	`    u = zee[0] + zee[n - 1];`
	`  } else {`	`  } else {`
	`    u = 3.0;`	`    u = 3.0;`
	`  }`	`  }`
	`  return u;`	`  return u;`
	`}`	`}`

Deep Learning with MATLAB Coder

TensorFlow Lite: Generate C++ code for pretrained models and deploy on Linux platforms

In R2022a, you can use the `loadTFLiteModel` (Deep Learning Toolbox) function to load a pretrained TensorFlow Lite model into a `TFLiteModel` (Deep Learning Toolbox) object. Use this object with the `predict` (Deep Learning Toolbox) function to perform inference with a pretrained TensorFlow Lite model. You can generate code for this functionality and deploy on Linux platforms either on your MATLAB host computer or on ARM processors.

To use this functionality, you must install the Deep Learning Toolbox Interface for TensorFlow Lite. For more information, see Prerequisites for Deep Learning with TensorFlow Lite Models (Deep Learning Toolbox). For an example, see Generate Code for TensorFlow Lite Model and Deploy on Raspberry Pi (Deep Learning Toolbox).

CMSIS-NN Library: Generate code for quantized deep learning layers and deploy on ARM Cortex-M targets

You can generate C static library code for networks containing these layers that uses the CMSIS-NN library and performs inference computations in 8-bit integers:

- Fully connected layer (`fullyConnectedLayer` (Deep Learning Toolbox)).

Your deep learning network can also contain the following layers. The generated code performs computations for these layers in 32-bit floating point type.

- Long short-term memory layer (`lstmLayer` (Deep Learning Toolbox))
- Softmax layer (`softmaxLayer` (Deep Learning Toolbox)).
- Input and output layers

C code generation for such quantized deep learning networks supports `SeriesNetwork` (Deep Learning Toolbox) objects and `DAGNetwork` (Deep Learning Toolbox) objects that can be converted to `SeriesNetwork` objects. The generated code takes advantage of the CMSIS-NN library version 5.7.0 and can be integrated into your project as a static library that you can deploy to a variety of ARM Cortex-M CPU platforms. See:

- Help topic: Code Generation for Quantized Deep Learning Networks
- Example: Generate Code for Quantized LSTM Network and Deploy on Cortex-M Target

Generate generic C/C++ code for dlnetwork workflows

Starting in R2022a, you can generate C or C++ code for the `predict` function of a `dlnetwork` (Deep Learning Toolbox) object inside an entry-point function. You can also generate code for the `dlarray` (Deep Learning Toolbox) data type that you use for inference with `dlnetwork`. The generated code does not depend on any third-party libraries.

Code generation support includes:

- Construction of formatted and unformatted `dlarray`

- Passing `dlarray` to entry-point functions and returning `dlarray` from entry-point functions
- Invoking a subset of functions on `dlarray` objects, including the object functions `softmax` (Deep Learning Toolbox), `sigmoid` (Deep Learning Toolbox), and `fullyconnect` (Deep Learning Toolbox)
- Passing formatted `dlarray` to the `dlnetwork` predict function inside an entry-point function

For more information, see Code Generation for dlarray.

Code generation from MATLAB for dlnetwork objects that contain image sequences

Starting in R2022a, you can generate code for a `dlnetwork` (Deep Learning Toolbox) object that has image sequence inputs. Code generation includes:

- A `dlarray` (Deep Learning Toolbox) input containing image sequences that have `'SSCT'` or `'SSCBT'` data formats.
- Multi-input `dlnetwork` with heterogeneous input layers.

For more information, see `dlnetwork` (Deep Learning Toolbox).

Deep Learning Arrays: Generate code for more functions that use dlarray

In R2022a, you can generate code for additional MATLAB functions that use `dlarray` (Deep Learning Toolbox) inputs. Code generation includes:

- Binary math operations — Use `power` to perform binary element-wise power (`.^`) operation.
- Other math operations — Perform matrix multiplication by using `mtimes`. Use `pagemtimes` to perform page-wise matrix multiplication.

Generate C++ code that performs inference computations in 8-bit integers for more layers

In R2022a, you can generate C++ code for these layers that uses the ARM Compute Library and performs inference computations in 8-bit integers:

- `averagePooling2dLayer` (Deep Learning Toolbox)
- `fullyConnectedLayer` (Deep Learning Toolbox)

See Code Generation for Quantized Deep Learning Networks.

Deep Learning Networks: Generate code for additional networks

Code generation by using the Intel MKL-DNN library supports these additional networks:

- `yolov3ObjectDetector` (Computer Vision Toolbox) – YOLO v3 object detector. This feature requires the functions in the Computer Vision Toolbox Model for YOLO v3 Object Detection support package.

- `yolov4ObjectDetector` (Computer Vision Toolbox) – YOLO v4 object detector. This feature requires the functions in the Computer Vision Toolbox Model for YOLO v4 Object Detection support package.
- `pointPillarsObjectDetector` – PointPillars network to detect objects in lidar point clouds. This feature requires the Lidar Toolbox™.

Code generation by using the ARM Compute library supports these additional networks:

- `yolov3ObjectDetector` (Computer Vision Toolbox) – YOLO v3 object detector. This feature requires the functions in the Computer Vision Toolbox Model for YOLO v3 Object Detection support package.
- `yolov4ObjectDetector` (Computer Vision Toolbox) – YOLO v4 object detector. This feature requires the functions in the Computer Vision Toolbox Model for YOLO v4 Object Detection support package.
- `pointPillarsObjectDetector` – PointPillars network to detect objects in lidar point clouds. This feature requires the Lidar Toolbox.

Deep Learning: Generate code for additional layers

In R2022a, C++ code generation that uses the Intel MKL-DNN library supports these additional layers:

- `nnet.keras.layer.ClipLayer`
- `nnet.keras.layer.PreluLayer`
- `nnet.keras.layer.TimeDistributedFlattenCStyleLayer`
- `nnet.onnx.layer.ClipLayer`
- `nnet.onnx.layer.GlobalAveragePooling2dLayer`
- `nnet.onnx.layer.PreluLayer`
- `nnet.onnx.layer.SigmoidLayer`
- `nnet.onnx.layer.TanhLayer`

In R2022a, C++ code generation with the ARM Compute library supports these additional layers:

- `nnet.keras.layer.ClipLayer`
- `nnet.keras.layer.PreluLayer`
- `nnet.keras.layer.TimeDistributedFlattenCStyleLayer`
- `nnet.onnx.layer.ClipLayer`
- `nnet.onnx.layer.GlobalAveragePooling2dLayer`
- `nnet.onnx.layer.PreluLayer`
- `nnet.onnx.layer.SigmoidLayer`
- `nnet.onnx.layer.TanhLayer`

In R2022a, you can generate C or C++ code that does not depend on any third-party libraries for these additional layers:

- `batchNormalizationLayer` (Deep Learning Toolbox)
- `nnet.keras.layer.ClipLayer`

- `nnet.keras.layer.PreluLayer`
- `nnet.keras.layer.TimeDistributedFlattenCStyleLayer`
- `nnet.onnx.layer.ClipLayer`
- `nnet.onnx.layer.GlobalAveragePooling2dLayer`
- `nnet.onnx.layer.PreluLayer`
- `nnet.onnx.layer.SigmoidLayer`
- `nnet.onnx.layer.TanhLayer`

See Networks and Layers Supported for Code Generation.

Improved performance of generated generic C/C++ code

In R2022a, the generated generic C/C++ code (that does not depend on third-party libraries) for the following layers has improved performance:

- `bilstmLayer` (Deep Learning Toolbox)
- `convolution2dLayer` (Deep Learning Toolbox)
- `fullyConnectedLayer` (Deep Learning Toolbox)
- `gruLayer` (Deep Learning Toolbox)
- `lstmLayer` (Deep Learning Toolbox)

In addition, you can generate generic C/C++ code that uses SIMD intrinsics for these layers. Use of SIMD intrinsics is likely to further improve the performance the generated code. To generate code that uses SIMD intrinsics, do one of the following:

- Specify a code replacement library that supports SIMD, for example, GCC ARM Cortex-A. To specify a code replacement library, set the code generation configuration parameter `CodeReplacementLibrary`. Alternatively, in the MATLAB Coder app, in the project build settings, on the **Custom Code** tab, set the **Code replacement library** parameter.
- Specify SIMD instruction set for the target hardware by setting the code generation configuration parameter `InstructionSetExtensions`. Alternatively, in the MATLAB Coder app, in the project build settings, on the **Speed** tab, set the **Leverage target hardware instruction set extensions** parameter.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2021b

MATLAB Programming for Code Generation

Implicit Expansion: Generate code for element-wise operations and functions with automatic expansion of operand dimensions

In R2021b, you can generate code for MATLAB operators and functions that apply implicit expansion. These binary element-wise operators and functions implicitly expand their inputs to be the same size, if the input arrays have compatible sizes. Two arrays have compatible sizes if, for every dimension, the dimension sizes of the arrays are either the same or one of them is one. See Compatible Array Sizes for Basic Operations.

For example, you can calculate the mean of each column in a matrix `A`, and then subtract the vector of mean values from each column by using `A - mean(A)`. The generated code for this operation is shown below.

MATLAB Code

```
function out = meanSubtraction(A)
out = A - mean(A);
end

a = coder.typeof(1,[Inf Inf])
codegen -config:lib -report meanSubtraction -args {a}
```

Generated Code with Implicit Expansion

```
static void binary_expand_op(emxArray_real_T *out, co
                             const emxArray_real_T *y
{
  int aux_0_1;
  ....
  for (i = 0; i < loop_ub; i++) {
    b_loop_ub = A->size[0];
    for (i1 = 0; i1 < b_loop_ub; i1++) {
      out->data[i1 + out->size[0] * i] =
          A->data[i1 + A->size[0] * aux_0_1] - y->dat
    }
    aux_1_1 += stride_1_1;
    aux_0_1 += stride_0_1;
  }
}
void meanSubtraction(const emxArray_real_T *A, emxArr
{
  emxArray_real_T *y;
  ....
  if (A->size[1] == y->size[1]) {
    ....
      for (lastBlockLength = 0; lastBlockLength < nbl
        out->data[lastBlockLength + out->size[0] * xp
            A->data[lastBlockLength + A->size[0] * xp
            y->data[xpageoffset] / (double)A->size[0]
      }
    }
  } else {
    binary_expand_op(out, A, y);
  }
  emxFree_real_T(&y);
}
```

If your MATLAB code includes operators or functions that apply implicit expansion, the generated code includes a secondary function that performs implicit expansion, in this case `binary_expand_op`. The function generated for the main operation (`meanSubtraction`) calls the secondary operation. See Generate Code With Implicit Expansion Enabled.

Compatibility Considerations

In R2021b, by default, code generation supports implicit expansion. The code generator introduces modifications in the generated code for implicit expansion. These modifications might cause the code generated for functions that support implicit expansion to look and perform differently as compared to the code from previous releases. New errors might be generated due to mismatch of output sizes or types. For more information on how to control this feature, see “Code generation behavior change due to implicit expansion” on page 4-3.

This feature does not change the generated code for operations or functions applied on constant or fixed-size inputs.

Generate code for MATLAB code that uses class aliases

In R2021b, you can generate C/C++ code for MATLAB code that uses class aliases. When you need to change a MATLAB class name, you can create an alias to preserve compatibility with the code written before the name change. Once defined, you can use the aliases anywhere you use the class name.

See “Class Aliasing: Create aliases for renamed classes to maintain backward compatibility”.

Access name of currently running MATLAB function during debugging by using coder.mfunctionname

In R2021b, you can access the name of the currently running MATLAB function either in the generated code or in MATLAB execution by inserting a call to `coder.mfunctionname` in the body of your MATLAB function. For example, when debugging either your MATLAB code or the generated code, you can use this functionality to print the name of the currently running function.

Functionality being removed or changed

Code generation behavior change due to implicit expansion

In R2021b, the code generated for element-wise binary functions and operations that support implicit expansion might appear and perform differently as compared to the code from previous releases. The code generator introduces modifications in the generated code to perform implicit expansion. The changes in the generated code might result in excess code to expand the operands. In addition, the expansion of the operands might affect the performance of the generated code.

Implicit expansion might change the size of the outputs from the supported operators and functions causing size and type mismatch errors in your workflow. This feature does not change the generated code for operations or functions on constant or fixed-size inputs.

For more information, see Generate Code With Implicit Expansion Enabled and Optimize Implicit Expansion in Generated Code.

Supported Functions

Expanded code generation for tables and timetables

In R2021b, code generation supports more capabilities and MATLAB toolbox functions when you use tables and timetables.

The supported functions for tables and timetables are:

- `innerjoin`
- `mergevars`
- `outerjoin`

For more information, see Code Generation for Tables and Code Generation for Timetables.

Code generation for more MATLAB functions

- `digraph`
- `graph`
- `hess`
- `ode78`
- `ode89`

Code generation for more toolbox functions

In R2021b, you can generate code for many additional toolbox functions and objects. For a list of all functions and objects that are supported for code generation, see:

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

These are links to the release notes of some toolboxes that added code generation support in R2021b:

Computer Vision Toolbox

See Generate C and C++ Code Using MATLAB Coder: Support for functions.

Image Processing Toolbox

See C Code Generation: Generate code from five functions using MATLAB Coder.

Signal Processing Toolbox

See C/C++ Code Generation Support: Code generation for filtering, spectral analysis, and vibration analysis.

Wavelet Toolbox

See C/C++ Code Generation: Automatically generate code for wavelet functions.

Generated Code Improvements

Generate C++11 enumerations that specify underlying type

C++11 and subsequent C++ standards enable you to specify the underlying type of an enumeration, just like MATLAB does. In addition, to comply with `AUTOSAR C++14 Rule A7-2-2` (Polyspace Bug Finder), the underlying types of enumerations in your C++ code must be explicitly defined.

If you set the target language standard to `'C++11 (ISO)'`, the code generator now converts a MATLAB enumeration class to a C++ enumeration that explicitly defines the underlying type.

In the previous release, the C++11 code generated for MATLAB enumerations had the same appearance as the generated C++03 code in R2021b. The current behavior produces C++11 code that is easier to read and use.

See Code Generation for Enumerations.

MATLAB Code	R2021a Generated Code	R2021b Generated Code

`classdef Bearing < int16`
`   enumeration`
`      North (0)`
`      East  (90)`
`      South (180)`
`      West  (270)`
`   end`
`end`
 	
`classdef Vertical < int32`
`enumeration`
`    Up(0)`
`    Down(1)`
`end`
`end`
	`typedef short Bearing;`
	
	`// enum Bearing`
	`const Bearing North = 0;`
	`const Bearing East = 90;`
	`const Bearing South = 180;`
	`const Bearing West = 270;`
	
	`enum Vertical`
	`{`
	`  Up = 0, // Default value`
	`  Down`
	`};`
		`enum Bearing : short`
		`{`
		`  North = 0, // Default value`
		`  East = 90,`
		`  South = 180,`
		`  West = 270`
		`};`
		
		`enum Vertical : int`
		`{`
		`  Up = 0, // Default value`
		`  Down`
		`};`

Compatibility Considerations

In the previous release, the representation of the enumerated type in generated C++11 code depended on the base type of the MATLAB enumeration:

- If the base type was the native integer type for the target platform (for example, `int32`), the code generator produced a C++ 11 enumeration. The generated C++11 enumeration did not contain an explicit specification of the underlying type.
- If the base type was different from the native integer type, the MATLAB enumeration members were converted to constants in the generated C++11 code.

In R2021b, irrespective of the base type, the MATLAB enumeration is converted to a C++11 enumeration. In addition, the C++11 enumeration explicitly specifies the underlying type.

Code Generation Workflow

Specify custom hardware targets during code generation

The code generator enables you to extend the range of supported hardware by using the `target.create` and `target.add` functions to register new devices. In R2021b, after you register a new device, you can create a `coder.Hardware` object for the device that contains the hardware board parameters for C/C++ code generation from MATLAB code.

```
hw = coder.hardware('My New Device')
```

To use this object `hw` for code generation, assign it to the `Hardware` property of a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object that you pass to `codegen`.

```
cfg = coder.config('lib');
cfg.Hardware = hw;
```

The registered device also appears as an option on the **Hardware** tab of the MATLAB Coder app. If you use the app to generate code, you can specify the device directly from the drop-down list.

See `coder.hardware` and Register New Hardware Devices.

If you have Embedded Coder®, you can now set up connectivity between MATLAB and your custom target hardware and run processor-in-the-loop (PIL) simulations on the target. See Set Up PIL Connectivity by Using `target` Package (Embedded Coder).

Performance

SIMD code generation for Intel hardware

In R2021b, you can generate single instruction, multiple data (SIMD) code from MATLAB code by using Intel SSE technology. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms. To generate code that uses SIMD intrinsics, set the new configuration parameter **Leverage target hardware instruction set extensions** to SSE2. This parameter is on the Speed pane. If you have Embedded Coder, you can generate code that uses additional SIMD instruction sets. For more information, see Generate SIMD Code for MATLAB Functions.

C Code Generation: Generate portable C code that has improved performance for five functions

You can generate portable C code that has faster execution speed than in previous releases for these functions:

- `hsv2rgb`
- `imadjust`
- `imfill`
- `imfilter`
- `imreconstruct`

The optimizations include multithreading and algorithm improvements. Generating portable C code requires MATLAB Coder.

Generate optimized code by unrolling parallel for loops

In R2021b, the code generator uses the configurable **Loop unrolling threshold** value to determine whether to automatically unroll parallel for-loops (`parfor-loops`).

When the code generator unrolls a `parfor-loop`, it produces a copy of the loop body for each iteration. For a small number of loop iterations that perform some simple calculation, parallelization is inefficient as it introduces overheads, which includes time taken for thread creation, data synchronization between threads, and thread deletion. Unrolling the loops that have a large number of iterations can significantly increase code generation time and generate inefficient code.

The default value of the **Loop unrolling threshold** is 5. By modifying the threshold, you can fine-tune loop unrolling. To modify the threshold, use either of these steps:

- In a configuration object for standalone code generation, set the `LoopUnrollThreshold` property.
- In the MATLAB Coder app, on the **Speed** tab, set **Loop unrolling threshold**.

Eliminated dead code lines containing variable indices

In R2021a, the code generated from a MATLAB function contained dead code involving variable indices. In R2021b, the code generator identifies the dead code that has a variable index and eliminates those. Eliminating the dead code or unnecessary data copies conserves RAM and ROM consumption, and improves execution speed.

Consider the MATLAB function `mLoopDeadCode`.

```
function B = mLoopDeadCode(A, idx)
    B = zeros(size(A));
    B(idx) = 2 * A(idx);
    B(idx) =  3 + A(idx);
end
```

In R2021a, the code generator produced this C code:

```
void mLoopDeadCode(const double A[4], double idx, double B[4])
{
  double B_tmp;
  B[0] = 0.0;
  B[1] = 0.0;
  B[2] = 0.0;
  B[3] = 0.0;
  B_tmp = A[(int)idx - 1];
  B[(int)idx - 1] = 2.0 * B_tmp;
  B[(int)idx - 1] = B_tmp + 3.0;
}
```

The code contained a dead code line involving the variable index `(int)idx - 1`. This line was executed but the result was never used.

In R2021b, the code generator produced this C code:

```
void mLoopDeadCode(const double A[4], double idx, double B[4])
{
  B[0] = 0.0;
  B[1] = 0.0;
  B[2] = 0.0;
  B[3] = 0.0;
  B[(int)idx - 1] = A[(int)idx - 1] + 3.0;
}
```

The generated code does not contain the dead code line and unnecessary data copies. The code generator identifies the dead code that has a variable index and eliminates that. This optimization improves RAM and ROM consumption and execution speed.

Improved execution speed through common subexpression elimination

In R2021a, the code generated from a MATLAB function contained redundant subexpressions that were used to repeatedly cast the data type of the same expression value. In R2021b, the generated code uses a temporary variable to hold the value of these subexpressions, which eliminates redundant conversions of data type. This optimization improves the execution speed of the generated code.

This table compares the code generated in R2021b with the code generated in R2021a. For the comparison, use the supporting files that are present in the working folder of the example Generate C ++ Classes for MATLAB® Classes That Model Simple and Damped Oscillators.

MATLAB Code	R2021a Generated Code	R2021b Generated Code

`function [time,position] = evolution(obj,initialPosition,`
`initialVelocity,timeInterval,timeStep)`
`  numSteps = floor(timeInterval/timeStep);`
`  n = numSteps + 1;`
`  position = zeros(n,1);`
`  time = zeros(n,1);`
`  position(1) = initialPosition;`
`  for i = 1:numSteps`
`    position(i+1) = obj.dynamics(initialPosition,`
`    initialVelocity,i*timeStep);`
`        time(i+1) = i*timeStep;`
`            end`
`        end`
	`void simpleOscillator::evolution`
	`(double initialPosition, double`
	`  initialVelocity, double timeInterval,`
	`  double timeStep, coder::array<double,`
	`  1U> &b_time, coder::array<double, 1U> &position)`
	`  {`
	`    double numSteps;`
	`    int i;`
	`    int loop_ub;`
	`    numSteps = std::floor(timeInterval / timeStep);`
	`    position.set_size((static_cast<int>(numSteps + 1.0)));`
	`    loop_ub = static_cast<int>(numSteps + 1.0);`
	`    for (i = 0; i < loop_ub; i++) {`
	`      position[i] = 0.0;`
	`    }`
	
	`    b_time.set_size((static_cast<int>(numSteps + 1.0)));`
	`    loop_ub = static_cast<int>(numSteps + 1.0);`
	`    for (i = 0; i < loop_ub; i++) {`
	`      b_time[i] = 0.0;`
	`    }`
		`void simpleOscillator::evolution`
		`(double initialPosition, double initial`
		`    double timeInterval, double timeS`
		`    coder::array<double, 1U> &b_time,`
		`    coder::array<double, 1U> &positio`
		`{`
		`  double numSteps;`
		`  int i;`
		`  int loop_ub_tmp;`
		`  numSteps = std::floor(timeInterval /`
		`  loop_ub_tmp = static_cast<int>(numSte`
		`  position.set_size(loop_ub_tmp);`
		`  b_time.set_size(loop_ub_tmp);`
		`  for (i = 0; i < loop_ub_tmp; i++) {`
		`    position[i] = 0.0;`
		`    b_time[i] = 0.0;`
		`  }`

In R2021a, the generated code contained subexpressions to repeatedly cast the data type of the same expression `(numSteps + 1.0)`. In R2021b, the generated code uses the temporary variable `loop_ub_tmp` for holding the value of the subexpressions, thereby eliminating the redundancy.

Generation of vectorized MEX code in JIT compilation mode

In R2021b, when you use just-in-time compilation for MEX code generation with the memory integrity checks disabled, the code generator generates vectorized code. To generate vectorized code for integer type and for loops, in addition to memory integrity checks, you must disable integer saturation and responsiveness checks respectively. This optimization improves execution speed of the generated MEX code. For more information, see Control Run-Time Checks.

Optimized dynamic array access

In R2021b, a new configuration parameter **CacheDynamicArrayDataPointer** is introduced in MATLAB Coder to improve the run-time performance of dynamic arrays. It hoists the data pointer to a temporary variable and uses this temporary variable to access the matrix data.

By default, the parameter is enabled for MEX, static library, dynamic linked library, and executable configurations.

To disable the parameter, do one of the following:

- In a code generation configuration object, set the `CacheDynamicArrayDataPointer` property to false.
- Alternatively, open the MATLAB Coder app. On the **Advanced** tab, deselect the **Cache dynamic array data** option.

Limitation:

This parameter is not supported for C++ coder::array.

For more information, see Optimize Dynamic Array Access.

Specify threads to parallelize for and parfor-loops

In R2021b, you can control the number of threads required to execute parallel loops in the C/C++ code that you generate from MATLAB code. The cross-compilation enables you to generate code on the host machine and execute it on the target machine.

The table shows a `for` and `parfor`-loop example to generate C/C++ code from MATLAB code by using the `codegen` command.

MATLAB Function	Commands To Generate Code	C/C++ Generated Code

`for`**-loop example**
 	
`function y = forExample(n) %#codegen`
`    y = zeros(1,n);`
`    for i = 1:n`
`        y(i) = 42;`
`    end`
`end`
	`n = 1000;`
	`cfg = coder.config('lib');`
	`cfg.EnableAutoParallelization = true;`
	`cfg.NumberOfCpuThreads = 3;`
	
	`codegen -config cfg forExample -args {n} -report`
		`#pragma omp parallel for num_threads(3`
		`omp_get_max_threads() ? omp_get_max_thr`
		`    for (b_i = 0; b_i < i; b_i++) {`
		`      y->data[b_i] = 42.0;`
		`    }`

`parfor`**-loop example**
 	
`function y = parforExample(n) %#codegen`
`    y = ones(1,n);`
`    parfor(i = 1:n)`
`        y(i) = i;`
`    end`
`end`
	`n = 1000;`
	`cfg = coder.config('lib');`
	`cfg.NumberOfCpuThreads = 4;`
	
	`codegen -config cfg parforExample -args {n} -report`
		`#pragma omp parallel for num_threads(4`
		` omp_get_max_threads() ? omp_get_max_th`
		
		`    for (i = 0; i <= ub_loop; i++) {`
		`      y->data[i] = (double)i + 1.0;`
		`  }`

The following table lists the ways to set number of threads required to parallelize `for`-loops in the generated code and their precedence order.

Precedence	Options to Set Number of Threads	Description
1	**Parfor** (for only `parfor`-loop)	`parfor (k = 1:10, 6)`
2	**Configuration property/option:** `NumberOfCpuThreads`	`cfg.NumberOfCpuThreads = 8;`

3	**Target Processor property/option:** `NumberOfCores`, `NumberOfThreadsPerCore`	`processor.NumberOfCores = 4;`
		`processor.NumberOfThreadsPerCore = 2;`

If you do not select any precedence order, then the number of threads is set to `omp_get_max_threads()`, which returns a maximum number of available threads during run time.

For more information, see Specify Maximum Number of Threads to Run Parallel `for`-Loops in the Generated Code.

Deep Learning with MATLAB Coder

Deep Learning Workflow: Update network parameters after code generation

In R2021b, you can update learnable and state parameters of deep learning networks without regenerating code for the network. You can update the network parameters for `SeriesNetwork`, `DAGNetwork` and `dlnetwork` objects. Use the `coder.regenerateDeepLearningParameters` function to regenerate files containing network learnables and states parameters. Parameter update supports MEX and standalone code generation for the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) and the ARM Compute libraries.

See:

- Help topic: Update Network Parameters After Code Generation
- Example: Post-Code-Generation Update of Deep Learning Network Parameters

Deep Learning Arrays: Generate code for more functions that use dlarray

In R2021b, you can generate code for additional MATLAB functions that use `dlarray` (Deep Learning Toolbox) inputs. Code generation support includes:

- Unary math operations — Find the inverse tangent by using `atan2`.
- Binary math operations — Use `minus(-)`, `plus(+)`, `rdivide(./)`, and `times(.*)` to perform binary element-wise math operations.
- Reduction operations — Perform reduction operations on `dlarray` by using `mean`, `prod`, and `sum`.
- Comparison operations — Use `max` and `min` to find the maximum or minimum elements of a single `dlarray` or between two formatted `dlarray` inputs.
- Indexing operations — Use `colon`, `:` for indexing into a `dlarray`.
- Logical operations — Use functions such as `and` and `eq` to perform logical operations on the data within `dlarray`. For other supported logical operations, see Logical Operations.
- Size manipulation functions — Manipulate the dimensions of a `dlarray` by using `reshape` and `squeeze`.
- Transposition operations — Use `ctranspose`, `permute`, `ipermute`, and `transpose` to transpose `dlarray` matrices.
- Concatenation functions — Concatenate deep learning arrays by using `cat`, `horzcat`, and `vertcat`.
- Conversion functions — Change the underlying `dlarray` data type by using the `cast` function.
- Size identification functions — Query the dimensions of the `dlarray` data by using `iscolumn`, `ismatrix`, `isrow`, `isscalar`, and `isvector`.

Custom Layers: Use dlarray in deep learning networks that have custom layers

You can now generate code for custom deep learning layers that use deep learning arrays. Custom layer code generation supports unformatted and formatted `dlarray` (Deep Learning Toolbox) for

MEX and standalone workflows. For other usage notes and limitations of custom layers with `dlarray`, see Supported Layers.

Code generation from MATLAB for dlnetwork that contains sequences

In R2021b, you can generate code for `dlnetwork` (Deep Learning Toolbox) that have vector sequence inputs. Code generation support includes:

- `dlarray` (Deep Learning Toolbox) containing vector sequences that have `'CT'` or `'CBT'` data formats.
- A `dlnetwork` object that has multiple inputs. For ARM Compute, the `dlnetwork` can have sequence and non-sequence input layers. For Intel MKL-DNN, input layers must be all sequence input layers.

For more information, see `dlnetwork` (Deep Learning Toolbox).

Generate generic C/C++ code for more deep learning layers

In R2021b, you can generate C or C++ code that does not depend on any third-party libraries for these additional deep learning layers:

- `clippedReluLayer` (Deep Learning Toolbox)
- `concatenationLayer` (Deep Learning Toolbox)
- `convolution2dLayer` (Deep Learning Toolbox)
- `eluLayer` (Deep Learning Toolbox)
- `groupNormalizationLayer` (Deep Learning Toolbox)
- `leakyReluLayer` (Deep Learning Toolbox)
- `maxPooling2dLayer` (Deep Learning Toolbox)
- `scalingLayer` (Reinforcement Learning Toolbox)
- `nnet.keras.layer.FlattenCStyleLayer`
- `nnet.keras.layer.GlobalAveragePooling2dLayer`
- `nnet.keras.layer.SigmoidLayer`
- `nnet.keras.layer.TanhLayer`
- `nnet.keras.layer.ZeroPadding2dLayer`
- `nnet.onnx.layer.ElementwiseAffineLayer`
- `nnet.onnx.layer.FlattenInto2dLayer`
- `nnet.onnx.layer.FlattenLayer`
- `nnet.onnx.layer.IdentityLayer`
- `nnet.onnx.layer.VerifyBatchSizeLayer`

See Networks and Layers Supported for Code Generation.

Deploy generic C/C++ code on ARM Cortex-M processors

In R2021b, you can deploy generic C/C++ code that does not depend on any third-party libraries on STMicroelectronics® Discovery boards and STMicroelectronics Nucleo boards that use ARM Cortex®-

M processors. For deployment on these devices, you must install one of these two support packages and the corresponding required products, as described in the support package documentation:

- For deployment on STMicroelectronics Discovery boards, install the Embedded Coder Support Package for STMicroelectronics Discovery Boards (https://www.mathworks.com/hardware-support/st-discovery-board.html).

  Supported boards:

  - STM32F746G-Discovery
  - STM32F769I-Discovery
  - STM32F4-Discovery
- For deployment on STMicroelectronics Nucleo boards, install the Simulink Coder Support Package for STMicroelectronics Nucleo Boards (https://www.mathworks.com/hardware-support/st-nucleo.html).

  Supported boards:

  - Nucleo-F401RE
  - Nucleo-F103RB
  - Nucleo-F302R8
  - Nucleo-F031K6
  - Nucleo-L476RG
  - Nucleo-L053R8
  - Nucleo-F746ZG
  - Nucleo-F411RE
  - Nucleo-F767ZI
  - Nucleo-H743ZI/Nucleo-H743ZI2

For an example application, see Generate Code for LSTM Network and Deploy on Cortex-M Target .

Generate C++ code that performs inference computations in 8-bit integers for more layers

In R2021b, you can generate C++ code for these layers that uses the ARM Compute Library and performs inference computations in 8-bit integers:

- `maxPooling2dLayer` (Deep Learning Toolbox)
- `reluLayer` (Deep Learning Toolbox)

See Code Generation for Quantized Deep Learning Networks.

Generate C++ code that uses third-party libraries for more deep learning layers

In R2021b, C++ code generation that uses the Intel MKL-DNN library or the ARM Compute library supports this additional layer:

- `groupNormalizationLayer` (Deep Learning Toolbox)
- `nnet.onnx.layer.FlattenInto2dLayer`
- `nnet.onnx.layer.VerifyBatchSizeLayer`

See Networks and Layers Supported for Code Generation.

Functionality being removed or changed

cnncodegen Function: Support for CPU targets removed

Errors

In R2021b, the `cnncodegen` function does not generate C++ code for Intel and ARM CPU targets. To generate C++ code for deep learning layers and networks for these targets, use the `codegen` function.

See Code Generation for Deep Learning Networks with MKL-DNN and Code Generation for Deep Learning Networks with ARM Compute Library.

Support for ARM Compute library versions 18.11 and 19.02 removed

Errors

In R2021b, generation of C++ code that uses versions 18.11 or 19.02 of the ARM Compute library is no longer supported.

See:

- `coder.ARMNEONConfig`
- Prerequisites for Deep Learning with MATLAB Coder

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2021a

MATLAB Programming for Code Generation

Generate code for inherited constructors

In R2021a, the code generated for the default constructor of a subclass in your MATLAB code has the same run-time behavior as the MATLAB execution.

The run-time behavior of the default constructor of a MATLAB class follows these rules:

- If the class inherits from one or more parent classes, the default constructor forwards all its arguments to the constructor of the first parent class. In cases of multiple inheritence, the default constructor does not forward the input arguments to any of the other parent classes.
- If the class does not inherit from any class, the default constructor does not accept any input arguments.

In previous releases, the code generated for the default constructor of a class did not accept any input arguments, irrespective of whether or not this class inherited from a parent class.

See Implicit Call to Inherited Constructor.

Generate code for name=value syntax for passing name-value arguments

In R2021a, you can generate code for the `name=value` syntax for passing name-value arguments to MATLAB functions.

In MATLAB, the syntax `foo(name=value)` is equivalent to the syntax `foo("name",value)`. You can include multiple name-value arguments in your MATLAB function call. The name-value arguments must appear *after* all the other arguments in the call.

Code generation does not support function argument validation.

For more information on the `name=value` syntax, see Name-Value Arguments.

Generate code for property information functions isprop and properties

In R2021a, you can generate code for these functions that provide information about class properties:

- `isprop`
- `properties`

Code generation usage notes and limitations for the `properties` function:

- The function call `properties(obj)` is supported for code generation only if `obj` is an instance of a MATLAB class defined in a `.m` file.
- The order in which the code generated for the function `properties` returns the properties of an object might be different from MATLAB execution.

Supported Functions

Expanded code generation for categorical arrays

In R2021a, code generation supports more MATLAB toolbox functions when you use `categorical` arrays.

The supported functions for `categorical` arrays are:

- `issorted`
- `issortedrows`
- `sort`
- `sortrows`

For more information, see Code Generation for Categorical Arrays.

Expanded code generation for tables and timetables

In R2021a, code generation supports more capabilities and MATLAB toolbox functions when you use tables and timetables.

The supported functions for tables are:

- `issortedrows`
- `join`
- `renamevars`
- `rows2vars`
- `sortrows`
- `splitvars`
- `stack`
- `unstack`
- `varfun`

The supported functions for timetables are:

- `issorted`
- `issortedrows`
- `join`
- `renamevars`
- `sortrows`
- `splitvars`
- `stack`
- `unstack`
- `varfun`

For more information, see Code Generation for Tables and Code Generation for Timetables.

Code generation for more MATLAB functions

- `convhulln`
- `delaunayn`
- `logm`
- `sylvester`
- `voronoin`

Code generation for more toolbox functions

In R2021a, you can generate code for many additional toolbox functions and objects. For a list of all functions and objects that are supported for code generation, see:

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

These are links to the release notes of some toolboxes that added code generation support in R2021a:

Computer Vision Toolbox

See "Extended Capability: Perform GPU and C/C++ code generation" (Computer Vision Toolbox)

Image Processing Toolbox

See "C Code Generation: Generate code from the adapthisteq function using MATLAB Coder" (Image Processing Toolbox)

Signal Processing Toolbox

See "C/C++ Code Generation Support: Code generation for filtering, signal modeling, spectral analysis, and statistics" (Signal Processing Toolbox).

Statistics and Machine Learning Toolbox

- See "Generate C/C++ code for performing incremental learning using linear regression or binary classification model functions (requires MATLAB Coder)" (Statistics and Machine Learning Toolbox)
- See "Generate C/C++ code for prediction by using a machine learning model with heterogeneous data (requires MATLAB Coder)" (Statistics and Machine Learning Toolbox)

Wavelet Toolbox

See "C/C++ Code Generation: Automatically generate code for wavelet functions" (Wavelet Toolbox)

Generated Code Improvements

Multisignature MEX support for multiple entry-point functions

In R2021a, you can generate one MEX function for multiple entry-point functions containing multiple signatures during code generation. This one MEX function reduces the overhead involved in generating separate MEX functions for different entry-point functions. The generated MEX function works with all the entry-point functions and all the signatures provided during code generation.

Suppose that you want to generate a MEX function from multiple entry-point functions `myAdd` and `myMul` that works with these entry-point functions for three different data types: `double`, `int8`, and `int16`. Specify the three arguments as: `{1,2}`, `{int8(1), int8(2)}`, and `{int16(1), int16(2)}`.

To generate code for `myAdd` and `myMul` functions, at the MATLAB command prompt, run this `codegen` command:

```
codegen -config:mex myAdd.m -args {1,2} -args {int8(1),int8(2)} myMul.m -args {1,2} -args {int16(1),int16(2)} -o 'myMath'
```

This syntax generates one MEX function `myMath` for all the signatures that you specified in the `codegen` command.

You can verify the output values by using the generated MEX function `myMath` at the command prompt. Make sure that the values you pass to `myMath` match the input properties that you specified before code generation.

```
myMath("myAdd",3,4)

ans =

     7

myMath("myAdd",int8(5),int8(6))

ans =

    int8
    11

myMath("myMul",3,4)

ans =

     12

myMath("myMul",int16(5),int16(6))

ans =

    int16
    30
```

For more information, see Generate One MEX Function That Supports Multiple Signatures.

Catch and handle exceptions for run-time errors that the generated standalone C++ code throws

In R2021a, if you generate standalone C++ code with run-time error detection and reporting enabled, the generated code throws `std::runtime_error` exceptions for the run-time errors. When you call the generated C++ entry-point functions, you can catch and handle these exceptions by using a `try-catch` block in your external C++ code.

For example, consider this MATLAB function:

```
function y = foo(x)
y = sqrt(x);
end
```

Generate C++ code for the function `foo` that accepts double scalar inputs. Enable run-time error detection and reporting in the generated code.

```
cfg = coder.config('dll');
cfg.RuntimeChecks = true;
codegen -config cfg -lang:c++ foo -args 1 -report
```

The code generator produces a C++ function that has signature `double foo(double x)`, which both accepts and returns double scalar values. Because the `sqrt` function returns real outputs for only nonnegative inputs, the generated code produces an error for negative input values. The function `rtErrorWithMessageID` defined in the generated file `foo.cpp` throws the exception for this error.

```
static void rtErrorWithMessageID(const char *b, const char *aFcnName,
                                 int aLineNum)
{
  std::stringstream outStream;
  ((outStream << "Domain error. To compute complex results from real x, use \'")
   << b)
      << "(complex(x))\'.";
  outStream << "\n";
  ((((outStream << "Error in ") << aFcnName) << " (line ") << aLineNum) << ")";
  throw std::runtime_error(outStream.str());
}
```

See Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors.

Compatibility Considerations

In previous releases, the generated error reporting function used `fprintf` to write error messages to `stderr`. Then it used `abort` to terminate the application. In R2021a, the error reporting function throws a `std::runtime_error` exception instead.

C++11 as default language standard for generated C++ code

In R2021a, the default language standard is set to `'C++11' (ISO)` for generated C++ code and `'C99' (ISO)` for generated C code. The default value of the language standard setting `TargetLangStandard` is set to `'Auto'`.

The language standard assigned is dependent on the language specified in the `TargetLang` setting of your `coder.EmbeddedCodeConfig` or `coder.CodeConfig` object. If `TargetLang` is set to `'C+`

+', the 'Auto' setting assigns 'C++11' (ISO) as the language standard. If TargetLang is set to 'C', 'C99' (ISO) is assigned as the language standard.

Compatibility Considerations

In R2021a, TargetLangStandard is set to 'Auto' by default. If your existing workflows require a specific language standard, set the TargetLangStandard to the standard required by your project.

Improvement to C++ code generated for enumerations with nonnative size

In R2021a, if your enumeration class derives from a class that is different from the native integer type for the target platform (for example, int32), the corresponding enumeration members are converted to constants. These constants belong to the namespace that contains the enumeration type definition in the generated C++ code.

In previous releases, the enumeration members that had nonnative size in your MATLAB code were converted to macros defined by using the #define directive and were not contained in a namespace. The current behavior produces code that is better organized and easier to read and use.

MATLAB Code	R2020b Generated Code	R2021a Generated Code

This enumeration is defined inside the package pkg:
 	
`classdef(Enumeration) MyColorsAddClassName < int16`
`    enumeration`
`        Purple(0),`
`        Orange(1),`
`        Yellow(2)`
`    end`
 	
`    methods(Static)`
`        function y = addClassNameToEnumNames()`
`            y = true;`
`        end`
`    end`
`end`
	`// Type Definitions`
	`namespace pkg`
	`{`
	`  typedef short MyColorsAddClassName;`
	`}`
	
	`#ifndef pkg_MyColorsAddClassName_constants`
	`#define pkg_MyColorsAddClassName_constants`
	
	`// enum pkg_MyColorsAddClassName`
	`#define pkg_MyColorsAddClassName_Purple ((::pkg::MyColorsAddClassNam`
	`#define pkg_MyColorsAddClassName_Orange ((::pkg::MyColorsAddClassNam`
	`#define pkg_MyColorsAddClassName_Yellow ((::pkg::MyColorsAddClassNam`
	`#endif //pkg_MyColorsAddClassName_constants`
		`// Type Definitions`
		`namespace pkg {`
		`typedef short MyColorsAddClassName;`
		
		`// enum pkg_MyColorsAddClassName`
		`const MyColorsAddClassName MyColorsAddC`
		`const MyColorsAddClassName MyColorsAddC`
		`const MyColorsAddClassName MyColorsAddC`
		
		`} // namespace pkg`

See Code Generation for Enumerations.

Compatibility Considerations

If your entry-point function accepts or returns an enumeration that has a nonnative size, modify your legacy C++ function that calls the generated entry-point function to use the new symbols for the enumeration members (for example, pkg::MyColorsAddClassName_Purple instead of pkg_MyColorsAddClassName_Purple).

Generate UTF-8 encoded C/C++ files that work across locales and platforms

In R2021a, the generated C/C++ code files and the comments in these files use UTF-8 encoding. You can use these files across locales and platforms.

Code generation supports Unicode characters only in the comments in your MATLAB code. If you use Unicode characters in string or character literals, the code generator produces an error.

Bundled CXSparse files in generated code

In R2021a, the generated `CXSparse` files are bundled into two files, one for real computations and one for complex computations. These files are in the `CXSparse` folder with the name `cs_ri_bundle.cpp`, which contains code for real computations and `cs_ci_bundle.cpp`, which contains code for complex computations. This bundle build also applies to code generated in C.

This feature improves the build times of the code generator for MEX and standalone builds and simplifies the parsing of the build file structure.

Compatibility Considerations

In previous releases, if you wanted to include the `CXSparse` files in your project, you included the individual files in your code. To now reconcile your workflows, you include the bundled files in your project.

Code Generation Workflow

Improved Representations for Coder Type Objects

Starting in R2021a, the representation of coder type objects with `coder.typeof()` and `coder.newtype()` is more succinct and excludes internal state values. This feature is applicable for the following types:

- `categorical`
- `datetime`
- `dlarray`
- `duration`
- `table`
- `timetable`

The following table shows a few examples of the new representations of these classes and objects passed into `coder.typeof()`:

Coder Type Object	New Representation
`categorical`	`matlab.coder.type.CategoricalType` `0x0 categorical` `Categories : 0x0 homogeneous cell` `Ordinal : 1x1 logical` `Protected : 1x1 logical`
`duration`	`matlab.coder.type.DurationType` `1x1 duration` `Format : 1x8 char`
`table`	`matlab.coder.type.TableType` `0x0 table` `Data : 1x0 homogeneous cell` `Description : 1x0 char` `UserData : 0x0 double` `DimensionNames : {'Row'}    {'Variables'}` `VariableNames :` `VariableDescriptions : 1x0 homogeneous cell` `VariableUnits : 1x0 homogeneous cell` `VariableContinuity : 0x0 double` `RowNames : 0x0 homogeneous cell`

In the new representations, non-constant properties display their type and size, while constant properties display their values. The object properties can be edited as needed. You can assign scalar values to object properties. The values are implicitly converted to coder type values. Values assigned to constant properties are implicitly converted to constants.

Compatibility Considerations

The new representation of coder types can affect your workflows. If you require the legacy representation of your coder type, use the `getCoderType` function on the variable that has the new representation of your class or object.

For example, to get the legacy representation of a `table`, use the variable that has the new representation `tt` to call the `getCoderType` function:

```
t = table;
tt = coder.typeof(t);
ttLegacy = tt.getCoderType()
```

In the Coder Type Editor, the code generator includes the function `getCoderType` for these coder types. You use this function to return the legacy representation of coder types.

Configuration Parameter Dialog Box: New layout and added functionalities

In R2021a, the configuration parameter dialog box that you use to edit properties of `coder.MexCodeConfig`, `coder.CodeConfig`, `coder.EmbeddedCodeConfig`, and other related code configuration objects has a new layout. The dialog box also has certain added functionalities including improved search functionality, more informative tooltips, and option for generating equivalent MATLAB script. See Specify Configuration Parameters in Command Line Workflow Interactively.

Display status of code generation at command line

In R2021a, when generating code by using the `codegen` command, you can display messages indicating the status of the code generation process at the MATLAB command line. To configure this behavior, set the `Verbosity` property in a code configuration object or use the equivalent `codegen` command option.

Verbosity =	Equivalent codegen Option	Behavior
`'Silent'`	`-silent`	If code generation succeeds without warning, all messages are suppressed, including when you generate a report. Warning and error messages are displayed.
`'Info'` (default)	This is the default behavior. No explicit option.	Compared to the `'Silent'` mode, if code generation succeeds, these additional messages are displayed: • `Code generation successful` • Link to the generated report, if any
`'Verbose'`	`-v`	In addition to the messages shown in the `'Info'` mode, code generation status and target build log messages are displayed.

Compatibility Considerations

In previous releases, if you did not generate a report, the default behavior of successful code generation was to not print any message. In R2021a, the default behavior of the code generator is to print a message saying code generation is successful.

To get the previous default behavior of the `codegen` command, add the `-silent` option.

Format generated code by using clang-format

In R2021a, the `CodeFormattingTool` flag enables you to choose how to format the code. This flag has these settings:

- `Clang-format`: The code generator formats your code by using `clang-format`.
- `Auto`: Uses an internal heuristic to determine if the generated code is formatted by `clang-format` or a MathWorks® formatting tool. To determine whether your code is formatted by `clang-format`, in a `coder.config` object, set the `Verbosity` option to `'Verbose'`.
- `MathWorks`: Causes the code generator to revert to the legacy code formatting setting.

The `CodeFormattingTool` setting is available for all configuration objects, namely `coder.EmbeddedCodeConfig`, `coder.CodeConfig`, and `coder.MexCodeConfig`.

Compatibility Considerations

In R2021a, the default formatting of the generated code might change. If this causes issues in your workflow, set the `CodeFormattingTool` under your configuration object as `MathWorks`.

More options to specify multiple entries in code configuration objects

In R2021a, there are additional options to specify multiple file names, paths, or reserved names as string arrays and a cell array of character vectors in the code configuration objects. For example, to specify multiple reserved names, you can execute this command:

```
cfg = coder.config('lib');
cfg.ReservedNameArray = ["name1","name2","name3"];
cfg.ReservedNameArray
```

```
ans = 

  1×3 string array

    "name1"    "name2"    "name3"
```

These configuration properties work for multiple entries in code configuration objects as string arrays or a cell array of character vectors:

- `CustomInclude`
- `CustomLibrary`
- `CustomSource`
- `ReservedNameArray`
- `ReplacementTypes.HeaderFiles`

In previous releases, you could specify only character vectors in a code configuration object.

Functionality being removed or changed

Capability to specify multiple entries in code configuration objects by using character vector will be removed

Behavior change in future release

In a future release, specifying multiple file names, paths, or reserved names in code configuration objects by using character vectors or string scalars that have delimiters will be removed. Use string arrays and a cell array of character vector instead.

The Verbose property of code configuration objects to be removed

Behavior change in future release

In a future release, the `Verbose` property of `coder.CodeConfig` and `coder.EmbeddedCodeConfig` will be removed.

To configure the code generation progress display, use the `Verbosity` property of these objects.

Target hardware data management

R2021a provides these `target` package enhancements for target hardware data management.

Function	Enhancement
`target.remove`	If you specify the name-value argument, `'IncludeAssociations', true`, the function removes the specified target object and associated objects from an internal database. The function does not remove an associated object if it is referenced by other target objects. The function displays information about the removed objects, which you can suppress by using the name-value argument, `'SuppressOutput', true`.For more information, see `target.remove`.
`target.add`	The function displays information about the objects that it adds to an internal database. The function also returns a vector that contains the added objects. You can suppress the text output by using the name-value argument, `'SuppressOutput', true`. For more information, see `target.add`.
`target.export`	When you run the function generated by `target.export`, it returns the registered target object and associated target objects. Previously, the function did not return associated target objects. For more information, see `target.export`.

Support Package for NVIDIA Jetson and NVIDIA DRIVE platforms

In R2021a, you can use the MATLAB Coder Support Package for NVIDIA® Jetson and NVIDIA DRIVE Platforms to communicate with, deploy, and run code for the ARM CPUs on NVIDIA platforms such as Jetson and DRIVE. To download the support package, use the Add-on Explorer. For more information on the supported workflows, see MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms.

Performance

Multithreading capabilities for more Image Processing Toolbox functions

In R2021a, if you use a compiler that supports the Open Multiprocessing (OpenMP) application interface, you can generate multithreaded C/C++ functions for some Image Processing Toolbox™ functions that are included in MATLAB code. This enhancement improves the function execution speed.

The new optimized functions that have multithreading capabilities are:

- `bwlabel`
- `houghpeaks`
- `otsuthresh`
- `bwareaopen`
- `bwboundaries`
- `imbilatfilt`

The code generator generates multithreaded code for the `houghpeaks` function only when the function takes a large Hough transform matrix as an input. For more information, see Algorithm Acceleration Using Parallel for-Loops (parfor).

Automatic parallelization of for loops in generated code

In R2021a, you can generate parallel `for` loops automatically in the code you generate from your MATLAB code. Automatic parallelization of a section of code might significantly improve the execution speed of the generated code.

Automatic parallelization of `for` loops supports these build types for C/C++ targets: MEX, static library, dynamically linked library, and executable.

This table compares the code generated in R2021a with automatic parallelization enabled with the code generated in R2020b.

MATLAB Code	R2020b Generated Code	R2021a Generated Code

`function [x, y] = autoparExample(x, y)`
`%#codegen`
`% Automatic parallelization example - explicit and implicit loops.`
 	
`% Explicit for-loop.`
`for i = 10:numel(x)`
`    x(i) = sqrt(x(i));`
`end`
 	
`% Generates implicit for-loop.`
`y = y * 17;`
`end`
	`void autoparExample(double x[2000], double y[2000])`
	`{`
	`  int i;`
	`  /*  Explicit for-loop.  */`
	`  for (i = 0; i < 1991; i++) {`
	`    x[i + 9] = sqrt(x[i + 9]);`
	`  }`
	`  /*  Generates implicit for-loop.  */`
	`  for (i = 0; i < 2000; i++) {`
	`    y[i] *= 17.0;`
	`  }`
	`}`
		`void autoparExample(double x[2000], dou`
		`{`
		`  int i;`
		`  /*  Explicit for-loop.  */`
		`#pragma omp parallel for num_threads(om`
		
		`  for (i = 0; i < 1991; i++) {`
		`    x[i + 9] = sqrt(x[i + 9]);`
		`  }`
		`  /*  Generates implicit for-loop.  */`
		`#pragma omp parallel for num_threads(om`
		
		`  for (i = 0; i < 2000; i++) {`
		`    y[i] *= 17.0;`
		`  }`
		`}`

To automatically generate parallel `for` loops, do one of the following:

- In a code generation configuration object, set the `EnableAutoParallelization` property to `true`.
- Alternatively, in the MATLAB Coder app, on the **Speed** tab, select the **Enable automatic parallelization** option.

You might want to disable automatic parallelization for a particular loop if that loop performs better in serial execution. To do this, place the `coder.loop.parallelize('never')` pragma immediately before an explicit `for` loop in your MATLAB code. This pragma overrides the global `EnableAutoParallelization` setting and prevents parallelization of that loop. For example, even if automatic parallelization is enabled, the code generator does not parallelize this loop:

```
% Pragma to disable automatic parallelization of for-loops
coder.loop.parallelize('never');
for i = 1:n
    y(i) = y(i)*sin(i);
end
```

See Automatically Parallelize `for` Loops in Generated Code and `coder.loop.parallelize`.

More optimized inlining behavior of public methods of generated C++ classes

In R2021a, the code generator uses a more refined set of rules to determine whether to inline a public method in the generated C++ code. Except in one special situation, the new rules are identical to the rules for inlining ordinary functions.

The same inlining rules apply to ordinary functions and public methods in these situations:

- The body of the function or the method contains an explicit `coder.inline('always')` or `coder.inline('never')` directive. This directive gets the highest precedence.
- You set the code configuration property `InlineBetweenUserFunctions` or the equivalent code generation setting **Inline between user functions** in the MATLAB Coder app to `'Never'`, `'Speed'`, or `'Always'`.

- A call to a method appears inside another method of the same class.

In a special situation, inlining a public method in the generated C++ code changes a private property in your MATLAB code to a public property in the generated code and breaks data encapsulation. For example, suppose that a public method `myMethod` that uses a private property `prop` of the object is called by an entry-point function. If `myMethod` is inlined in the generated code, the property `prop` must be visible from outside the object and changed to a public property. To limit this occurrence, if `InlineBetweenUserFunctions` is set to `'Readability'`, the code generator does not inline the public method calls that appear outside the class definition.

In previous releases, the code generator had a more restrictive strategy for inlining of public methods. The code generator did not inline a public method call unless the method contained the `coder.inline('always')` directive.

See:

- Generate C++ Classes for MATLAB Classes
- Control Inlining to Fine-Tune Performance and Readability of Generated Code

Generated code quality improvements

R2021a includes these generated code quality improvements:

- When possible, the code generator converts matrices or arrays that have a single element (for example, `int a[1]` or `int a[1][1][1]`) to scalars (for example, `int a`). This conversion improves the performance and readability of the generated code.
- The generated code creates fewer data copies when performing certain array operations such as shrinking arrays in place, deleting array elements, and concatenating arrays.

Deep Learning with MATLAB Coder

Generate code for convolutional LSTM networks

In R2021a, you can generate code for deep learning networks that contain both a convolution layer and a long short-term memory (LSTM) layer. The generated code can use either the Intel MKL-DNN library or the ARM Compute library.

For an example that generates code for a convolutional LSTM network, see Code Generation for Convolutional LSTM Network That Uses Intel MKL-DNN.

Generate generic C/C++ code for deep learning layers

In R2021a, you can generate C or C++ code that does not depend on any third-party libraries for these deep learning layers:

- Input layers: `featureInputLayer` (Deep Learning Toolbox), `imageInputLayer` (Deep Learning Toolbox), `sequenceInputLayer` (Deep Learning Toolbox)
- Output layers: `classificationLayer` (Deep Learning Toolbox), `regressionLayer` (Deep Learning Toolbox)
- Combination layers: `additionLayer` (Deep Learning Toolbox), `multiplicationLayer` (Deep Learning Toolbox)
- Activation layers: `reluLayer` (Deep Learning Toolbox), `softmaxLayer` (Deep Learning Toolbox), `sigmoidLayer` (Deep Learning Toolbox), `softplusLayer` (Reinforcement Learning Toolbox), `tanhLayer` (Deep Learning Toolbox)
- Input resize and reshape layers: `depthToSpace2dLayer` (Image Processing Toolbox), `resize2dLayer` (Image Processing Toolbox)
- Long short-term memory (LSTM) layers: `lstmLayer` (Deep Learning Toolbox) and `bilstmLayer` (Deep Learning Toolbox)
- `dropoutLayer` (Deep Learning Toolbox)
- `fullyConnectedLayer` (Deep Learning Toolbox)
- `gruLayer` (Deep Learning Toolbox)
- Custom deep learning layers that you create. The layer definition must specify the pragma `%#codegen`. See Define Custom Deep Learning Layer for Code Generation (Deep Learning Toolbox).
- Custom output layers.

See:

- Help topic: Generate Generic C/C++ Code for Deep Learning Networks
- Example: Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning

Generate code for dlnetwork workflows that use deep learning arrays

In R2021a, you can generate code for the `dlarray` (Deep Learning Toolbox) data type that you use for inference with `dlnetwork` (Deep Learning Toolbox). Code generation support includes:

- Construction of formatted and unformatted `dlarray`
- Passing `dlarray` to entry-point functions and returning `dlarray` from entry-point functions
- Invoking a subset of functions on `dlarray` objects, including the object functions `softmax` (Deep Learning Toolbox), `sigmoid` (Deep Learning Toolbox), and `fullyconnect` (Deep Learning Toolbox).
- Passing formatted `dlarray` to the `dlnetwork` predict function inside an entry-point function

See:

- Help topic: Code Generation for dlarray
- Example: Generate Digit Images Using Variational Autoencoder on Intel CPUs

Generate code for convolution layers that performs inference computations in 8-bit integers

In R2021a, you can generate C++ code for these convolution layers that uses the ARM Compute Library and performs inference computations in 8-bit integers:

- 2-D convolution layer (`convolution2dLayer` (Deep Learning Toolbox))
- 2-D grouped convolution layer (`groupedConvolution2dLayer` (Deep Learning Toolbox)). The value of the `NumGroups` input argument must be equal to `2`.

To generate code that performs inference computations in 8-bit integers, in your `coder.ARMNEONConfig` object `dlcfg`, set these additional properties:

```
dlcfg.CalibrationResultFile = 'dlquantizerObjectMatFile';
dlcfg.DataType = 'int8';
```

Alternatively, in the MATLAB Coder app, on the **Deep Learning** tab, set **Target library** to `ARM Compute`. Then set the **Data type** and **Calibration result file path** parameters.

Here `'dlquantizerObjectMatFile'` is the name of the MAT-file that `dlquantizer` (Deep Learning Toolbox) generates for specific calibration data. For the purpose of calibration, set the `ExecutionEnvironment` property of the `dlquantizer` object to `'CPU'`.

See Code Generation for Quantized Deep Learning Network on Raspberry Pi.

Generate code for more layers

In R2021a, C++ code generation that uses the Intel MKL-DNN library or the ARM Compute library supports these additional layers:

- 2-D depth to space layer (`depthToSpace2dLayer` (Image Processing Toolbox))
- Feature input layer (`featureInputLayer` (Deep Learning Toolbox))
- Flatten layer (`flattenLayer` (Deep Learning Toolbox))
- 2-D resize layer (`resize2dLayer` (Image Processing Toolbox))
- Sequence folding layer (`sequenceFoldingLayer` (Deep Learning Toolbox))
- Sequence unfolding layer (`sequenceUnfoldingLayer` (Deep Learning Toolbox))

See Networks and Layers Supported for Code Generation.

Generate code that uses newer versions of ARM Compute and Intel MKL-DNN libraries

In R2021a, you can generate more efficient C++ code for layers and networks that use these newer versions of ARM Compute and Intel MKL-DNN libraries:

- ARM Compute library for computer vision and machine learning, version 20.02.1. See https://developer.arm.com/ip-products/processors/machine-learning/compute-library.
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) v1.4. See https://01.org/dnnl.

See Prerequisites for Deep Learning with MATLAB Coder.

Compatibility Considerations

In R2021a, generation of C++ code that uses these versions of ARM Compute and Intel MKL-DNN libraries is not supported:

- ARM Compute library for computer vision and machine learning, versions 18.05 and 18.08
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) v1.0

Functionality being removed or changed

cnncodegen Function: Support for CPU targets to be removed
Warns

In a future release, the `cnncodegen` function will not generate C++ code for Intel and ARM CPU targets. To generate C++ code for deep learning layers and networks for these targets, use the `codegen` function.

See Code Generation for Deep Learning Networks with MKL-DNN and Code Generation for Deep Learning Networks with ARM Compute Library.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2020b

MATLAB Programming for Code Generation

Generate code for MATLAB code that accesses fields of a structure array

In R2020b, you can generate code for MATLAB code that accesses field data by indexing into a structure array. See Access Data in Structure Array (MATLAB).

Code generation supports field access of fixed-size structure arrays. horzcat containing field access of variable-size structure arrays is also supported in the generated code.

Expansion of the structure array on the left side of the assignment is not supported in generated code.

Supported Functions

Expanded code generation support for eig function

New options of the `eig` function are supported for C/C++ code generation.

Compared to previous releases, code generation in R2020b supports these cases:

- Cholesky factorization specified as `'chol'` for the symmetric generalized eigenvalue problem.
- If you specify the LAPACK library callback class, then the code generator supports these options:
  - The `'balance'` and `'nobalance'` options for the standard eigenvalue problem. For example, you can specify this syntax:

    ```
    [V,D] = eig(A, balanceOption);
    ```
  - The computation of left eigenvectors.

For more information, see `eig`, `coder.LAPACKCallback`, and LAPACK Calls in Generated Code.

Expanded code generation for datetime and duration arrays

In R2020b, code generation supports more MATLAB toolbox functions when you use `datetime` and `duration` arrays.

The supported functions for `datetime` arrays are:

- `colon`
- `datevec`
- `hms`
- `hour`
- `interp1`
- `max`
- `mean`
- `min`
- `minute`
- `posixtime`
- `repmat`
- `ymd`

The supported functions for `duration` arrays are:

- `ceil`
- `colon`
- `floor`
- `interp1`
- `intersect`

- `ismember`
- `issorted`
- `issortedrows`
- `linspace`
- `max`
- `mean`
- `median`
- `min`
- `mode`
- `repmat`
- `setdiff`
- `setxor`
- `sort`
- `sortrows`
- `std`
- `sum`
- `union`
- `unique`

For more information, see Code Generation for Datetime Arrays and Code Generation for Duration Arrays.

Expanded code generation for tables and timetables

In R2020b, code generation supports more capabilities and MATLAB toolbox functions when you use tables and timetables.

Supported table and timetable capabilities are:

- Table and timetable variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).
- Timetables that have `datetime` vectors as row times.

The supported functions for tables are:

- `addvars`
- `convertvars`
- `intersect`
- `ismember`
- `movevars`
- `removevars`
- `setdiff`
- `setxor`

- `union`
- `unique`

The supported functions for timetables are:

- `addvars`
- `cat`
- `convertvars`
- `horzcat`
- `intersect`
- `ismember`
- `movevars`
- `removevars`
- `retime`
- `setdiff`
- `setxor`
- `synchronize`
- `union`
- `unique`

For more information, see Code Generation for Tables and Code Generation for Timetables.

Expanded code generation for categorical arrays

In R2020b, code generation supports more MATLAB toolbox functions when you use `categorical` arrays.

The supported functions for `categorical` arrays are:

- `addcats`
- `cat`
- `countcats`
- `eq`
- `ge`
- `gt`
- `horzcat`
- `histcounts`
- `intersect`
- `ismember`
- `le`
- `lt`
- `max`
- `mergecats`

- `min`
- `ne`
- `removecats`
- `renamecats`
- `reordercats`
- `setcats`
- `setdiff`
- `setxor`
- `union`
- `unique`
- `vertcat`

For more information, see Code Generation for Categorical Arrays.

Code generation for more MATLAB functions

- `besselh`
- `besselk`
- `bessely`
- `getenv`
- `setenv`
- `matches`
- `pcg`
- `ordeig`
- `im2gray`
- `cmap2gray`

Code generation for more toolbox functions

5G Toolbox

- `generate`
- `networkTrafficFTP`
- `networkTrafficOnOff`
- `networkTrafficVoIP`
- `nrOFDMDemodulate`
- `nrOFDMInfo`
- `nrOFDMModulate`
- `nrPRACHOFDMInfo`
- `nrPRACHOFDMModulate`
- `nrResourceGrid`
- `nrTBS`

- `nrULSCHDemultiplex`
- `nrULSCHMultiplex`
- `pcapngWriter`
- `pcapWriter`
- `write`
- `writeCustomBlock`
- `writeGlobalHeader`
- `writeInterfaceDescriptionBlock`

Audio Toolbox

- `acousticFluctuation`
- `audioDelta`
- `audioFeatureExtractor`
- `cepstralCoefficients`
- `vggish`
- `yamnet`

Antenna Toolbox

- `comm.Ray`

Communications Toolbox

- `arrayConfig`
- `bleAngleEstimate`
- `bleAngleEstimateConfig`
- `blePCAPWriter`
- `bluetoothFrequencyHop`
- `comm.ChannelFilter`
- `comm.Ray`
- `comm.RayTracingChannel`
- `getElementPosition`
- `getNumElements`
- `getPayloadLength`
- `getPhyConfigProperties`
- `nextHop`
- `pcapngWriter`
- `pcapWriter`
- `write` of `blePCAPWriter`
- `write` of `pcapWriter` and `pcapngWriter`
- `writeCustomBlock`
- `writeGlobalHeader`

- `writeInterfaceDescriptionBlock`

Computer Vision Toolbox

- `estimateGeometricTransform2D`
- `estimateGeometricTransform3D`
- `insertObjectMask`
- `pcbin`
- `ransac`

DSP System Toolbox

- `getFrequencyVector`

Fixed-Point Designer

- `nnz`

Image Processing Toolbox

- `poly2label`
- `rgbwide2ycbcr`
- `ycbcr2rgbwide`

Navigation Toolbox

- `dynamicCapsuleList` and all its object functions
- `dynamicCapsuleList3D` and its object functions `addEgo`, `addObstacle`, and `checkCollision`
- `gnssSensor`
- `plan` of `plannerAStarGrid`
- `plannerAStarGrid`
- `referencePathFrenet` and all its object functions
- `stateSpaceSE3`
- `trajectoryGeneratorFrenet` and its object function `connect`
- `validatorOccupancyMap3D`
- `wheelEncoderAckermann`
- `wheelEncoderBicycle`
- `wheelEncoderDifferentialDrive`
- `wheelEncoderUnicycle`

Optimization Toolbox

- `fsolve`
- `lsqcurvefit`
- `lsqnonlin`

Robotics System Toolbox

- `analyticalInverseKinematics`
- `gazebogenmsg`
- `generateIKFunction`
- `interpolate`
- `manipulatorRRT`
- `packageGazeboPlugin`
- `plan`
- `shorten`

Sensor Fusion and Tracking Toolbox

- `copy`
- `geoTrajectory`
- `initsingerekf`
- `singer`
- `singerjac`
- `singermeas`
- `singermeasjac`
- `singerProcessNoise`
- `stateinfo`
- `trackerGridRFS`

Signal Processing Toolbox

See "C/C++ Code Generation Support: Generate code for feature extraction, signal measurements, and vibration analysis" (Signal Processing Toolbox).

Wavelet Toolbox

- `ewt`
- `iswt`
- `iswt2`
- `scaleSpectrum`
- `swt`
- `swt2`
- `timeSpectrum`
- `wcoherence`
- `wdenoise2`

WLAN Toolbox

- `generate`
- `getNumPostFECPaddingBits`

- `interpretHESIGBCommonBits`
- `interpretHESIGBUserBits`
- `networkTrafficFTP`
- `networkTrafficOnOff`
- `networkTrafficVoIP`
- `pcapngWriter`
- `pcapWriter`
- `scramblerRange`
- `wlanInterpretScramblerState`
- `wlanNonHTDataBitRecover`
- `wlanNonHTOFDMDemodulate`
- `write`
- `writeCustomBlock`
- `writeGlobalHeader`
- `writeInterfaceDescriptionBlock`

Generated Code Improvements

Generate MEX function that has C++ classes for MATLAB classes

In R2020b, when you generate C++ MEX functions, the default behavior of the code generator produces C++ classes for the classes in your MATLAB code. These include all MATLAB classes such as value classes, handle classes, and system objects. In previous releases, if you set the target language for MEX code generation as C++, the code generator produced structures for the classes in your MATLAB code.

The generated C++ MEX code is now closer to the generated C++ standalone code, which also contains C++ classes for MATLAB classes. This MEX function enables you to create a more reliable prototype that you can test inside the MATLAB environment before you generate and deploy standalone C++ code.

See Generate C++ Classes for MATLAB Classes.

Compatibility Considerations

You can change the default behavior of the code generator to produce C++ code that contains structures for MATLAB classes, similar to previous releases. Do one of the following:

- In a `coder.MexCodeConfig` object, set `TargetLang` to `'C++'` and `CppPreserveClasses` to `false`.
- In the MATLAB Coder app, in the **Generate** step, set **Language** to **C++**. In the project build settings, on the **Code Appearance** tab, clear the **Generate C++ classes from MATLAB classes** check box.

Improved organization of generated C++ code into namespaces

In R2020b, if you generate C++ code from MATLAB code, the code generator default behavior is to:

- Place all MathWorks code (for example, code for the sparse data type) into a separate namespace that has the name `coder`. To change this default name of the namespace (for example, to `myNamespace`), do one of the following:
  - In a code configuration object, set the parameter `CppNamespaceForMathworksCode` to `'myNamespace'`.
  - In the MATLAB Coder app, in the **Generate Code** step, on the **Code Appearance** tab, set the **Namespace for MathWorks Code** parameter to `myNamespace`.
- Generate C++ namespaces for packages in your MATLAB code. If your MATLAB code has nested packages (for example, `pkg1` inside `pkg2`), the generated namespaces have the same nesting.

  The code generated for classes and enumerations that are inside packages now uses shorter names for the classes and enumerations, which are identical to their names in the MATLAB code.

When creating packages for your MATLAB code that are intended for code generation, follow these guidelines:

- Do not create a package that has the name `'coder'`.

- If you set the CppNamespaceForMathworksCode property (or the equivalent parameter in the app) to a nondefault name, do not create a package that has that name.

Namespaces help organize your code into logical parts, prevent name collisions, and enable you to more easily integrate your generated C++ code into a larger C++ project. Namespaces also increase compliance with the MISRA C++ standards for safety-critical code.

See Organize Generated C++ Code into Namespaces.

Compatibility Considerations

In previous releases, the generated C++ code was not partitioned into these namespaces. So, in R2020b, the code generated for MATLAB entry-point functions that have input or output arguments that are classes or enumerations in packages has a different interface compared to previous releases. To generate code similar to code generated in previous releases:

- To disable creating a separate namespace for MathWorks code, do one of the following:
  - In a code configuration object, set the parameter CppNamespaceForMathworksCode to the empty character vector `''`.
  - In the MATLAB Coder app, clear the **Namespace for MathWorks Code** parameter.
- To disable converting MATLAB packages to C++ namespaces, do one of the following:
  - In a code configuration object, set the parameter CppPackagesToNamespaces to `false`.
  - In the MATLAB Coder app, in the **Generate Code** step, on the **Code Appearance** tab, clear the **MATLAB package to C++ namespace** check box.

Improved identifier names in generated C++ code

In R2020b, the generated C++ code has these naming improvements:

- Contains overloaded functions or methods that have the same name but support multiple signatures. In previous releases, if an overloaded function `fcn` in your MATLAB code supported two different signatures, the code generator produced two different C++ functions, such as `b_fcn` and `c_fcn`.
- Reuses the same identifier name across different namespace hierarchies. For example, the same type name `myType` can appear in two different namespace hierarchies with top-level namespaces `myNamespace_1` and `myNamespace_2`.

  In previous releases, the code generator did not reuse an identifier name. For example, if the type name `myType` appeared in a namespace hierarchy that started with `myNamespace_1`, the same name did not appear in a different hierarchy that started with `myNamespace_2`. Instead, the type name was renamed to something like `b_myType`.
- Certain classes in the generated code have shorter names compared to previous releases:
  - Code generated for the sparse data type contains the class `sparse`. In previous releases, this class was named `coder_internal_sparse`.
  - Code generated for an anonymous function contains the class `anonymous_function`. In previous releases, this class was named `coder_internal_anonymous_func`.
  - Code generated for a nested function contains the class `nested_function`. In previous releases, this class was named `coder_internal_nested_function`.

For more information on C++ language features in the generated code, see C++ Code Generation.

Compatibility Considerations

In previous releases, certain functions, classes, types, and so on were named differently compared to R2020b. The exact differences are described in this release note.

Improved file partitioning for generated C++ code

In R2020b, type definition of a generated C++ class is placed in a header file with the same MATLAB name as the class. The class implementations are placed in a separate `cpp` file. To integrate the generated C++ classes with your custom code, the header files of the classes need to be included in your code.

Compatibility Considerations

In previous releases, to integrate the generated code in your custom code, you had to include the header file that contained the declarations of the generated functions that your custom code called. In this release, for some rare cases, you might need to also include the header files that contain the type definitions that the generated code uses.

Clearer pattern of ordering of local variable declarations

In R2020a, the order of local variable declarations in the generated C/C++ code did not follow an obvious pattern. In R2020b, the grouping of local variable declarations in the generated C/C++ code is by type and in order of decreasing array size. For variables and array of the same type and size, the declarations are in alphabetical order.

For example, this table shows a sample of local variable declaration groupings in R2020a and R2020b. The R2020b declarations follow a clear pattern making it easier to locate a variable declaration.

Sample Local Variable Declarations in R2020a	Sample Local Variable Declarations in R2020b

`static boolean_T imgEdge[307200];`
`static uint8_T b_img2[307200];`
`static uint8_T label[307200];`
`static real32_T imgRect[230400];`
`static real32_T b_img[307200];`
`static real32_T img2[307200];;`
`int32_T b_r;`
`int32_T c;`
`int32_T i;`
`uint32_T q;`
`real32_T thresh;`
`real_T ex;`
`real_T pos;`
`boolean_T b_boundingbox_data[60];`
`int32_T c_boundingbox_data[60];`
`int32_T boundingbox_data[240];`
`uint8_T v_data[1];`
`uint8_T label_data[58564];`
`uint32_T points1[8];`
`real32_T tmp_data[8];`
`real32_T tform_T[9];`
`real32_T b_x[20];`
`real32_T e_BW[400];`
`real32_T f_BW[400];`
`real_T b_B[2];`
`real_T cor[9];`
`visioncodegen_BlobAnalysis_1 *obj;`
	`static real32_T b_img[307200];`
	`static real32_T img2[307200];`
	`static real32_T imgRect[230400];`
	`static uint8_T b_img2[307200];`
	`static uint8_T label[307200];`
	`static boolean_T imgEdge[307200];`
	`visioncodegen_BlobAnalysis_1 *obj;`
	`real_T cor[9];`
	`real_T b_B[2];`
	`real_T ex;`
	`real_T pos;`
	`int32_T boundingbox_data[240];`
	`int32_T c_boundingbox_data[60];`
	`int32_T b_r;`
	`int32_T c;`
	`int32_T i;`
	`real32_T e_BW[400];`
	`real32_T f_BW[400];`
	`real32_T b_x[20];`
	`real32_T tform_T[9];`
	`real32_T tmp_data[8];`
	`real32_T thresh;`
	`uint32_T points1[8];`
	`uint32_T q;`
	`uint8_T label_data[58564];`
	`uint8_T v_data[1];`
	`boolean_T b_boundingbox_data[60];`

Code Generation Workflow

Reserve C/C++ identifier names by using coder.reservedName

In R2020b, you can generate code that does not use certain identifier names. You specify these names by using the `coder.reservedName` function. Use this function in your MATLAB code to reserve these identifier names for exclusive use in external C/C++ code that you want to integrate with the generated code. If you get errors due to name collisions between the generated code and external C/C++ code, use this functionality to resolve such errors.

To reserve identifier names `name1`, `name1`, and `name3`, include this function call in your MATLAB code:

```
coder.reservedName('name1', 'name2', 'name3')
```

This code generation setting, that has existed in the previous releases, provides the same functionality:

- In a code configuration object, the `ReservedNameArray` property
- Alternatively, in the MATLAB Coder app, on the **Code Appearance** tab, the **Reserved names** parameter

Access license checkout information by using report information object

In R2020b, the `Summary` property of the report information object has an additional property `ToolboxLicenses`. This property is a string vector that indicates which toolbox licenses were checked out during code generation. If you generate MEX code, these licenses are checked out again when you load the MEX function.

For example, define the function `foo` that calls the `iqr` function.

```
function r = foo(x) %#codegen
r = iqr(x);
end
```

Generate MEX code by using the `codegen` command. Also, create a report information object `info`.

```
codegen foo -args {zeros(1,100)} -reportinfo info
```

Inspect the `Summary` property of the object `info`.

```
info.Summary
```

```
ans = 

  Summary with properties:

            Success: true
               Date: '24-Apr-2020 04:36:51'
         OutputFile: 'C:\coder\R2020b\License discovery\foo_mex.mexw64'
          Processor: 'Generic->MATLAB Host Computer'
            Version: 'MATLAB Coder 5.0 (R2020b)'
    ToolboxLicenses: "statistics_toolbox"
```

The string vector `info.Summary.ToolboxLicenses` indicates that the Statistics and Machine Learning Toolbox™ license was checked out during code generation. This license is checked out again when you load the generated MEX function.

See Access Code Generation Report Information Programmatically and `coder.ReportInfo Properties`.

Directly package generated standalone code by using codegen command

In R2020b, you can use the `codegen` command with the `-pacakge` option to generate standalone code and package the generated code files and their dependencies into a compressed ZIP file in a single step. You can then use the ZIP file to relocate, unpack, and rebuild the code files in another development environment.

For example, to generate a static C library for the MATLAB function `foo` and package the library code and its dependencies into a ZIP file named `foo.zip`, run this command:

```
codegen -config:lib foo -package foo.zip
```

This packaging functionality is also provided by the `packNGo` function.

Query capability for target.get function

Use the `target.get` function to obtain a list of target feature objects that are saved in memory. You can refine the query to list only objects with properties that match specified name-value pairs.

Previously, you used this function only to retrieve a specified target feature object from memory.

Intel C and C++ toolchain support for Windows

You can compile generated code by using Intel C and C++ compilers for Windows. R2020b supports:

- Intel Parallel Studio XE 2020 with Microsoft Visual Studio 2017, 2019
- Intel Parallel Studio XE 2019 with Microsoft Visual Studio 2015, 2017, 2019
- Intel Parallel Studio XE 2018 with Microsoft Visual Studio 2015, 2017, 2019

Support for Intel Parallel Studio XE 2017 is removed.

For more information, see Supported Compilers.

Performance

Global Settings for Function Inlining: Fine-tune readability and speed of generated code

Inlining is a technique that replaces a function call with the contents (body) of that function. Inlining is a code optimization technique that eliminates the overhead of a function call, thereby improving speed. But inlining can produce larger C/C++ code and reduce code readability.

In R2020b, these new inlining settings give you greater control over speed and readability of the generated MEX and standalone C/C++ code.

Code Configuration Parameter	Description	Options
In a code configuration object: `InlineBetweenUserFunctions` In the MATLAB Coder app: On the **All Settings** tab, **Inline between user functions**	Controls inlining behavior at all call sites where a function that you wrote calls another function that you wrote	`'Always'` | `'Speed'` (default) | `'Readability'` | `'Never'`
In a code configuration object: `InlineBetweenMathWorksFunctions` In the MATLAB Coder app: On the **All Settings** tab, **Inline between MathWorks functions**	Controls inlining behavior at all call sites where a MathWorks function calls another MathWorks function	`'Always'` | `'Speed'` (default) | `'Readability'` | `'Never'`
In a code configuration object: `InlineBetweenUserAndMathWorksFunctions` In the MATLAB Coder app: On the **All Settings** tab, **Inline between user and MathWorks functions**	Controls inlining behavior at all call sites where a function that you wrote calls a MathWorks function, or a MathWorks function calls a function that you wrote	`'Always'` | `'Speed'` (default) | `'Readability'` | `'Never'`

Option descriptions:

- `'Always'`: Always performs inlining at a call site.
- `'Speed'`: Uses internal heuristics to determine whether to perform inlining at a call site. This setting usually leads to highly optimized code. This setting is the default setting.
- `'Readability'`: Almost never inlines function calls, except for calls to very small functions. Preserves modularity of code without sacrificing too much speed, whenever possible. Results in highly readable code.
- `'Never'`: Never inlines function calls. Results in maximum readability. This setting might significantly reduce the performance of the generated code.

You might have different speed and readability requirements for the code generated for functions that you write and MathWorks functions. These new settings enable you to separately control the inlining behavior for these two parts of the generated code base and at the boundary between them. For example, you might want to simultaneously:

- Preserve the modularity in the code that you write for better readability, even if that reduces the speed of the generated code. For this behavior, set `InlineBetweenUserFunctions` to `'Readability'`.
- Generate highly optimized code for MathWorks functions, even if that results in less readable code because you are less likely to inspect this part of your code base. For this behavior, set `InlineBetweenMathWorksFunctions` to `'Speed'`.
- In the generated code, separate functions that you write and MathWorks functions so that the generated code does not look very different from your MATLAB code. For this behavior, set `InlineBetweenUserAndMathWorksFunctions` to `'Readability'`.

Interaction with other existing inlining controls:

- The `-O disable:inline` option of the `codegen` command disables inlining between all functions. This option is equivalent to setting all the three new parameters to `'Never'`. If there is a conflict, this option overrides the individual values of the three new parameters.
- The `coder.inline` directive placed inside the body of a MATLAB function overrides the effect of the `InlineBetweenUserFunctions`, `InlineBetweenMathWorksFunctions`, and `InlineBetweenUserAndMathWorksFunctions` parameters.

See Control Inlining to Fine-Tune Performance and Readability of Generated Code.

Compatibility Considerations

- Choosing nondefault values for the existing code configuration parameters `InlineThreshold`, `InlineThresholdMax`, and `InlineStackLimit` now produces a warning during code generation. For a call site, if there is a conflict between these nondefault values and the values of the new parameters, the behavior of the code generator is determined by these nondefault values. These three existing parameters will be removed in a future release.
- If you set the `InlineBetweenUserFunctions`, `InlineBetweenMathWorksFunctions`, and `InlineBetweenUserAndMathWorksFunctions` parameters to their default values (`'Speed'`), the code generator behaves the same as in previous releases. The exception to this rule is when you use the `-O disable:inline` option with the `codegen` command. In previous releases, this option disabled inlining only for calls between functions that you wrote and did not affect calls to or from MathWorks functions. In R2020b, this option disables inlining for all function calls.

JIT support for half-precision floating-point data type

In R2020b, you can use Just-In-Time (JIT) compilation for MEX code generation with half-precision floating-point data types in MATLAB.

Deep Learning with MATLAB Coder

Deep Learning: Generate code for Long Short-Term Memory (LSTM) layer

In R2020b, you can generate C++ code for an LSTM network, a stateful LSTM network, or a bidirectional LSTM network that uses the Intel MKL-DNN library.

You can also generate C++ code for an LSTM network that has its the activation properties (`GetActivationFunction` and `SetActivationFunction`) set to nondefault values in which the generated code uses the ARM Compute library.

An LSTM layer learns long-term dependencies between time steps in time series and sequence data. This layer performs additive interactions, which can help improve gradient flow over long sequences during training. See `lstmLayer`, `bilstmLayer`, and Networks and Layers Supported for C++ Code Generation.

Deep Learning: Generate code for custom layers

In R2020b, you can generate C++ code for custom deep learning layers that uses the Intel MKL-DNN or the ARM Compute library.

See:

- Define Custom Deep Learning Layers (Deep Learning Toolbox)
- Define Custom Deep Learning Layer for Code Generation (Deep Learning Toolbox)
- Networks and Layers Supported for C++ Code Generation

Deep Learning: Generate code that uses Intel MKL-DNN library on macOS platform

In R2020b, you can generate C++ code for deep learning layers and networks that uses the Intel MKL-DNN library on the macOS platform. See Prerequisites for Deep Learning with MATLAB Coder.

Deep Learning: Generate code for additional layers

In R2020b, C++ code generation that uses the Intel MKL-DNN library supports these additional layers:

- Dice pixel classification layer that provides a categorical label for each image pixel or voxel using generalized Dice loss (`dicePixelClassificationLayer`)
- Focal loss layer that predicts object classes using focal loss (`focalLossLayer`)
- Gated recurrent unit (GRU) layer (`gruLayer`)
- Multiplication layer (`multiplicationLayer`)
- Box regression layer for Fast and Faster R-CNN (`rcnnBoxRegressionLayer`)
- Classification layer for region proposal networks (RPNs) (`rpnClassificationLayer`)
- Scaling layer for actor or critic network (`scalingLayer`)

- Sequence input layer (`sequenceInputLayer`)
- Softplus layer for actor or critic network (`softplusLayer`)
- Space to depth layer (`spaceToDepthLayer`)
- Word embedding layer that maps word indices to vectors (`wordEmbeddingLayer`)

In R2020b, C++ code generation with the ARM Compute library supports this additional layer:

- Dice pixel classification layer that provides a categorical label for each image pixel or voxel using generalized Dice loss (`dicePixelClassificationLayer`)
- Focal loss layer that predicts object classes using focal loss (`focalLossLayer`)
- Gated recurrent unit (GRU) layer (`gruLayer`)
- Multiplication layer (`multiplicationLayer`)
- Box regression layer for Fast and Faster R-CNN (`rcnnBoxRegressionLayer`)
- Classification layer for region proposal networks (RPNs) (`rpnClassificationLayer`)
- Scaling layer for actor or critic network (`scalingLayer`)
- Softplus layer for actor or critic network (`softplusLayer`)
- Space to depth layer (`spaceToDepthLayer`)

Functionality being removed or changed

cnncodegen Function: Support for CPU targets to be removed

Behavior change in future release

In a future release, the `cnncodegen` function will not generate C++ code for Intel and ARM CPU targets. To generate C++ code for deep learning layers and networks for these targets, use the `codegen` function.

See Code Generation for Deep Learning Networks with MKL-DNN and Code Generation for Deep Learning Networks with ARM Compute Library.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2020a

MATLAB Programming for Code Generation

Generate code for half-precision floating-point data type

In R2020a, you can generate C/C++ code for half-precision floating-point data types in MATLAB. Half-precision data types occupy only 16 bits of memory, but their floating-point representation enables them to handle wider dynamic ranges than integer or fixed-point data types of the same size.

For a full list of features that support half-precision code generation, see `half`.

Code generation for datetime arrays

In R2020a, you can generate C/C++ code for `datetime` arrays. For more information, see Code Generation for Datetime Arrays.

Code generation for timetables

In R2020a, you can generate C/C++ code for timetables that have `duration` vectors as row times. For more information, see Code Generation for Timetables.

Supported Functions

Code generation for more MATLAB functions

- `array2timetable`
- `convhull`
- `datetime`
- `fscanf`
- `ischange`
- `isnat`
- `isregular`
- `NaT`
- `onCleanup`
- `smoothdata`
- `table2timetable`
- `timetable`
- `timetable2table`
- `topkrows`

Code generation for more toolbox functions

5G Toolbox

- `nrCORESETConfig`
- `nrPDCCHConfig`
- `nrPDCCHResources`
- `nrPDCCHSpace`
- `nrPDSCHConfig`
- `nrPDSCHDMRS`
- `nrPDSCHDMRSConfig`
- `nrPDSCHDMRSIndices`
- `nrPDSCHIndices`
- `nrPDSCHPTRS`
- `nrPDSCHPTRSConfig`
- `nrPDSCHPTRSIndices`
- `nrPDSCHReservedConfig`
- `nrPRACH`
- `nrPRACHConfig`
- `nrPRACHGrid`
- `nrPRACHIndices`

- `nrPUSCHConfig`
- `nrPUSCHPTRSIndices`
- `nrPUSCHPTRSConfig`
- `nrPUSCHPTRS`
- `nrPUSCHIndices`
- `nrPUSCHDMRSIndices`
- `nrPUSCHDMRSConfig`
- `nrPUSCHDMRS`
- `nrSearchSpaceConfig`
- `nrSRS`
- `nrSRSConfig`
- `nrSRSIndices`

Audio Toolbox

- `acousticLoudness`
- `acousticSharpness`
- `calibrateMicrophone`
- `detectSpeech`
- `phon2sone`
- `sone2phon`

Automated Driving Toolbox

- `cubicLaneBoundary`
- `findCubicLaneBoundaries`
- `findParabolicLaneBoundaries`
- `insertLaneBoundary`
- `parabolicLaneBoundary`

Communications Toolbox

- `bluetoothIdealReceiver`
- `bluetoothPhyConfig`
- `bluetoothWaveformConfig`
- `bluetoothWaveformGenerator`
- `channelDelay`
- `cranerainpl`

Computer Vision Toolbox

See “Code Generation: Generate C/C++ code using MATLAB Coder” (Computer Vision Toolbox).

DSP System Toolbox

- `dsp.SOSFilter`

Fixed-Point Designer

- `nextpow2`
- `normalizedReciprocal`

Image Processing Toolbox

- `imbilatfilt`

Instrument Control Toolbox

- `maskWrite`
- `modbus`
- `read` from a MODBUS server
- `write` to a MODBUS server
- `writeread`

Model Predictive Control Toolbox

- `mpcActiveSetOptions`
- `mpcActiveSetSolver`
- `mpcInteriorPointOptions`
- `mpcInteriorPointSolver`
- `mpcmoveCodeGeneration`
- `nlmpcmoveCodeGeneration`

Navigation Toolbox

- `copy` of `plannerRRT` and `plannerRRTStar`
- `getParticles`
- `monteCarloLocalization`
- `plan` of `plannerHybridAStar`
- `plan` of `plannerRRT` and `plannerRRTStar`
- `plannerHybridAStar`
- `plannerRRT`
- `plannerRRTStar`
- `validatorVehicleCostmap`

Optimization Toolbox

See “Code Generation for Quadratic Problems: Generate C code for problems with linear constraints and quadratic objectives” (Optimization Toolbox).

Phased Array System Toolbox

- blkdiagbfweights
- cranerainpl

Robotics System Toolbox

- checkCollision
- collisionBox
- collisionCylinder
- collisionMesh
- collisionSphere

Sensor Fusion and Tracking Toolbox

- angvel
- deleteBranch, initializeBranch of trackerTOMHT
- deleteTrack, initializeTrack of trackerPHD
- residual, residualmag, residualaltimeter of ahrs10filter
- residual, residualaccel, residualgps, residualmag, residualgyro of insfilterAsync
- residual, residualgps, residualmvo of insfilterErrorState
- trackGOSPAMetric

SerDes Toolbox

- impulse2pulse
- optPulseMetric
- prbs
- pulse2impulse
- pulse2wave
- wave2pulse

Signal Processing Toolbox

See “C/C++ Code Generation Support: Generate code for time-frequency analysis, feature extraction, spectral analysis, multirate signal processing, and filter design” (Signal Processing Toolbox).

Wavelet Toolbox

- dualtree
- dualtree2
- dwpt
- haart
- haart2
- hht
- idualtree

- idualtree2
- idwpt
- ihaart
- ihaart2
- modwtvar
- qbiorthfilt
- qorthwavf
- wdenoise

WLAN Toolbox

- getTRSConfiguration
- wlanHETBConfig

Generated Code Improvements

Generate C++ classes from MATLAB classes

In R2020a, when you generate C++ libraries or executables, the default behavior of the code generator is to produce C++ classes for the classes in your MATLAB code. These include all MATLAB classes such as value classes, handle classes, and system objects. In previous releases, if you set the target language for standalone code generation as C++, the code generator produced structures for the classes in your MATLAB code.

See Generate C++ Classes for MATLAB Classes.

Compatibility Considerations

While attempting to generate standalone code that contains C++ classes for MATLAB classes, you might get a warning message if both of these conditions are true:

- You choose to generate reentrant code by enabling the `MultiInstanceCode` parameter in a code configuration object, or by enabling the **Generate re-entrant code** parameter in the MATLAB Coder app.
- The destructor of a class in your MATLAB code has a persistent variable or calls another function that declares and uses a persistent variable.

In such situations, to generate code that contains C++ classes for MATLAB classes, disable the `MultiInstanceCode` or the **Generate re-entrant code** parameter.

Alternatively, you can change the default behavior of the code generator to produce C++ code that contains structures for MATLAB classes. Do one of the following:

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set `TargetLang` to `'C++'` and `CppPreserveClasses` to `false`.
- In the MATLAB Coder app, in the **Generate** step, set **Language** to **C++**. In the project build settings, on the **Code Appearance** tab, clear the **Generate C++ classes from MATLAB classes** check box.

Use dynamically allocated C++ arrays in generated function interfaces

In most cases, when you generate code for a MATLAB function that accepts or returns an array, there is an array at the interface of the generated C/C++ function. For an array size that is unknown at compile time, or whose bound exceeds a predefined threshold, the memory for the generated array is dynamically allocated on the heap. In R2020a, if you choose C++ as the target language for code generation, the generated code implements such dynamically allocated arrays as a C++ class template named `coder::array`.

The `coder::array` template is defined in a header file named `coder_array.h` in the build folder. To use dynamically allocated arrays in your custom C++ code (for example, a custom main function) that you want to integrate with the generated code, include the `coder_array.h` header file in your custom `.cpp` files.

To learn how to use the `coder::array` template, see Use Dynamically Allocated C++ Arrays in the Generated Function Interfaces.

Compatibility Considerations

In previous releases, the generated C++ code implemented dynamically allocated arrays by using the C style `emxArray` data structure. In your custom C++ code that you integrated with the generated code, you used the API for creating and allocating `emxArray` data structures. In R2020a, to change the default behavior of the code generator and produce `emxArray` data structures in the generated C ++ code, do one of the following:

- In a code configuration object (`coder.MexCodeConfig`, `coder.CodeConfig`, or `coder.EmbeddedCodeConfig`), set the `DynamicMemoryAllocationInterface` parameter to `'C'`.
- In the MATLAB Coder app, on the **Memory** tab, set **Dynamic memory allocation interface** to `Use C style EmxArray`.

Generate code that uses the C++11 standard math library

In R2020a, you can generate C++ libraries and executables that use the C++11 (ISO) or ISO®/IEC 14882:2011(E) standard math library by doing one of the following:

- Use the `codegen` command with the `-lang:c++` and `-std:c++11` options.
- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set the `Targetlang` parameter to `'C++'` and the `TargetlangStandard` parameter to `'C++11 (ISO)'`.
- In the MATLAB Coder app, in the **Generate** step, set **Language** to **C++**. In the project build settings, on the **Custom Code** tab, set the **Standard math library parameter** to `C++11 (ISO)`.

See Change the Standard Math Library.

Manage memory for threadprivate variables in generated C++ code

In the Open Multiprocessing (Open MP) application interface, each thread has its own copy of a variable that is declared as threadprivate. The memory for a threadprivate variable is allocated on the heap and must be freed by calling its destructor at the end of the thread execution. When you generate code for `parfor` loops that uses the Open MP interface, the generated code can contain threadprivate variables if either of these conditions is true:

- Your MATLAB code contains `persistent` variables.
- You set the maximum stack usage parameter to a low value.

In such cases, if you generate standalone C++ code, the generated initialize and terminate functions contain code that manages memory for the threadprivate variables. For each copy of a threadprivate variable, the initialize function uses the `new` operator to allocate memory. If the variable has a default value, the function also initializes the value of each copy to the default value. The terminate function uses the `delete` operator to free the memory for each copy at the end of the execution.

Generated C++ MEX code also uses the same constructs to manage memory for threadprivate variables.

The generated code allocates memory to all threads that are available for the Open MP application in your system, even if the generated code uses only a subset of these threads. This behavior can potentially incur memory overhead and reduce the performance of generated code that contains threadprivate variables.

The generated standalone code automatically includes a call to the initialize function at the beginning of the entry-point functions. To make sure that the memory allocated to threadprivate variables is properly freed, call the terminate function after you call the generated entry-point functions for the last time. See Use Generated Initialize and Terminate Functions.

Generate C++ code that complies with MISRA C++:2008 Rule 3-4-1

In R2020a, generated C++ code contains variable declarations that have minimized block scope. These variable declarations increase the likelihood of generating C++ code that is compliant with the Rule 3-4-1 of the MISRA C++:2008 guidelines.

This table shows the declaration of variables in the generated C++ code in R2019b and R2020a. In R2020a generated code, the variable declaration has minimum block scope. The minimum block scope reduces the visibility of these variables and improves compliance with MISRA C++.

MATLAB Code	R2019b Generated Code	R2020a Generated Code
`function y1 = minimalScopingExample(a,b)`	`double minimalScopingExample(double a, double b)`	`double minimalScopingExample(double a,`
`%#codegen`	`{`	`{`
	`  double b_y1;`	`  double b_y1;`
`for count = 1:10`	`  int count;`	`  for (int count = 0; count < 10; count`
`    if(count > 3)`	`  int u1;`	`    int u1;`
`        myVar = 1;`	`  double temp;`	`    if (count + 1 > 3) {`
`    elseif(count > 7)`	`  for (count = 0; count < 10; count++) {`	`      u1 = 1;`
`        temp = b;`	`    if (count + 1 > 3) {`	`    } else if (count + 1 > 7) {`
`        b = a;`	`      u1 = 1;`	`      double temp;`
`        a = temp;`	`    } else if (count + 1 > 7) {`	`      temp = b;`
`        myVar = -1;`	`      temp = b;`	`      b = a;`
`    else`	`      b = a;`	`      a = temp;`
`        myVar = 0;`	`      a = temp;`	`      u1 = -1;`
`    end`	`      u1 = -1;`	`    } else {`
`    y1= max(b,myVar);`	`    } else {`	`      u1 = 0;`
	`      u1 = 0;`	`    }`
`end`	`    }`	
`end`		`    b_y1 = u1;`
	`    b_y1 = u1;`	`    if (b > b_y1) {`
	`    if (b > b_y1) {`	`      b_y1 = b;`
	`      b_y1 = b;`	`    }`
	`    }`	`  }`
	`  }`	
		`  return b_y1;`
	`  return b_y1;`	`}`
	`}`	

Generate one MEX function that supports multiple signatures

In R2020a, you can generate one MEX function that supports multiple signatures for the same entry-point function in code generation. This one MEX function reduces the overhead involved in generating separate MEX functions for each signature of your entry-point function. The generated MEX function works with all the signatures provided during code generation. For more information, see Generate One MEX Function That Supports Multiple Signatures.

Code Generation Workflow

Coder Type Editor: Create and edit input types interactively

When generating C/C++ code at the command line, you can specify the type, size, and complexity of the input arguments of your MATLAB entry-point functions by using `coder.Type` objects. In R2020a, you can create and edit `coder.Type` objects interactively by using the Coder Type Editor. To launch the Coder Type Editor, run this command at the MATLAB command line:

```
coderTypeEditor
```

See Create and Edit Input Types by Using the Coder Type Editor and `coderTypeEditor`.

Intel C and C++ toolchain support for Windows

You can compile generated code by using Intel C and C++ compilers for Windows. R2020a supports:

- Intel Parallel Studio XE 2017 with Microsoft Visual Studio 2015, 2017
- Intel Parallel Studio XE 2018 with Microsoft Visual Studio 2015, 2017
- Intel Parallel Studio XE 2019 with Microsoft Visual Studio 2015, 2017, 2019

For more information, see Supported Compilers.

Performance

Default code generation setting optimizes build to minimize run time

In R2020a, the default compiler optimization setting for standalone code generation optimizes build to minimize the run time of the generated executable. In previous releases, the default compiler setting optimized build to minimize the build time.

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), the `BuildConfiguration` parameter has the default value `'Faster Runs'`.
- In the MATLAB Coder app, on the **Hardware** tab, the **Build Configuration** parameter has the default value `Faster Runs`.

Compatibility Considerations

- In R2020a, the build time for the generation of C/C++ libraries or executables from MATLAB code can be longer compared to previous releases. To shorten the build time, change the value of the build configuration parameter to `'Faster Builds'`.
- Compared to the `'Faster Builds'` setting, `'Faster Runs'` activates more compiler optimizations and can expose more bugs in your third-party C/C++ compiler. To eliminate build errors caused by such bugs, try to generate code again with the build configuration parameter set to `'Faster Builds'`.

Improved performance of code generated for fast Fourier transform (FFT) functions

Compared to previous releases, the generated code in R2020a can have improved performance in these cases:

- Code generated for `fft2`, `fftn`, `ifft2`, and `ifftn` that uses FFTW library calls
- Standalone code (that does not use FFTW library calls) generated for fast Fourier transform functions with real inputs of even length

Improved code quality for functions that allocate handle objects

In previous releases, if your MATLAB function created multiple instances of a handle class, the generated code contained a separate local buffer variable for each instance of the handle class. In R2020a, the generated code contains a single array of buffers.

In previous releases, if several instances of the handle class were passed to another function in the MATLAB code, the generated code also passed pointers to the corresponding buffers. In R2020a, the generated code passes a single pointer to the array of buffers.

As a result of these changes, if you allocate a large number of handle objects in a function and then pass them to another function in your MATLAB code, the generated code passes a single pointer instead of passing a pointer for each handle object. This code pattern increases the efficiency of the generated code.

MATLAB Code	R2019b Generated Code	R2020a Generated Code
`classdef A < handle`	`typedef struct {`	`typedef struct {`
`    properties`	`  double property;`	`  double property;`
`        property`	`} A;`	`} A;`
`    end`		
`end`	`double foo(double arg)`	`double foo(double arg)`
	`{`	`{`
`function result = foo(arg)`	`  A lobj_0;`	`  A lobj_0[2];`
`[x, y] = baz(arg);`	`  A lobj_1;`	`  A *x;`
`result = x.property -...`	`  A *x;`	`  A *y;`
`         y.property`	`  A *y;`	`  baz(arg, &lobj_0[0], &x,`
	`  baz(arg, &lobj_0, &lobj_1,`	`                        &y);`
`function [x, y] = baz(arg)`	`      &x, &y);`	`  return x->property -`
`coder.inline('never')`	`  return x->property -`	`               y->property;`
`a = A;`	`               y->property;`	`}`
`a.property = arg;`	`}`	
`b = A;`		`void baz(double arg,`
`b.property = arg * 2;`	`void baz(double arg,`	`A *iobj_0, A **x, A **y)`
`if arg > 0`	`A *iobj_0, A *iobj_1,`	`{`
`    x = a;`	`A **x, A **y)`	`  *y = &iobj_0[0];`
`    y = b;`	`{`	`  iobj_0[0].property = arg;`
`else`	`  iobj_1->property = arg;`	`  iobj_0[1].property = arg *`
`    x = b;`	`  iobj_0->property =`	`                         2.0;`
`    y = a;`	`               arg * 2.0;`	`  if (arg > 0.0) {`
`end`	`  if (arg > 0.0) {`	`    *x = *y;`
	`    *x = iobj_1;`	`    *y = &iobj_0[1];`
	`    *y = iobj_0;`	`  } else {`
	`  } else {`	`    *x = &iobj_0[1];`
	`    *x = iobj_0;`	`  }`
	`    *y = iobj_1;`	`}`
	`  }`	
	`}`	

In R2019b, the generated function `foo` declared two local buffer variables `lobj_0` and `lobj_1` for the two allocations of the handle class `A`. Also, pointers to these two variables were passed as additional parameters to `baz`. In R2020a, `foo` declares a single array of buffers `lobj_0[2]`, and a single pointer to this array is passed to `baz`.

Deep Learning with MATLAB Coder

Deep Learning: Generate code for Long Short-Term Memory (LSTM) layer

In R2020a, you can generate C++ code for an LSTM network, a stateful LSTM network, or a bidirectional LSTM network that uses the ARM Compute library. An LSTM layer learns long-term dependencies between time steps in time series and sequence data. This layer performs additive interactions, which can help improve gradient flow over long sequences during training.

See `lstmLayer`, `bilstmLayer`, and Networks and Layers Supported for C++ Code Generation.

Deep Learning: Generate code for more layers, networks, and classes

Additional Layers

In R2020a, C++ code generation with the ARM Compute library supports these additional layers:

- Anchor box layer for object detection (`anchorBoxLayer`)
- Bidirectional Long Short-Term Memory (BiLSTM) layer (`bilstmLayer`)
- Concatenation layer (`concatenationLayer`)
- Layer that applies 2-D cropping to the input (`crop2dLayer`)
- Exponential linear unit (ELU) layer (`eluLayer`)
- 2-D global max pooling layer (`globalMaxPooling2dLayer`)
- Long Short-Term Memory (LSTM) layer (`lstmLayer`)
- Layer that implements ONNX identity operator (`nnet.onnx.layer.IdentityLayer`)
- Sequence input layer (`sequenceInputLayer`)
- SSD merge layer for object detection (`ssdMergeLayer`)
- Word embedding layer that maps word indices to vectors (`wordEmbeddingLayer`)

In R2020a, C++ code generation with the Intel MKL-DNN library supports this additional layer:

- Anchor box layer for object detection (`anchorBoxLayer`)
- Concatenation layer (`concatenationLayer`)
- Exponential linear unit (ELU) layer (`eluLayer`)
- 2-D global max pooling layer (`globalMaxPooling2dLayer`)
- Layer that implements ONNX identity operator (`nnet.onnx.layer.IdentityLayer`)
- SSD merge layer for object detection (`ssdMergeLayer`)

Additional Networks

In R2020a, C++ code generation with the ARM Compute library or the Intel MKL-DNN library supports these additional networks:

- Pretrained DarkNet-19 and DarkNet-53 convolutional neural networks (`darknet19` and `darknet53`)

- Pretrained DenseNet-201 convolutional neural network (`densenet201`)
- Pretrained Inception-ResNet-v2 convolutional neural network (`inceptionresnetv2`)
- Pretrained NASNet-Large convolutional neural network (`nasnetlarge`)
- Pretrained NASNet-Mobile convolutional neural network (`nasnetmobile`)
- Pretrained ResNet-18 convolutional neural network (`resnet18`)
- Pretrained Xception convolutional neural network (`xception`)

Additional classes

In R2020a, C++ code generation with the ARM Compute library or the Intel MKL-DNN library supports this additional class:

- `ssdObjectDetector`

See Networks and Layers Supported for C++ Code Generation.

Deep Learning: Generate code that uses newer versions of ARM Compute and Intel MKL-DNN libraries

In R2020a, you can generate more efficient C++ code for layers and networks that use these newer versions of ARM Compute and Intel MKL-DNN libraries:

- ARM Compute library for computer vision and machine learning, version 19.05. See https://developer.arm.com/ip-products/processors/machine-learning/compute-library.
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) v1.0. See https://01.org/dnnl.

See Prerequisites for Deep Learning with MATLAB Coder.

Compatibility Considerations

In R2020a, generation of C++ code that uses these versions of ARM Compute and Intel MKL-DNN libraries is not supported:

- ARM Compute library for computer vision and machine learning, version 18.03
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) v0.14

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2019b

MATLAB Programming for Code Generation

Code generation for tables

In R2019b, you can generate C/C++ code for tables. For more information, see Code Generation for Tables.

Code generation for duration arrays

In R2019b, you can generate C/C++ code for duration arrays. For more information, see Code Generation for Duration Arrays.

Code generation for hexadecimal and binary literals

In R2019b, you can generate C/C++ code for MATLAB hexadecimal and binary literals. The literals appear as integers in the generated code. In certain situations, the code generator might perform optimizations that remove these literals from the generated code.

For more information, see Hexadecimal and Binary Values (MATLAB).

Supported Functions

Code generation for more MATLAB functions

- `array2table`
- `cell2table`
- `datevec`
- `days`
- `duration`
- `fgetl`
- `fgets`
- `fillmissing`
- `fileread`
- `height`
- `histcounts2`
- `hms`
- `hours`
- `islocalmax`
- `islocalmin`
- `ismissing`
- `matchpairs`
- `milliseconds`
- `minutes`
- `newline`
- `read`
- `rmmissing`
- `rmoutliers`
- `seconds`
- `standardizeMissing`
- `struct2table`
- `table`
- `table2array`
- `table2cell`
- `table2struct`
- `tcpclient`
- `width`
- `write`
- `years`

Code generation for more toolbox functions

5G Toolbox

- `nrCarrierConfig`
- `nrChannelEstimate`
- `nrCSIRS`
- `nrCSIRSConfig`
- `nrCSIRSIndices`
- `nrTimingEstimate`

Audio Toolbox

- `audioPluginGridLayout`
- `audioTimeScaler`
- `designAuditoryFilterBank`
- `pinknoise`
- `shiftPitch`
- `stretchAudio`

Automated Driving Toolbox

- `acfObjectDetectorMonoCamera`
- `birdsEyeView`
- `imageToVehicle` of `birdsEyeView`
- `segmentLaneMarkerRidge`
- `transformImage` of `birdsEyeView`
- `vehicleToImage` of `birdsEyeView`

Communications Toolbox

- `bleATTPDU`
- `bleATTPDUConfig`
- `bleATTPDUDecode`
- `bleChannelSelection`
- `bleGAPDataBlock`
- `bleGAPDataBlockConfig`
- `bleGAPDataBlockDecode`
- `bleIdealReceiver`
- `bleL2CAPFrame`
- `bleL2CAPFrameConfig`
- `bleL2CAPFrameDecode`
- `bleLLAdvertisingChannelPDU`
- `bleLLAdvertisingChannelPDUConfig`

- `bleLLAdvertisingChannelPDUDecode`
- `bleLLControlPDUConfig`
- `bleLLDataChannelPDU`
- `bleLLDataChannelPDUConfig`
- `bleLLDataChannelPDUDecode`
- `bleWaveformGenerator`
- `gsmCheckTimeMask`
- `gsmDownlinkConfig`
- `gsmFrame`
- `gsmInfo`
- `gsmUplinkConfig`

Computer Vision Toolbox

See “Code Generation: Generate C/C++ code using MATLAB Coder” (Computer Vision Toolbox).

Image Processing Toolbox

- `imregcorr`

Navigation Toolbox

- `cart2frenet`
- `frenet2cart`
- `nav.StateSpace`
- `nav.StateValidator`
- `plan`
- `stateSpaceDubins`
- `stateSpaceReedsShepp`
- `stateSpaceSE2`
- `trajectoryOptimalFrenet`
- `validatorOccupancyMap`

Optimization Toolbox

See `fmincon` Code Generation: Generate C code for nonlinear constrained optimization (requires MATLAB Coder) (Optimization Toolbox).

Phased Array System Toolbox

- `backscatterBicyclist`
- `clusterDBSCAN`
- `discoverClusters`
- `estimateEpsilon`
- `getNumScatterers`

- move of backscatterBicyclist
- ompdecomp
- omphybweights
- reflect of backscatterBicyclist

Robotics System Toolbox

- ackermannKinematics
- bicycleKinematics
- differentialDriveKinematics
- jointSpaceMotionModel
- taskSpaceMotionModel
- unicycleKinematics

Sensor Fusion and Tracking Toolbox

- complementaryFilter
- ctrect
- ctrectcorners
- ctrectjac
- ctrectmeas
- ctrectmeasjac
- fuserSourceConfiguration
- gmphd
- initcagmphd
- initctgmphd
- initctrectgmphd
- initcvgmphd
- objectTrack
- toStruct
- trackFuser
- trackOSPAMetric

Signal Processing Toolbox

These Signal Processing Toolbox™ functions now support C/C++ code generation:

- **Time-Frequency Analysis** — fsst, ifsst, tfridge, wvd, and xwvd
- **Spectral Analysis of Nonuniformly Sampled Signals** — plomb
- **Transforms** — dftmtx and rceps
- **Digital Filtering** — eqtflength, fftfilt, and tf2ss
- **Waveform Generation** — chirp, diric, gmonopuls, and sawtooth
- **Spectral Windows** — chebwin

Statistics and Machine Learning Toolbox

- See Generate C/C++ code for probability distribution functions (requires MATLAB Coder) (Statistics and Machine Learning Toolbox)
- See “saveLearnerForCoder and loadLearnerForCoder Functions: Save and load machine learning models for code generation” (Statistics and Machine Learning Toolbox)

Wavelet Toolbox

- `filterbank` of `shearletSystem`
- `framebounds` of `shearletSystem`
- `isheart2`
- `mdwtdec`
- `mdwtrec`
- `meyeraux`
- `numshears`
- `shearletSystem`
- `sheart2`

WLAN Toolbox

- `wlanAPEPLength`
- `wlanHESIGBCommonBitRecover`
- `wlanHESIGBUserBitRecover`
- `wlanPSDULength`

Generated Code Improvements

Generate C++ code that has more C++ language and object-oriented features

In R2019b, you can generate C++ code with more C++ language features. Your generated C++ code can appear more aligned with what is expected from the C++ language. You can more easily integrate your generated C++ code into existing C++ projects. You can now generate C++ code:

- In a namespace. Namespaces enable you to more easily integrate your generated C++ code into a larger C++ project. Namespaces also increase compliance with the MISRA C++ standards for safety-critical code. See C++ Code Generation.
- With an object-oriented interface. You call the generated entry-point functions as class methods. The methods for each class instance are thread-safe and reentrant. The generated class constructor and destructor automatically perform initialization and termination, for example, when memory must be allocated and freed. See Generate C++ Code with Class Interface.
- That uses more C++-specific libraries, such as `cstddef` and `cstdlib`. This usage improves MISRA C++ compliance.

Choose the style of generated #include guards

To prevent compilation errors due to multiple inclusion, the code generator produces `#include` guards in the generated header files. In R2019b, you can choose to generate the `#include` guards as `#pragma once` constructs. In previous releases, the code generator produced only `#ifndef` constructs. If distinct header files in your code project use the same preprocessor macros, then generate code with `#pragma once`.

To change the header guard style:

- From the command line, set the `HeaderGuardStyle` property of your code generation configuration object. For example:

  ```
  cfg = coder.config('lib');
  cfg.HeaderGuardStyle = 'UsePragmaOnce';
  ```
- From the app, at the **Generate Code** step, in **More Settings**, **Code Appearance**, set the **Header guard style** to `Use pragma once`.

C strings for null-terminated MATLAB strings

In R2019b, to improve the readability of the generated C/C++ code, the code generator generates C strings for null-terminated MATLAB strings and character row vectors instead of character arrays.

An example of generated code in R2019a and R2019b is in this table.

MATLAB Code	R2019a Generated Code	R2019b Generated Code

`function t = charArrayNullAtEnd()`
`    t = ['Hello', 0];`
`end`
	`void charArrayNullAtEnd(char t[6])`
	`{`
	`    int i0;`
	`    static const char cv0[6] = { 'H', 'e', 'l', 'l', 'o', '\x00' };`
	`    for (i0 = 0; i0 < 6; i0++) {`
	`    t[i0] = cv0[i0];`
	`    }`
	`}`
		`void charArrayNullAtEnd(char t[6])`
		`{`
		`    int i;`
		`    static const char cv[6] = "Hello";`
		`    for (i = 0; i < 6; i++) {`
		`      t[i] = cv[i];`
		`    }`
		`}`

For more information, see Generate C/C++ Strings from MATLAB Strings and Character Row Vectors.

Improved naming for generated temporary variables

In R2019b, you get improved naming for generated temporary variables that increases the readability of the generated C/C++ code. The naming is independent across different subfunctions. As a result, the generated C/C++ code varies less when there is a small change in the corresponding MATLAB code. For example, adding a new subfunction in the MATLAB code does not change the variable names in the generated C/C++ code that correspond to the existing subfunctions.

The table shows the generated code in R2019a and R2019b. In R2019b generated code, the temporary variable name starts with `i` instead of `i0` and is independent across subfunctions.

MATLAB Code	R2019a Generated Code	R2019b Generated Code

`function m = foo(q,n)`
`    m = q(1:n)*q(n:-1:1)+ sub1(q,n);`
`end`
 	
`function y = sub1(q,n)`
`    coder.inline('never');`
`    y = q(1:n);`
`end`
	`static void sub1(const emxArray_real_T*q, double n, emxArray_real_T*y)`
	`{`
	`    int loop_ub;`
	`    int i4;`
	`    /* Temporary variable*/`
	`...`
	
	`void foo(const emxArray_real_T*q, double n, double m_data[], int m_size[2])`
	`{`
	`    int i0; /* Temporary variable*/`
	`    int i1;`
	`    int i2;`
	`...`
		`static void sub1(const emxArray_real_T*q, double n, emxArray_real_T*y)`
		`{`
		`    int loop_ub;`
		`    int i;`
		`    /* Temporary variable*/`
		`...`
		
		`void foo(const emxArray_real_T*q, double n, double m_data[], int m_size[2])`
		`{`
		`    int i; /* Temporary variable*/`
		`    int i1;`
		`    int i2;`
		`...`

Code Generation Workflow

MATLAB Coder features in MATLAB Online

In R2019b, you can use most of the MATLAB Coder features through your web browser for teaching, learning, and convenient lightweight access. To access these features, sign in with your MathWorks account. For information about license requirements, visit the MATLAB Online product page.

Certain MATLAB Coder desktop features are not available in MATLAB Online. See Specifications and Limitations in the MATLAB Online product page.

Automatically call initialize function from entry-point functions in the generated code

While generating standalone code, the code generator produces an initialize function that initializes the data used by the entry-point functions. In past releases, you had to manually call the initialize function before invoking the entry-point functions. In R2019b, the code generator includes a call to the initialize function at the beginning of the entry-point functions. In the generated code, the initialize function is called automatically only once, even when there are multiple entry-point functions.

The code generator includes a call to the initialize function at the beginning of the generated entry-point functions by default. To generate code that does not automatically call the initialize function, do one of the following:

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set `RunInitializeFcn` to `false`.
- In the MATLAB Coder app, on the **All Settings** tab, set **Automatically run the initialize function** to `No`.

See Use Generated Initialize and Terminate Functions.

Compatibility Considerations

Compared to previous releases, if you generate code for an entry-point MATLAB function `foo`, the generated files `foo.c`, `foo_data.c`, `foo_initialize.c`, and `foo_terminate.c` now contain an additional boolean `isInitialized_foo`. The generated code uses this variable to make sure that the initialize function `foo_initialize` is called automatically exactly once.

To restore the old behavior of the code generator, disable the option to automatically call the initialize function in the generated code.

Access code insights and build logs programmatically by using the report information object

In R2019b, the `coder.ReportInfo` object contains two new properties:

- The `CodeInsights` property contains messages about potential issues in the generated code including potential differences in behavior from MATLAB code and potential row-major issues. These messages also appear in the code generation report **Code Insights** tab.

- The `BuildLogs` property contains the build logs produced during code generation. The build logs contain compilation and linking errors and warnings. These messages also appear in the code generation report **Build Logs** tab.

You can use the report information object to programmatically access this information. For example, you can display the code insights at the MATLAB command line. To perform this action, in your build script, access the `CodeInsights` property.

For more information, see Access Code Generation Report Information Programmatically and `coder.ReportInfo Properties`.

Convert codegen command to equivalent MATLAB Coder project

In R2019b, you can use `codegen` with the `-toproject` option to convert a `codegen` command to an equivalent MATLAB Coder project file. You can then generate code from the project file by using another `codegen` command or the MATLAB Coder app.

For example, to convert a `codegen` command with input arguments `input_arguments` to the project file `myProject.prj`, run:

```
codegen input_arguments -toproject myProject.prj
```

Input arguments to `codegen` include:

- Names of entry-point functions
- Input type definitions specified by using the `-args` option
- Code generation options, including parameters specified in configuration objects
- Names of custom source files to include in the generated code

You can also use the `-toproject` option to convert an incomplete `codegen` command to a project file. For example, to create a project file `myProjectTemplate.prj` that contains only the code generation parameters stored in the configuration object `cfg`, run:

```
codegen -config cfg -toproject myProjectTemplate.prj
```

`myProjectTemplate.prj` does not contain specifications of entry-point functions or input types. You cannot generate code from this project file. You can open `myProjectTemplate.prj` in the MATLAB Coder app and use it as a template to create full project files that you can use to generate code.

Running `codegen` with the `-toproject` option does not generate code. It creates only the project file.

See Convert `codegen` Command to Equivalent MATLAB Coder Project.

Create code configuration object from MATLAB Coder project

In R2019b, you can use the `-toconfig` option with the `coder` command to export the code configuration settings stored in a MATLAB Coder project file to a code configuration object. For example, executing the following command returns a code configuration object `cfg` corresponding to `myProject.prj`.

```
cfg = coder('-toconfig','myProject.prj')
```

This table specifies which code configuration object is returned for different project file settings.

Project File Settings in MATLAB Coder App	**Code Configuration Object Returned**
Build type is MEX.	`coder.MexCodeConfig`
Build type is static library, dynamically linked library, or executable. One of the following conditions is true: • You do not have Embedded Coder. • You have Embedded Coder. On the **All Settings** tab, **Use Embedded Coder features** is set to `No`.	`coder.CodeConfig`
Build type is static library, dynamically linked library, or executable. You have Embedded Coder. On the **All Settings** tab, **Use Embedded Coder features** is set to `Yes`.	`coder.EmbeddedCodeConfig`

See Share Build Configuration Settings.

Export of hardware device data

R2019b provides the `target.export` function, which enables you to share hardware device data across computers and users. For more information, see Export Hardware Device Data.

Data validation for hardware device features

R2019a introduced a new mechanism for registering hardware devices, which uses the target feature classes, `target.Processor` and `target.LanguageImplementation`. R2019b exposes `target.Object`, which is the base class for target feature classes. To validate the data integrity of objects that belong to target feature classes, use the `IsValid` property or `validate` method.

Consider an example where you create a `target.Processor` object and associate an existing language implementation with the object.

```
myProcessor = target.create('Processor');
myProcessor.LanguageImplementations = target.get('LanguageImplementation', ...
                                                 'ARM Compatible-ARM Cortex');
```

To validate the created object, run `myProcessor.IsValid` or `myProcessor.validate()`.

```
myProcessor.IsValid

ans =
  logical
  0

myProcessor.validate()

Error using target.Processor/validate
Target data validation failed.
* Undefined property "Name" in "Processor" object.
* Undefined identifier in "Processor" object.
```

The validation fails because these `target.Processor` properties are not specified:

- `Name` — Processor name
- `Id` — Object identifier

You can specify a processor name, which also specifies the object identifier.

```
myProcessor.Name = 'MyProcessor';
```

Check the validity of `myProcessor`.

```
myProcessor.IsValid
```

```
ans =
  logical
  1
```

The validity of the object is established.

When you use the `target.add` function to register a target feature object, the software also checks the validity of the object.

For more information, see Register New Hardware Devices.

Upgrade of hardware device definitions

R2019b provides the `target.upgrade` function, which enables you to upgrade existing definitions of hardware devices. The function uses a specific upgrade procedure to create objects from definitions in current data artifacts. By default, the function also creates the file, `registerUpgradedTargets.m`. To register the upgraded definitions, run `registerUpgradedTargets.m`.

For more information, see Upgrade Data Definitions for Hardware Devices.

Compatibility Considerations

Support for the use of `rtwTargetInfo.m` files to register hardware devices will be removed in a future release. To update the registration mechanism, use the `target.upgrade` function.

Performance

Improved function inlining readability and predictability

In R2019b, the code generator uses improved heuristics for function inlining. The generated code is more stable and robust to changes in MATLAB code that can influence function inlining. You can generate more consistent C/C++ code from different iterations of MATLAB code.

For more information on function inlining, see Control Inlining.

Deep Learning with MATLAB Coder

Deep Learning: Generate code for more layers and networks

Additional Layers

In R2019b, MATLAB Coder supports C++ code generation for additional deep learning layers.

Layer	Description	ARM Compute Library	Intel MKL-DNN
Custom output layers	All output layers including custom classification or regression output layers created by using `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`. For an example showing how to define a custom classification output layer and specify a loss function, see Define Custom Classification Output Layer (Deep Learning Toolbox). For an example showing how to define a custom regression output layer and specify a loss function, see Define Custom Regression Output Layer (Deep Learning Toolbox).	Yes	Yes
`globalAveragePooling2dLayer`	Global average pooling layer for spatial data	Yes	Yes
`groupedConvolution2dLayer`	2-D grouped convolutional layer	Yes If you specify an integer for `numGroups`, then the value must be less than or equal to `2`.	Yes
`nnet.keras.layer.FlattenCStyleLayer`	Flattens activations into 1-D assuming C-style (row-major) order	Yes	Yes

Layer	Description	ARM Compute Library	Intel MKL-DNN
`nnet.keras.layer.GlobalAveragePooling2dLayer`	Global average pooling layer for spatial data	Yes	Yes
`nnet.keras.layer.SigmoidLayer`	Sigmoid activation layer	Yes	Yes
`nnet.keras.layer.TanhLayer`	Hyperbolic tangent activation layer	Yes	Yes
`nnet.keras.layer.ZeroPadding2dLayer`	Zero padding layer for 2-D input	Yes	Yes
`nnet.onnx.layer.ElementwiseAffineLayer`	Layer that performs element-wise scaling of the input followed by an addition	Yes	Yes
`nnet.onnx.layer.FlattenLayer`	Flatten layer for ONNX™ network	Yes	Yes
`tanhLayer`	Hyperbolic tangent (tanh) layer	Yes	Yes
`transposedConv2dLayer`	Transposed 2-D convolution layer	Yes	Yes
`YOLOv2OutputLayer`	Output layer for YOLO v2 object detection network	Yes	Yes
`YOLOv2ReorgLayer`	Reorganization layer for YOLO v2 object detection network	Yes	Yes
`YOLOv2TransformLayer`	Transform layer for YOLO v2 object detection network	Yes	Yes

Additional Pretrained Networks

In R2019b, MATLAB Coder supports C++ code generation for this network.

Network Name	Description	ARM Compute Library	Intel MKL-DNN
`MobileNet-v2`	MobileNet-v2 convolutional neural network. For the pretrained MobileNet-v2 model, see `mobilenetv2`.	Yes	Yes

YOLO v2 Object Detector

In R2019b, you can generate code for an object detector trained by using a YOLO v2 network. This class is now supported for code generation for these target deep learning libraries.

Class	Description	ARM Compute Library	Intel MKL-DNN
`yolov2ObjectDetector`	• Only the `detect` method of the `yolov2ObjectDetector` is supported for code generation. • The `roi` argument to the `detect` method must be a code generation constant (`coder.const()`) and a 1x4 vector. • Only the `Threshold`, `SelectStrongest`, `MinSize`, and `MaxSize` name-value pairs for `detect` are supported. • The `labels` output of `detect` is returned as a cell array of character vectors, for example, `{'car','bus'}`.	Yes	Yes

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2019a

MATLAB Programming for Code Generation

Code generation support for class properties with string scalar initial values

In R2019a, you can generate code for a function that uses a class property whose initial value is a string scalar. For example:

```
classdef myClass
  properties
    prop = "myString";
  end
end

function out = useClass
obj = myClass;
out = obj.prop;
end
```

Code generation behavior change for character vector or string scalar input to ismethod

Starting in R2019a, specifying the first input (`obj`) to `ismethod` as a character vector or string scalar results in a code generation error.

Compatibility Considerations

In previous releases, if you specified the first input to `ismethod` as a character vector or string scalar, the code generator treated the value as the name of a class.

Code generation for categorical arrays

In R2019a, you can generate C/C++ code for categorical arrays. For more information, see Code Generation for Categorical Arrays.

Supported Functions

Code generation for sparse matrix inputs for more functions

These functions now support sparse matrix inputs for code generation:

- `chol`
- `tril`
- `triu`

Expanded code generation support for the vecdim input argument

Code generation now supports the `vecdim` input argument for these functions:

- `all`
- `any`
- `max`
- `mean`
- `median`
- `min`
- `mode`
- `prod`
- `std`
- `sum`
- `var`

Code generation for more MATLAB functions

- `arrayfun`
- `accumarray`
- `bounds`
- `categorical`
- `categories`
- `cellstr`
- `cospi`
- `dec2base`
- `discretize`
- `hasFrame`
- `ind2rgb`
- `iscalendarduration`
- `iscategory`
- `isdatetime`

- `isduration`
- `isordinal`
- `isprotected`
- `isundefined`
- `normalize`
- `pause`
- `read` of `VideoReader`
- `readFrame` of `VideoReader`
- `tic`
- `toc`
- `qrupdate`
- `rat`
- `sinpi`
- `VideoReader`

Code generation for more 5G Toolbox functions

- `getTransportBlock`
- `nrDLSCH`
- `nrDLSCHDecoder`
- `nrLowPAPRS`
- `nrPUCCH0`
- `nrPUCCH1`
- `nrPUCCH2`
- `nrPUCCH3`
- `nrPUCCH4`
- `nrPUCCHHoppingInfo`
- `nrPUCCHPRBS`
- `nrPUSCH`
- `nrPUSCHCodebook`
- `nrPUSCHDecode`
- `nrPUSCHDescramble`
- `nrPUSCHPRBS`
- `nrPUSCHScramble`
- `nrTransformPrecode`
- `nrTransformDeprecode`
- `nrUCIDecode`
- `nrUCIEncode`
- `nrULSCH`
- `nrULSCHDecoder`

- `nrULSCHInfo`
- `resetSoftBuffer`
- `setTransportBlock`

Code generation for more Audio Toolbox functions

- `gtcc`
- `harmonicRatio`
- `imdct`
- `kbdwin`
- `mdct`
- `melSpectrogram`
- `spectralCentroid`
- `spectralCrest`
- `spectralDecrease`
- `spectralEntropy`
- `spectralFlatness`
- `spectralFlux`
- `spectralKurtosis`
- `spectralRolloffPoint`
- `spectralSkewness`
- `spectralSlope`
- `spectralSpread`
- `erb2hz`
- `bark2hz`
- `mel2hz`
- `hz2erb`
- `hz2bark`
- `hz2mel`
- `octaveFilterBank`
- `gammatoneFilterBank`

Code generation for more Automated Driving Toolbox functions

The following Automated Driving Toolbox™ path planning functions and objects now support code generation:

- `vehicleDimensions`
- `inflationCollisionChecker`
- `vehicleCostmap`
- `checkFree`

- checkOccupied
- getCosts
- setCosts
- pathPlannerRRT
- plan
- driving.Path
- interpolate
- driving.DubinsPathSegment
- driving.ReedsSheppPathSegment
- checkPathValidity
- smoothPathSpline

Code generation for more Communications Toolbox functions

- algintrlv
- algdeintrlv
- comm.DecisionFeedbackEqualizer
- comm.DPD
- comm.DPDCoefficientEstimator
- comm.LinearEqualizer
- intrlv
- deintrlv
- matintrlv
- matdeintrlv
- helscanintrlv
- helscandeintrlv
- pammod
- genqammod
- pamdemod

Code generation for more Computer Vision Toolbox functions

- acfObjectDetector
- detect of acfObjectDetector
- pointCloud
- findNearestNeighbors
- findNeighborsInRadius
- findPointsInROI
- removeInvalidPoints
- select

- `pcdownsample`
- `pcfitcylinder`
- `pcfitplane`
- `pcfitsphere`
- `pctransform`
- `pcregistercpd`
- `pcmerge`
- `pcdenoise`
- `pcnormals`
- `pcsegdist`
- `segmentLidarData`
- `ORBPoints`
- `detectORBFeatures`
- `disparityBM`
- `disparitySGM`
- `opticalFlow`
- `estimateFlow`
- `reset`

Code generation for more DSP System Toolbox functions

- `dsp.STFT`
- `dsp.ISTFT`
- `dsp.FourthOrderSectionFilter`

Code generation for more Image Processing Toolbox functions

- `affine3d`
- `inpaintCoherent`
- `rgb2lightness`

Code generation for more Phased Array System Toolbox functions

- `azelcut2pat`
- `clone`
- `move`
- `rcscylinder`
- `rcsdisc`
- `rcstruncone`
- `release`
- `reflect`

- reset
- rotpat

Code generation for more Robotics System Toolbox functions

- robotics.LidarSLAM class and methods addScan, copy, removeLoopClosures, and scansAndPoses
- robotics.PoseGraph object, robotics.PoseGraph3D object and functions addRelativePose, edges, edgeConstraints, and findEdgeID, nodes, and removeEdges
- optimizePoseGraph
- robotics.DubinsConnection object and connect function
- robotics.DubinsPathSegment object and interpolate function
- robotics.ReedsSheppConnection object
- robotics.ReedsSheppPathSegment object
- uavOrbitFollower System object™
- bsplinepolytraj
- cubicpolytraj
- quinticpolytraj
- rottraj
- transformtraj
- trapveltraj

Code generation for more Sensor Fusion and Tracking Toolbox functions

- ahrs10filter
- append of ggiwphd
- AsyncMARGGPSFuser
- altimeterSensor
- clone of ggiwphd
- correct of ahrs10filter
- correct of AsyncMARGGPSFuser
- correct of ErrorStateIMUGPSFuser
- correct of ggiwphd
- correctjpda
- correctUndetected of ggiwphd
- ErrorStateIMUGPSFuser
- extractState of ggiwphd
- fuseaccel of AsyncMARGGPSFuser
- fusealtimeter of ahrs10filter
- fusegps of AsyncMARGGPSFuser

- `fusegps` of `ErrorStateIMUGPSFuser`
- `fusegyro` of `AsyncMARGGPSFuser`
- `fusemag` of `ahrs10filter`
- `fusemag` of `AsyncMARGGPSFuser`
- `fusemvo` of `ErrorStateIMUGPSFuser`
- `jpdaEvents`
- `extractState` of `ggiwphd`
- `ggiwphd`
- `initcaggiwphd`
- `initctggiwphd`
- `initcvggiwphd`
- `labeledDensity` of `ggiwphd`
- `likelihood` of `ggiwphd`
- `merge` of `ggiwphd`
- `partitionDetections`
- `pose` of `ahrs10filter`
- `pose` of `AsyncMARGGPSFuser`
- `pose` of `ErrorStateIMUGPSFuser`
- `predict` of `ahrs10filter`
- `predict` of `AsyncMARGGPSFuser`
- `predict` of `ErrorStateIMUGPSFuser`
- `predict` of `ggiwphd`
- `prune` of `ggiwphd`
- `randrot`
- `reset` of `ahrs10filter`
- `reset` of `AsyncMARGGPSFuser`
- `reset`of `ErrorStateIMUGPSFuser`
- `scale` of `ggiwphd`
- `stateinfo` of `ahrs10filter`
- `stateinfo` of `AsyncMARGGPSFuser`
- `stateinfo` of `ErrorStateIMUGPSFuser`
- `trackerJPDA`
- `trackerPHD`
- `trackingSensorConfiguration`

Code Generation for SerDes Toolbox Functions

- `serdes.AGC`
- `serdes.CDR`
- `serdes.CTLE`

- `serdes.DFECDR`
- `serdes.FFE`
- `serdes.PassThrough`
- `serdes.SaturatingAmplifier`
- `serdes.VGA`

Code generation for more Signal Processing Toolbox functions

The following Signal Processing Toolbox functions now support C/C++ code generation:

- **Filter Design and Filtering:**

  `buttap`, `filtfilt`, `filtord`, `fir1`, `firls`, `kaiserord`, and `sos2tf`
- **Spectral Analysis:**

  `cpsd`, `czt`, `goertzel`, `mscohere`, `periodogram`, and `pwelch`
- **Time-Frequency Analysis**

  `iscola`, `istft`, and `stft`
- **Spectral Windows:**

  `barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `flattopwin`, `gausswin`, `nuttallwin`, `parzenwin`, `rectwin`, `taylorwin`, `triang`, and `tukeywin` now accept variable input.
- **Waveform Generation:**

  `gauspuls`, `pulstran`, `rectpuls`, `square`, and `tripuls`
- **Linear Predictive Coding:**

  `lsf2poly`, `poly2ac`, `poly2lsf`, `poly2rc`, `rc2ac`, `rc2poly`, and `rlevinson`

Code generation for more Statistics and Machine Learning Toolbox functions

- You can generate C/C++ code that predicts responses by using trained naive Bayes models.
  - `predict` — Classify observations, estimate posterior probabilities, or compute misclassification costs by applying a naive Bayes classification model to new data.
- You can specify a custom binary loss function for the `predict` function of the `CompactClassificationECOC`.
- The following functions support code generation:
  - `ksdensity` — Find kernel smoothing function estimates for univariate and bivariate data.
  - `mvksdensity` — Find kernel smoothing function estimates for multivariate data.
  - `ecdf` — Estimate empirical cumulative distribution function values.

Code generation for more Wavelet Toolbox functions

- `cwtfilterbank`
- `cwtfreqbounds`

Code generation for more WLAN Toolbox functions

MAC Frame Parser

- `wlanAMPDUDeaggregate`
- `wlanMPDUDecode`
- `displayIEs` of `wlanMACManagementConfig`

IEEE 802.11ax Signal Recovery

- `wlanHERecoveryConfig`
- `getSIGBLength` of `wlanHERecoveryConfig`
- `interpretHESIGABits` of `wlanHERecoveryConfig`
- `wlanHESIGABitRecover`
- `wlanLSIGBitRecover`

OFDM Demodulation and Information

- `wlanHEDemodulate`
- `wlanDMGOFDMDemodulate`
- `wlanS1GDemodulate`
- `wlanHEOFDMInfo`
- `wlanVHTOFDMInfo`
- `wlanDMGOFDMInfo`
- `wlanHTOFDMInfo`
- `wlanNonHTOFDMInfo`
- `wlanS1GOFDMInfo`

Reference Symbols

- `wlanReferenceSymbols`
- `wlanClosestReferenceSymbol`

802.11ay Channel Model

- `wlanTGayChannel`
- `info` of `wlanTGayChannel`
- `showEnvironment` of `wlanTGayChannel`
- `wlanURAConfig`

Code Generation Workflow

Access information about code generation programmatically by using the report information object

In R2019a, you can export information about code generation to a variable in your base MATLAB workspace. This variable contains an object whose properties contain information about:

- Code generation settings
- Input files
- Generated files
- Code generation error, warning, and information messages

To export code generation report information to the variable `info` in your base MATLAB workspace, do one of the following:

- In the MATLAB Coder app, on the **Debugging** tab, set **Export report information to variable** to the variable name `info`.
- At the command line, use the `codegen` command with the `-reportinfo` option. Specify the variable name after the `-reportinfo` option.

  ```
  codegen myFunction -reportinfo info
  ```
- At the command line, set the code configuration object property `ReportInfoVarName` to the character vector `'info'`.
- Generate and open the code generation report. Click **Export Report Information**. In the dialog box, specify the variable name `info`.

You can use the report information object to programmatically access information about code generation. For example, you can display the code generation messages at the MATLAB command line. To perform this action, in your build script, access the property that contains these messages.

For more information, see Access Code Generation Report Information Programmatically and `coder.ReportInfo Properties`.

Open code generation reports in any MATLAB installation without MATLAB Coder

In R2019a, you do not need MATLAB Coder to open a code generation report. You can open the report in any MATLAB installation. As a result, you can also share a code generation report with other MATLAB users who do not have MATLAB Coder.

For more information on the report, see Code Generation Reports.

Generate parallel for-loops on macOS platform

In R2019a, you can generate parallel `for`-loops on the macOS platform by using `parfor` in your MATLAB code. In previous releases, the code generated for `parfor`-loops on the macOS platform was not parallelized.

To run the code generated for a `parfor`-loop outside of MATLAB, you must install an OpenMP library. See Install OpenMP Library on macOS Platform.

Compatibility Considerations

If you generate code for legacy MATLAB Coder projects that use `parfor`-loops on the macOS platform and run the generated code outside of MATLAB, you must install an OpenMP library.

Alternatively, you can restore the legacy behavior of not parallelizing `parfor`-loops on the macOS platform. Do one of the following:

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set the `EnableOpenMP` property to `false`.
- In the MATLAB Coder app, on the **Advanced** tab, set **Enable OpenMP library if possible** to `No`.

Register new hardware devices

Extend the range of supported hardware by using the `target.Processor` and `target.LanguageImplementation` classes to register new devices.

For details, see Register New Hardware Devices.

Functionality being removed or changed

Template makefile (TMF) support will be removed
Warns

Support for template makefiles (TMF) will be removed in a future release.

By default, MATLAB Coder uses a toolchain approach to build libraries and executable programs. Alternatively, you can specify use of a template makefile. In a future release, MATLAB Coder will stop supporting template makefiles. If a configuration object or a project specifies a template makefile, modify the object or project to specify the toolchain approach. Use `coder.make.upgradeCoderConfigObject` or `coder.make.upgradeMATLABCoderProject`. See Project or Configuration Is Using the Template Makefile.

To customize the build process, see Build Process Customization. To create a custom toolchain, see Custom Toolchain Registration.

Performance

Faster C/C++ MEX function generation

MATLAB Coder creates a MEX function by generating and compiling C/C++ code or by using Just-In-Time (JIT) compilation technology. In R2019a, MEX creation with C/C++ compilation is faster than in previous releases. For more information about MEX generation with JIT compilation, see Speed Up MEX Generation by Using JIT Compilation.

Generated code quality improvements

R2019a includes these generated code quality improvements:

- Loop fusion for more cases, including variable-size upper bounds.
- Loop fusion only when the code generator determines that the benefits of loop fusion outweigh the cost.
- Elimination of some unnecessary assignments to global variables.

Loop fusion is an optimization that combines successive loops that have the same number of runs into a single loop. See MATLAB Coder Optimizations in Generated Code.

Deep Learning with MATLAB Coder

Deep Learning: Generate code for prediction on ARM processors by using codegen

In R2018b, you could generate C++ code for prediction from a trained convolutional neural network (CNN) by using `cnncodegen`. In R2019a, you can use `codegen` or the MATLAB Coder app. See Code Generation for Deep Learning Networks with ARM Compute Library.

Deep Learning: Generate code for more networks and layers

In R2019a, MATLAB Coder supports additional deep learning layers. With the additional layers, you can generate code for more networks.

Additional Layers Supported for MKL-DNN

Layer	Network
Layer that applies 2-D cropping to the input (`crop2dLayer`)	--
Max unpooling layer (`maxUnpooling2dLayer`)	SegNet
Global average pooling layer for spatial data (`nnet.keras.layer.GlobalAveragePooling2dLayer`)	Some networks imported from Keras
Scaling layer	Inception-v3

Additional Layers Supported for ARM Compute Library

Layer	Network
Clipped Rectified Linear Unit (ReLU) layer (`clippedReluLayer`)	--
Leaky Rectified Linear Unit (ReLU) layer (`leakyReluLayer`)	--
Global average pooling layer for spatial data (`nnet.keras.layer.GlobalAveragePooling2dLayer`)	Some networks imported from Keras
Scaling layer	Inception-v3

See Deep Learning Networks and Layers Supported for C++ Code Generation.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2018b

MATLAB Programming for Code Generation

Sparse Matrices: Generate code for the backslash operation

MATLAB Coder now supports code generation for the backslash operation on sparse matrices. The backslash operation (also called `mldivide` and matrix left division) solves linear systems of equations. To implement backslash, MATLAB Coder uses the SuiteSparse library and code from the CXSparse package. See https://faculty.cse.tamu.edu/davis/suitesparse.html.

Statistics and Machine Learning Toolbox Code Generation: Update deployed SVM model without regenerating code

If you have Statistics and Machine Learning Toolbox, you can generate C/C++ code for the prediction of a support vector machine (SVM) model using a coder configurer, then update model parameters of a deployed SVM model without having to regenerate the code. After training an SVM model, use the `learnerCoderConfigurer` function to create a coder configurer object, `ClassificationSVMCoderConfigurer` for an SVM classification model or `RegressionSVMCoderConfigurer` for an SVM regression model. A coder configurer offers convenient features to configure code generation options, generate C/C++ code, and update model parameters in the generated code.

- Configure code generation options and specify the coder attributes of SVM model parameters using object properties.
- Generate C/C++ code for the `predict` and `update` functions of the SVM model by using `generateCode`. Generating C/C++ code requires MATLAB Coder.
- Update model parameters in the generated C/C++ code without having to regenerate the code. This feature reduces the effort required to regenerate, redeploy, and reverify C/C++ code when you retrain the SVM model with new data or settings. Before updating model parameters, use `validatedUpdateInputs` to validate and extract the model parameters to update.

Class Support: Use objects in more functions and data types

These functions and data types now support value classes for code generation:

- `coder.load`
- `load`
- Global variables

Supported Functions

Sensor Fusion and Tracking Toolbox Code Generation: Generate code to accelerate and deploy your algorithm

For the list of Sensor Fusion and Tracking Toolbox™ functions supported for code generation in R2018b, see Sensor Fusion and Tracking Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

5G Toolbox Code Generation: Generate code for downlink physical layer

For the list of 5G Toolbox™ functions supported for code generation in R2018b, see 5G Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Fuzzy Logic Toolbox Code Generation: Generate code to load and evaluate Fuzzy Inference Systems

In R2018b, Fuzzy Logic Toolbox™ supports C/C++ code generation for built-in membership functions and these functions:

- `evalfis`
- `evalfisOptions`
- `getFISCodeGenerationData`

For the list of Fuzzy Logic Toolbox functions supported for code generation in R2018b, see Fuzzy Logic Toolbox in Functions and Objects Supported for C/C++ Code Generation — Category List.

Code generation for more MATLAB Functions

- `lscov`
- `func2str`
- `num2str`

Code generation for rng 'shuffle' option

Code generation now supports the `rng 'shuffle'` option.

The generated code for `rng('shuffle')` might produce different seeds than MATLAB produces.

Code generation for more Audio Toolbox functions

- `interpolateHRTF`
- `pitch`

Code generation for more Automated Driving System Toolbox functions

- `lateralControllerStanley`

Code generation for more Communications Toolbox functions

- `awgn`
- `bsc`
- `wgn`

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

- `phased.PulseCompressionLibrary`
- `phased.RangeAngleResponse`
- `phased.MonopulseFeed`
- `getMonopulseEstimator`
- `phased.MonopulseEstimator`
- `AlphaBetaFilter` and object functions `clone`, `correct`, `distance`, `likelihood`, and `predict`

Code generation for more Robotics System Toolbox functions

- `quaternion` and `quaternion` object functions
- `fixedwing`
- `multirotor`
- `uavWaypointFollower`
- `control`
- `derivative`
- `environment`
- `state`

Code generation for more Statistics and Machine Learning Toolbox functions

- `coxphfit`
- `update` of `CompactRegressionSVM`
- `update` of `CompactClassificationSVM`

Code generation for more WLAN Toolbox functions

- `wlanHEDataBitRecover`

- `wlanHEMUConfig`
- `wlanHESUConfig`
- `wlanMACFrame`
- `wlanMACFrameConfig`
- `wlanMACManagementConfig`
- `wlanMSDULengths`

Generated Code Improvements

Standard Math Library: Default to C99 standard math library for C

In R2018b, the default standard math library for C code generation is C99 (ISO) or ISO/IEC 9899:1999. In previous releases, the default standard math library was C89/C90 (ANSI). The C99 library provides more functionality, such as the `bool` data type and built-in constants for `Inf` and `NaN`. Type casting can also be reduced.

Compatibility Considerations

- The compiler that you use to generate code must be compatible with C99.
- When using C99, the bit pattern for `NaN` in the generated code can be different from the bit pattern for `NaN` in MATLAB. Both `NaN` representations comply with the IEEE 754-1985 standard. To revert to C89/C90, see Change the Standard Math Library.

Code Generation Workflow

MATLAB Support Package for Raspberry Pi Hardware: Deploy MATLAB function to Raspberry Pi

With MATLAB Support Package for Raspberry Pi® Hardware, you can interactively control a Raspberry Pi from your MATLAB host computer. Now, if you also have MATLAB Coder, the support package provides commands to automate deployment of your MATLAB function as a standalone executable on the Raspberry Pi. Use `targetHardware` to create a Raspberry Pi configuration object. Then, use `deploy` to deploy the function to the Raspberry Pi hardware.

Many functions in the support package are supported for code generation. For information about code generation support, see the function reference pages in the documentation for the support package MATLAB Support Package for Raspberry Pi Hardware.

Multiple Entry-Point Functions: Simplify input specification by passing an output as an input

In R2018b, the `coder.OutputType` function enables you to reuse the output type from one entry-point function as the input type to another entry-point function. You can:

- Simplify input specification when multiple entry-point functions use the same data type.
- Ensure synchronized type definitions across entry-point function interfaces.
- More easily partition and extend your code into multiple entry-point functions to provide greater functionality or versatility.

For more information, see Pass an Entry-Point Function Output as an Input.

Changes to Check for Run-Time Issues step in the MATLAB Coder app

R2018b, the **Check for Run-Time Issues** step of the MATLAB Coder app includes these changes:

- The app no longer accumulates persistent data across runs of **Check for Issues** and no longer generates a link for you to clear the MEX function.
- As in previous releases, line execution counts accumulate across runs of **Check for Issues**. To reset the counts, use the new **Reset line execution counts** link.

See Collect and View Line Execution Counts for Your MATLAB Code.

Performance

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

To improve the execution speed of code generated from algorithms that perform certain low-level vector and matrix computations (such as matrix multiplication), MATLAB Coder can generate calls to BLAS functions by using the CBLAS C interface to BLAS. If the input arrays for the matrix operations meet certain criteria, the code generator produces calls to relevant BLAS functions. In previous releases, only generated MEX called BLAS functions. In R2018b, generated standalone code can call BLAS functions.

BLAS is a software library for numeric computation of basic vector and matrix operations that has several highly optimized machine-specific implementations. MATLAB uses this library for basic matrix computations. For MEX functions, the code generator uses the BLAS library that is included with MATLAB. For standalone code, the code generator uses the CBLAS interface for the BLAS library that you specify. If you do not specify a BLAS library, the code generator produces code for the matrix operation instead of generating a BLAS call.

See Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls.

Compiler Support: Revert to JIT compilation for MEX code generation when supported compiler not detected

For MEX code generation, the code generator defaults to JIT compilation when a supported compiler is not detected. JIT compilation does not support custom code called through `coder.ceval` or the use of external libraries. For full functionality, install a supported compiler. See Supported and Compatible Compilers on the MathWorks website.

Nonfinite Support as Needed: Generate files for nonfinite data support only when the generated code uses nonfinite data

To support nonfinite data (`NaN` and `Inf`), the code generator produces the files `rt_nonfinite.c`, `rtGetInf.c`, and `rtGetNaN.c`, and associated header files. In previous releases, if you enabled support for nonfinite data, the code generator always produced these files. In R2018b, if you enable support for nonfinite data, the code generator produces the files only if the generated code uses nonfinite data.

Compatibility Considerations

If necessary for your legacy code, you can generate the nonfinite data support files even if the generated code does not use nonfinite data.

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set `SupportNonFinite` to `true` and `GenerateNonFiniteFilesIfUsed` to `false`.
- In the MATLAB Coder app, on the **All Settings** tab, set **Support nonfinite numbers** to `Yes` and **Generate nonfinite support files if used** to `No`.

Loop Unrolling Threshold: Optimize code generated for loops

In R2018b, the code generator uses a configurable threshold to determine whether to automatically unroll a `for`-loop.

When the code generator unrolls a `for`-loop, instead of producing a `for`-loop in the generated code, it produces a copy of the loop body for each iteration. For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can significantly increase code generation time and generate inefficient code.

If the number of loop iterations is less than the threshold, the code generator automatically unrolls the loop. If the number of iterations is greater than or equal to the threshold, the code generator produces a `for`-loop. The default value of the threshold is `5`. By modifying the threshold, you can fine-tune loop unrolling. To modify the threshold:

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set the `LoopUnrollThreshold` property.
- In the MATLAB Coder app, on the **Speed** tab, set **Loop unrolling threshold** .

For more details, see Unroll `for`-Loops.

The `memcpy` optimization can replace either a `for`-loop or individual copies of the loop body with a `memcpy` call. See memcpy Optimization.

Deep Learning with MATLAB Coder

Deep Learning Network Code Generation: Generate C++ code for inference from a trained convolutional neural network

In R2018b, you can use MATLAB Coder with Deep Learning Toolbox to generate C++ code for deep learning networks. You can generate code for inference from a convolutional neural network (CNN) that you train by using Deep Learning Toolbox or from a pretrained network that you import.

You can generate code for Intel CPUs or ARM processors. For Intel CPUs, the code generator takes advantage of the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). For ARM processors, the code generator takes advantage of the ARM Compute libraries. You must install the library for your processor. You must also install the support package MATLAB Coder Interface for Deep Learning.

For more information, see Deep Learning with MATLAB Coder.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2018a

MATLAB Programming for Code Generation

Sparse Matrices: Enable more efficient computation by using sparse matrices in generated code

In R2018a, you can generate code that uses sparse matrices and sparse matrix operations. Sparse matrices provide efficient storage for arrays with many zero elements. In generated code, sparse matrices can improve performance and reduce memory usage compared with full matrices. Computation time on sparse matrices scales only with the number of operations on nonzero elements.

In R2018a, code generation support focuses on sparse matrix construction, arithmetic, concatenation, indexing, and some element-wise functions. See Code Generation for Sparse Matrices.

Delete Method: Call clean-up code automatically when handle classes are deleted in generated code

In 2018a, you can generate code for MATLAB code that uses `delete` methods (destructors) for handle classes. To perform clean-up operations, such as closing a previously opened file before an object is destroyed, use a `delete` method. For the guidelines and restrictions on code generation for `delete` methods, see Code Generation for Handle Class Destructors.

Compatibility Considerations

In 2017b, you could explicitly call the `delete` method of the handle superclass, but not of any derived class. In 2018a, you cannot explicitly call `delete` methods for code generation. The generated code automatically calls them. If you attempt to explicitly call a `delete` method, you get an error.

Cell Array Support: Import cell arrays into generated code by using coder.load and load

In previous releases, you could not import cell arrays from saved data into generated code. In R2018a, cell arrays are supported for loading. For example, save a cell array in the file `data.mat`:

```
x = {1,'a',3};
save('data.mat', 'x');
```

Write a function `foo` that imports the cell array data.

```
function [out1, out2] = foo
    compileTimeData = coder.load('data.mat');
    out1 = compileTimeData.x;
    runTimeData = load('data.mat');
    out2 = runTimeData.x;
end
```

In R2018a, you can generate a MEX function `foo_mex` for this code:

```
codegen foo
```

Choose `coder.load` or `load` depending on whether you want to update saved data at run time. The function `load` is valid only for MEX code and Simulink simulation. If you change the values of data.mat and rerun the MEX function foo_mex, the second output variable out2 is updated to the new values at run time.

Supported Functions

Statistics and Machine Learning Toolbox Code Generation: Generate code for distance calculation on vectors and matrices, and for prediction by using k-nearest neighbor with Kd-tree search and nontree ensemble models

The following functions support code generation:

- `grp2idx` — Create an index vector from a grouping variable.
- `pdist` — Find the pairwise distance between pairs of observations.
- `squareform` — Format a distance matrix.

When you train a *k*-nearest neighbor classification model by using `fitcknn` for code generation, you can now use the *K*d-tree search algorithm. For more details, see the C/C++ Code Generation section of the `ClassificationKNN` class.

When you find nearest neighbors by using the functions `knnsearch` and `rangesearch` and the object functions `knnsearch` and `rangesearch` for code generation, you can now use the *K*d-tree search algorithm.

When you train an ensemble by using `fitcensemble` for code generation, you can now specify `'discriminant'` or `'knn'` as weak learners by using the `'Learners'` name-value pair argument. For more details, see the C/C++ Code Generation section of the `CompactClassificationEnsemble` class.

Code generation for MATLAB sprintf function

In R2018a, you can generate code from your MATLAB code that uses `sprintf` to construct formatted strings and character arrays.

Code generation for MATLAB sort function options

In R2018a, code generation supports additional options for MATLAB sort functions:

- Code generation of `issorted`, `issortedrows`, `sort`, and `sortrows` supports the `ComparisonMethod` and `MissingPlacement` options.
- Code generation of `issorted` and `issortedrows` supports the `'monotonic'`, `'strictascend'`, `'strictdescend'`, and `'strictmonotonic'` values for sorting direction.
- Code generation of `issortedrows` supports specification of the sorting direction as a cell array of character vectors.

Code generation support for `issortedrows` is new for R2018a.

Code generation for more MATLAB functions

- `maxk`
- `mink`

- `nzmax`
- `rescale`
- `spalloc`
- `sparse`
- `spdiags`
- `speye`
- `spfun`
- `spones`
- `vecnorm`

Code generation for more Audio Toolbox functions

- `cepstralFeatureExtractor`
- `splMeter`
- `voiceActivityDetector`

Code generation for more Communications System Toolbox functions

- `apskdemod`
- `apskmod`
- `dvbsapskdemod`
- `dvbsapskmod`
- `mil188qamdemod`
- `mil188qammod`
- `tpcdec`
- `tpcenc`

Code generation for more Computer Vision System Toolbox functions and objects

- `detectKAZEFeatures`
- `KAZEPoints`

Code generation for more DSP System Toolbox functions

- `dsp.ComplexBandpassDecimator`
- `getRateChangeFactors` object function of the `dsp.FarrowRateConverter` and `dsp.SampleRateConverter` System objects.

Code generation for more Phased Array System Toolbox System objects

- `phased.PulseWaveformLibrary`

Code generation for more Robotics System Toolbox functions

- `matchScansGrid`

Code generation for more Signal Processing Toolbox functions

- `emd`

Code generation for more Wavelet Toolbox functions

- `emd`

Code generation for more WLAN System Toolbox System objects

- `wlanTGaxChannel`

Generated Code Improvements

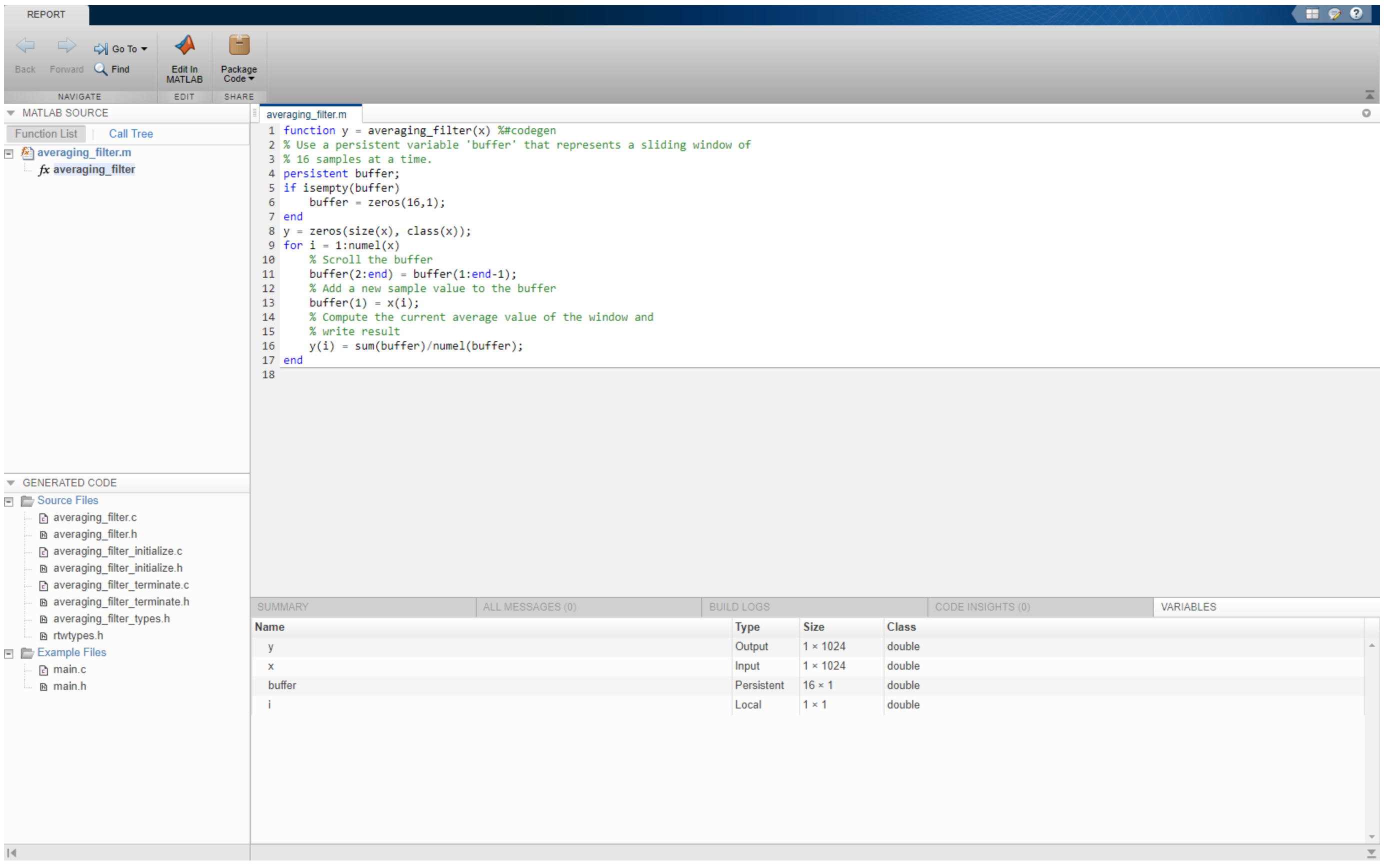
N-Dimensional Indexing: Enhance readability by preserving array dimensions in generated code

By default, the code generator creates one-dimensional arrays in C/C++ code for N-dimensional arrays in MATLAB code. C/C++ code that uses N-dimensional indexing can be easier to read and in certain cases, better suited for external code integration. In R2018a, you can enable full N-dimensional indexing. See Generate Code That Uses N-Dimensional Indexing.

Code Generation Workflow

New Code Generation Report: View more information and navigate through code generation results more easily

In R2018a, the MATLAB Coder code generation report has a new user interface, more information, additional functionality, and improved navigation. If you have Embedded Coder, the new report also provides interactive, bidirectional tracing between MATLAB and generated C/C++ code.



```
function y = averaging_filter(x) %#codegen
% Use a persistent variable 'buffer' that represents a sliding window of
% 16 samples at a time.
persistent buffer;
if isempty(buffer)
    buffer = zeros(16,1);
end
y = zeros(size(x), class(x));
for i = 1:numel(x)
    % Scroll the buffer
    buffer(2:end) = buffer(1:end-1);
    % Add a new sample value to the buffer
    buffer(1) = x(i);
    % Compute the current average value of the window and
    % write result
    y(i) = sum(buffer)/numel(buffer);
end
```

Name	Type	Size	Class
y	Output	1 × 1024	double
x	Input	1 × 1024	double
buffer	Persistent	16 × 1	double
i	Local	1 × 1	double

You can now:

- Find more information on the **Summary** tab, including code generation settings and your entry-point functions with the input argument data types that you specified.
- Look in one place, the **Code Insights** tab, for potential issues in the generated code.
- Package generated C/C++ files for relocation to another development environment.
- Navigate from the MATLAB code to context-sensitive information. For example, if you double-click a variable in the MATLAB code, you see the variable in the **Variables** tab.

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

See Code Generation Reports.

Compatibility Considerations

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

MEX Profiling: See execution times of generated MEX functions in MATLAB Profiler

In R2018a, you can profile execution times for MEX functions that you generate by using MATLAB Coder. Profiling the generated MEX can help you to identify performance issues early in the development cycle.

To use the MATLAB Profiler with a generated MEX function:

1. Enable MEX profiling.

   ```
   cfg = coder.config('mex');
   cfg.EnableMexProfiling = true;
   ```

   Alternatively, you can use `codegen` with the `-profile` option.

   The equivalent setting in the MATLAB Coder app is **Enable execution profiling**.
2. Generate the MEX file.
3. Run the MATLAB Profiler and view the Profile Summary Report.

   ```
   profile on;
   myFunction_mex;
   profile viewer;
   ```

If you have a test file that calls your MATLAB function, you can:

- Generate the MEX function and profile it in one step by using `codegen` with the `-profile` and `-test` options. If you turned on the MATLAB Profiler before, turn it off before you use these two options together.

  ```
  codegen MyFunction -test MyFunctionTest -profile
  ```
- Profile the MEX function in the **Verify** step of the app. If you turned on the MATLAB Profiler before, turn it off before you perform this action.

For more information, see Profile MEX Functions by Using MATLAB Profiler.

Run-Time Error Detection Support for error: Use error with run-time error detection and reporting in standalone code

In R2018a, run-time error detection in standalone code supports `error`. If the error condition occurs, execution terminates with a message that an error occurred. To see the actual message specified by `error`, generate and run a MEX function. See Run-Time Error Detection and Reporting in Standalone C/C++ Code.

Performance

Row-Major Array Layout: Simplify interfacing generated code with C environments by storing arrays in row-major layout

The code that you generate can store array elements in column-major or row-major array layout. In column-major array layout, the elements of the columns are contiguous in memory. In row-major, the elements of the rows are contiguous. MATLAB uses column-major array layout by default, whereas the C/C++ languages use row-major layout by default.

In previous releases, the code generator produced C/C++ code that used column-major array layout. In R2018a, you can choose to generate code that uses row-major array layout. Row-major layout can improve performance for certain algorithms and ease integration with other code that also uses row-major layout. For more information, see Array Layout.

More optimizations in generated code

R2018a includes these additional code generation optimizations:

- The code generator can now use the loop invariant code motion optimization with code that contains global or persistent variables. 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.
- If a while loop in the generated code would execute exactly one time, the code generator can eliminate it and replace it with the body of the loop.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2017b

MATLAB Programming for Code Generation

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.

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.

Supported Functions

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 `fitcsvm` 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 `fitclinear` for code generation, you can now specify either `'svm'` or `'logistic'` for the `'Learner'` name-value pair argument.

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`
- `strlength`

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 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`

- `range2tl`
- `sonareqsl`
- `sonareqsnr`
- `sonareqtl`
- `tl2range`

Code generation for more Robotics System Toolbox functions

- `lidarScan`
- `matchScans`

Code generation for more System Identification Toolbox objects

- `particleFilter`

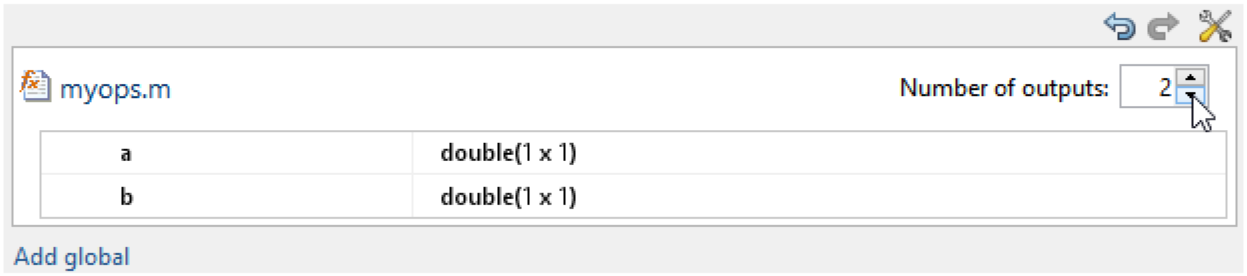
Code Generation for more WLAN System Toolbox functions

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

Code Generation Workflow

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.


myops.m

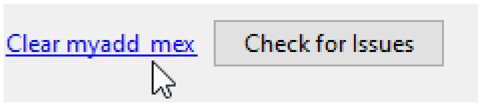
Number of outputs: 2

a	double(1 x 1)
b	double(1 x 1)

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.

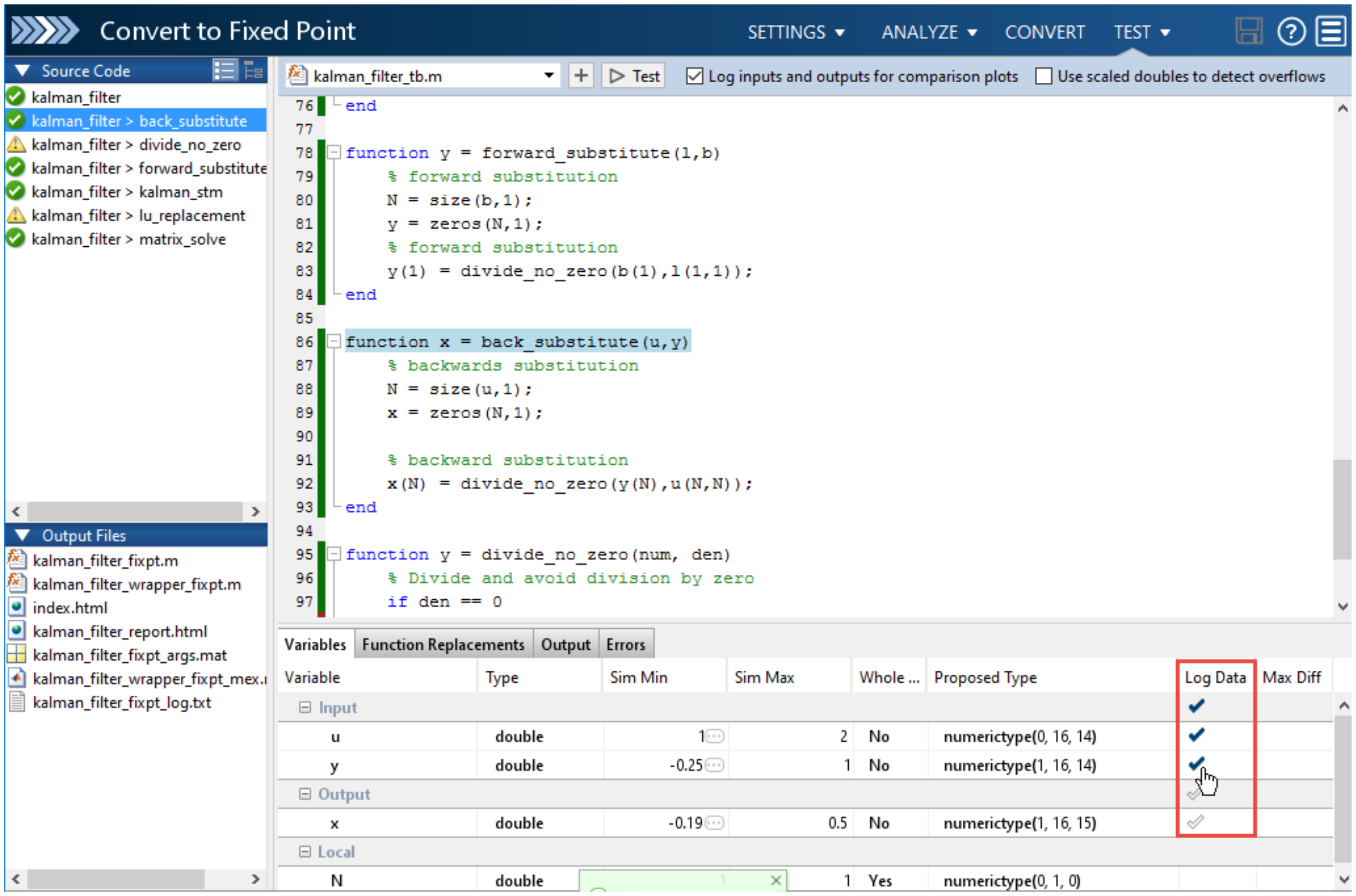


Clearing the MEX function resets data, such as persistent variables or line execution counts, that the **Check for Run-Time Issues** step accumulates.

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.

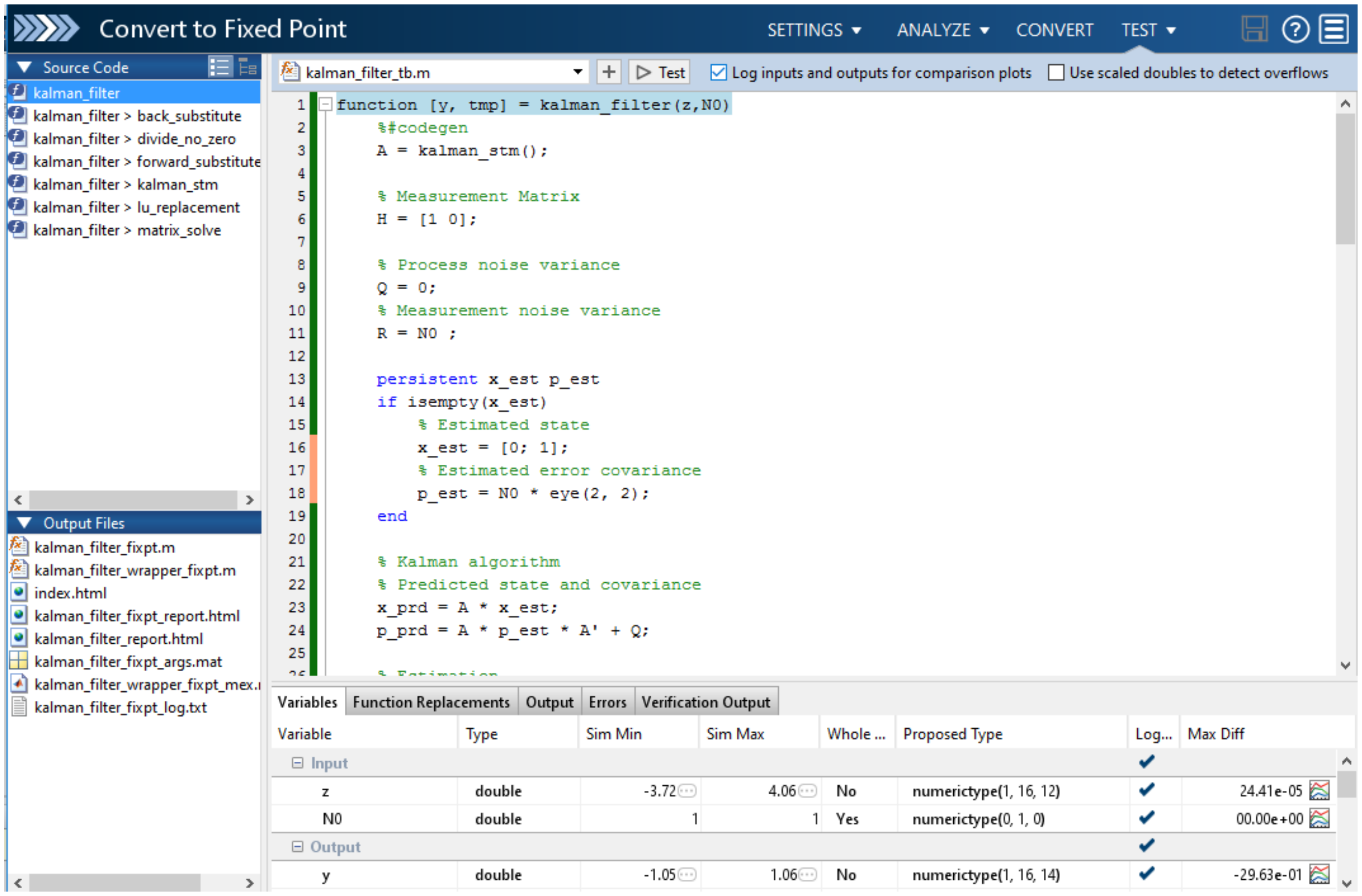

Convert to Fixed Point
SETTINGS
ANALYZE
CONVERT
TEST
Source Code
kalman_filter
kalman_filter > back_substitute
kalman_filter > divide_no_zero
kalman_filter > forward_substitute
kalman_filter > kalman_stm
kalman_filter > lu_replacement
kalman_filter > matrix_solve
Output Files
kalman_filter_fixpt.m
kalman_filter_wrapper_fixpt.m
index.html
kalman_filter_report.html
kalman_filter_fixpt_args.mat
kalman_filter_wrapper_fixpt_mex.
kalman_filter_fixpt_log.txt
kalman_filter_tb.m
Test
Log inputs and outputs for comparison plots
Use scaled doubles to detect overflows
Variables
Function Replacements
Output
Errors

```
end

function y = forward_substitute(l,b)
    % forward substitution
    N = size(b,1);
    y = zeros(N,1);
    % forward substitution
    y(1) = divide_no_zero(b(1),l(1,1));
end

function x = back_substitute(u,y)
    % backwards substitution
    N = size(u,1);
    x = zeros(N,1);

    % backward substitution
    x(N) = divide_no_zero(y(N),u(N,N));
end

function y = divide_no_zero(num, den)
    % Divide and avoid division by zero
    if den == 0
```

Variable	Type	Sim Min	Sim Max	Whole ...	Proposed Type	Log Data	Max Diff
Input						✓	
u	double	1	2	No	numerictype(0, 16, 14)	✓	
y	double	-0.25	1	No	numerictype(1, 16, 14)	✓	
Output						✓	
x	double	-0.19	0.5	No	numerictype(1, 16, 15)	✓	
Local							
N	double	1	1	Yes	numerictype(0, 1, 0)		

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
kalman_filter > back_substitute
kalman_filter > divide_no_zero
kalman_filter > forward_substitute
kalman_filter > kalman_stm
kalman_filter > lu_replacement
kalman_filter > matrix_solve
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.
kalman_filter_fixpt_log.txt
kalman_filter_tb.m
Test
Variables
Function Replacements
Output
Errors
Verification Output

[✓] Log inputs and outputs for comparison plots [ ] Use scaled doubles to detect overflows

```
function [y, tmp] = kalman_filter(z,N0)
    %#codegen
    A = kalman_stm();

    % Measurement Matrix
    H = [1 0];

    % Process noise variance
    Q = 0;
    % Measurement noise variance
    R = N0 ;

    persistent x_est p_est
    if isempty(x_est)
        % Estimated state
        x_est = [0; 1];
        % Estimated error covariance
        p_est = N0 * eye(2, 2);
    end

    % Kalman algorithm
    % Predicted state and covariance
    x_prd = A * x_est;
    p_prd = A * p_est * A' + Q;
```

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.

Performance

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.

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.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

MATLAB Programming for Code Generation

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.varsize` 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.

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.

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.

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);
```

Supported Functions

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.

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 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.

Generated Code Improvements

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 `emxArray`s 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.

`assert`	`limits`	`stdatomic`	`string`
`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	strstream
chrono	cuchar	map	system_error
cinttypes	cwchar	memory	thread
ciso646	cwctype	memory_resource	tuple
climits	deque	mutex	type_traits
clocale	exception	new	typeindex
cmath	execution	numeric	typeinfo
codecvt	filesystem	optional	unordered_map
complex	foreward_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.

Code Generation Workflow

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.

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`.

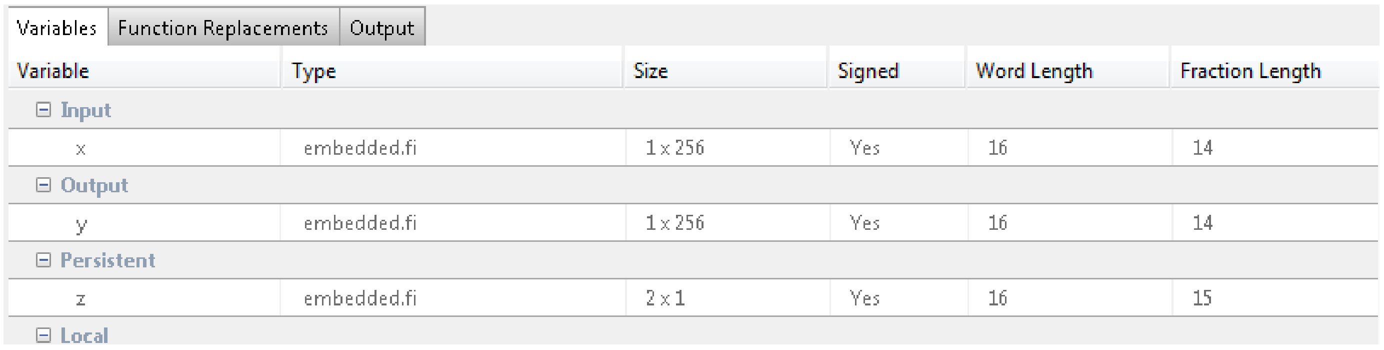
The `-test` option is supported only when generating MEX functions or when using a configuration object with `VerificationMode` set to `'SIL'`. Creation of a configuration object that has the `VerificationMode` parameter requires the Embedded Coder product.

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

See Verify MEX Functions at the Command Line.

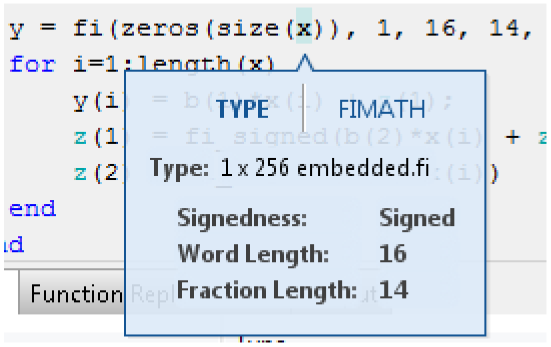
More fixed-size variable information in Convert to Fixed-Point step of MATLAB Coder app

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
    z(1
    z(2
end
```

Type: 1 x 256 embedded.fi

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.

Performance

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.

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.

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.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

MATLAB Programming for Code Generation

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.

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.

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'));
```

Supported 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.

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 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 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`.
- `extendedKalmanFilter`.
- `unscentedKalmanFilter`.
- `unscentedKalmanFilter`.

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

Generated Code Improvements

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;
}
```

Code Generation Workflow

Change to default standard math library for C++

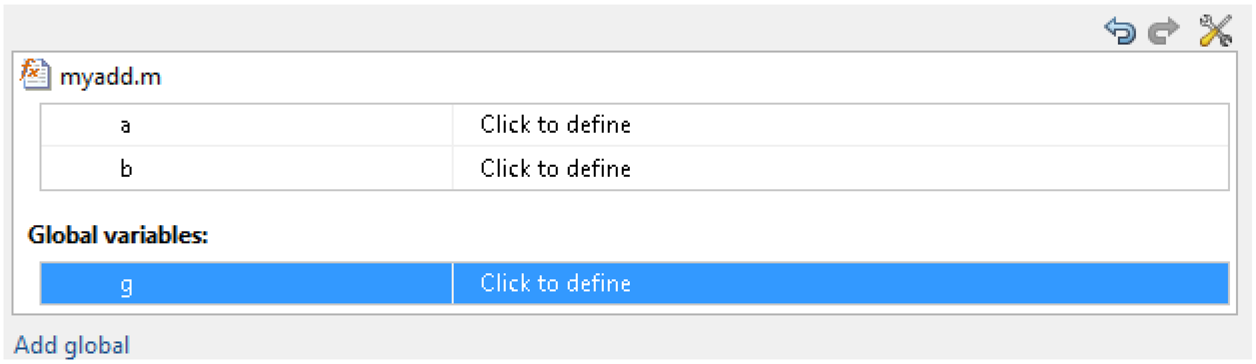
In R2016b, the default standard math library for C++ is ISO/IEC 14882:2003 C++ (`C++03 (ISO)`). In previous releases, the default standard math library for C++ was the same as the default standard math library for C.

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.



myadd.m

a	Click to define
b	Click to define

Global variables:

g	Click to define

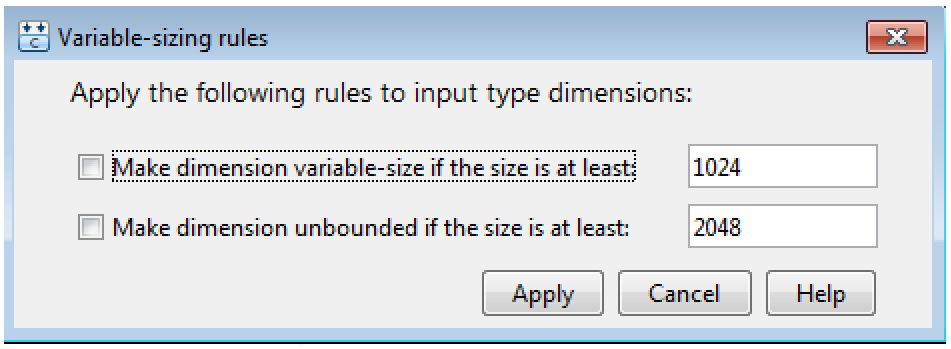
Add global

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.

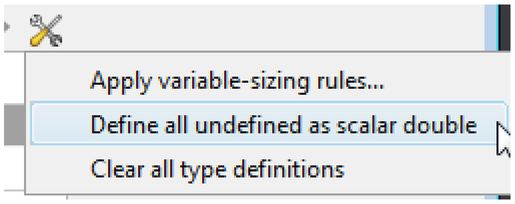
  Variable-sizing rules

  Apply the following rules to input type dimensions:

☐ Make dimension variable-size if the size is at least:	1024
☐ Make dimension unbounded if the size is at least:	2048

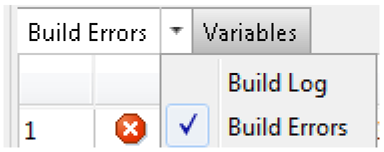
These rules apply to all current type definitions. If you change type definitions, the rules do not affect the new definitions unless you apply them. See Make Dimensions Variable-Size When They Meet Size Threshold.

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

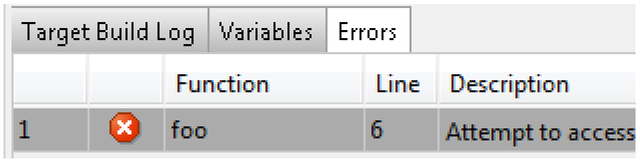

Apply variable-sizing rules...
Define all undefined as scalar double
Clear all type definitions

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.



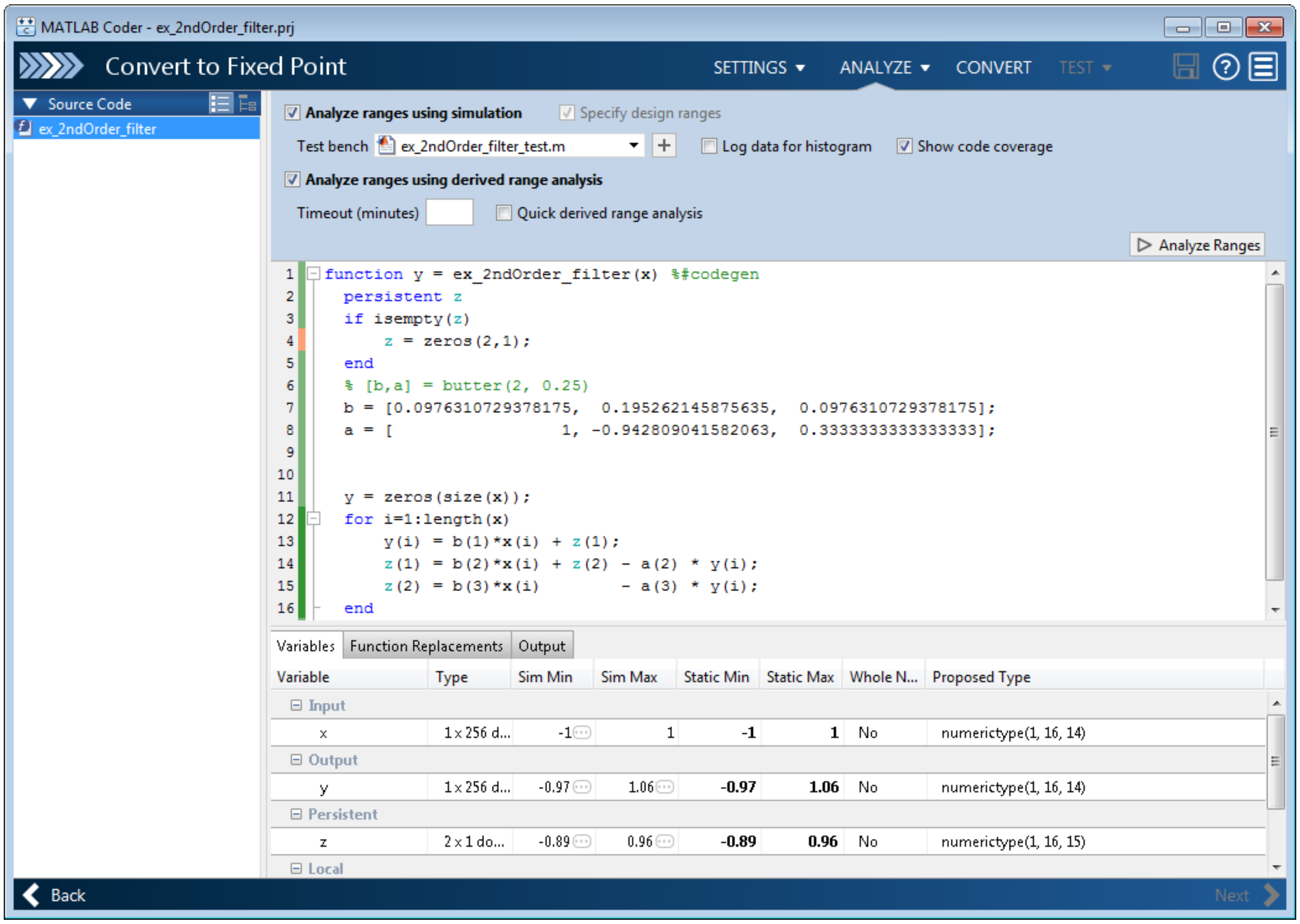
In R2016b, the **Build Errors** tab is named the **Errors** tab, and the **Build Log** tab is named the **Target Build Log** tab. These tabs are separate so that you can more easily find them.



		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.



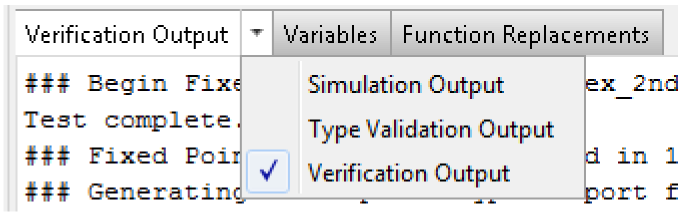
☑ Analyze ranges using simulation ☑ Specify design ranges
Test bench ex_2ndOrder_filter_test.m ☐ Log data for histogram ☑ Show code coverage
☑ Analyze ranges using derived range analysis
Timeout (minutes) ☐ Quick derived range analysis

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  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							

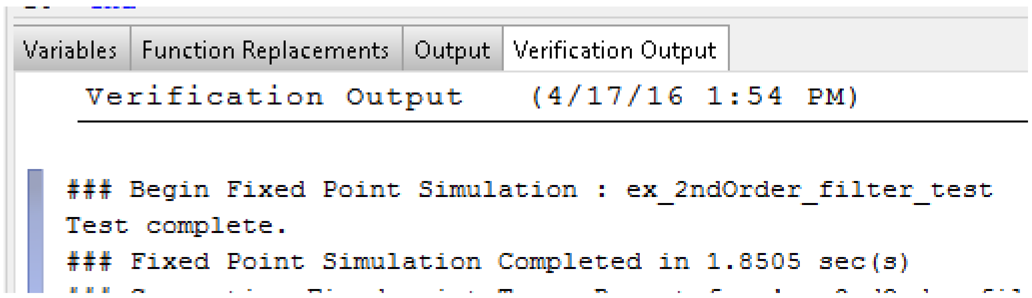
More discoverable logs and reports for fixed-point conversion in MATLAB Coder app

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.



```
### Begin Fixe                ex_2nd
Test complete.
### Fixed Poi                 d in 1
### Generating               port f
```

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.



```
Verification Output   (4/17/16 1:54 PM)

### Begin Fixed Point Simulation : ex_2ndOrder_filter_test
Test complete.
### Fixed Point Simulation Completed in 1.8505 sec(s)
```

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`.

Performance

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`.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

MATLAB Programming for Code Generation

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.

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(~)
```

Supported Functions

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.

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.

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

- `interpn`

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 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.

Code Generation Workflow

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

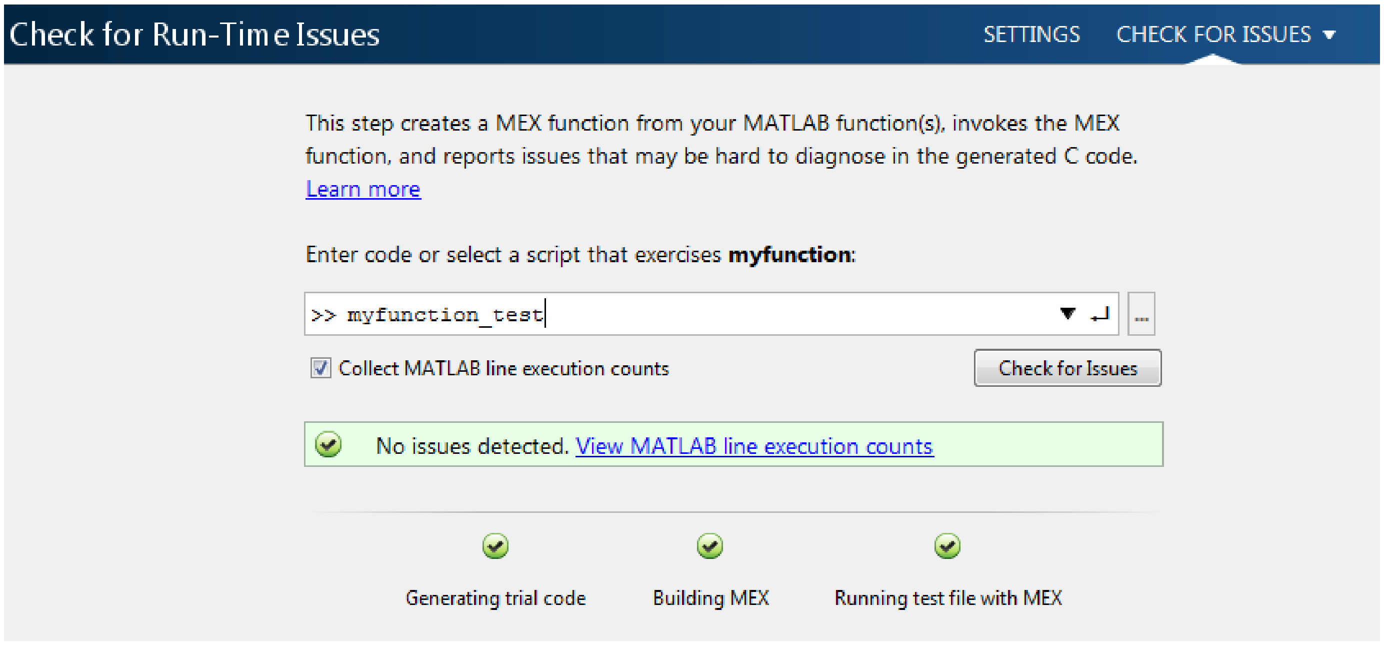
Starting with R2016a, MATLAB Coder is available for purchase as an add-on product for student-use software: MATLAB Student™ and MATLAB and Simulink Student Suite™. Student-use software provides the same tools that professional engineers and scientists use. Students use the software to develop skills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

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
Learn more
Collect MATLAB line execution counts
Check for Issues
View MATLAB line execution counts
Generating trial code
Building MEX
Running test file with MEX

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.

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

```
>> myfunction_test
```

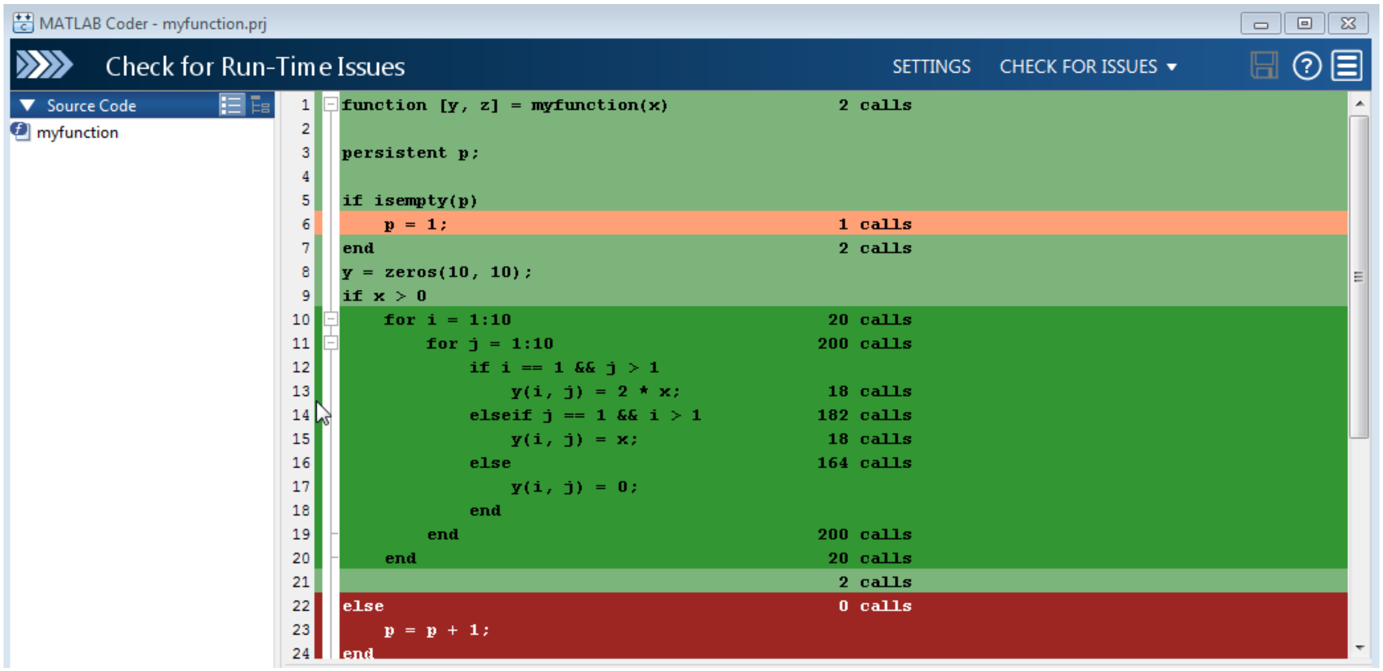
☑ Collect MATLAB line execution counts

No issues detected. View MATLAB line execution counts

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	One of the following situations: • 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. Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.
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.



```
function [y, z] = myfunction(x)                 2 calls

persistent p;

if isempty(p)
    p = 1;                                      1 calls
end                                             2 calls
y = zeros(10, 10);
if x > 0
    for i = 1:10                               20 calls
        for j = 1:10                          200 calls
            if i == 1 && j > 1
                y(i, j) = 2 * x;               18 calls
            elseif j == 1 && i > 1            182 calls
                y(i, j) = x;                   18 calls
            else                              164 calls
                y(i, j) = 0;
            end
        end                                   200 calls
    end                                        20 calls
                                                2 calls
else                                            0 calls
    p = p + 1;
end
```

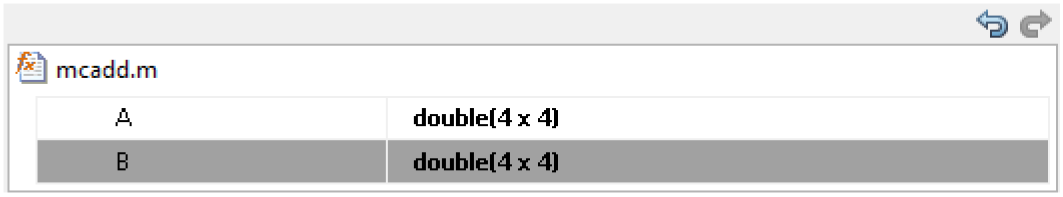
Line execution count collection is enabled by default. To disable the collection, clear the **Collect MATLAB line execution counts** check box. If line execution collection slows the run-time issue checking, consider disabling it.

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.

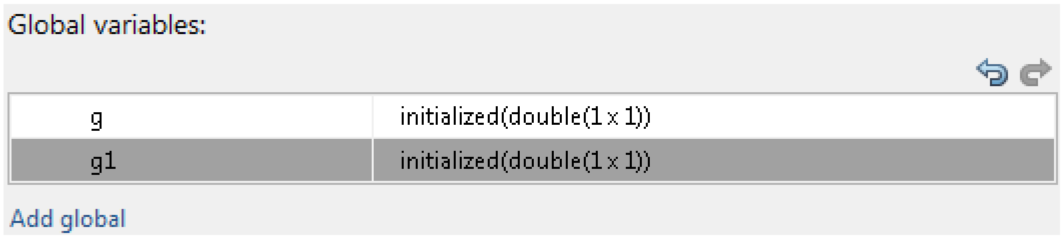
To revert or restore changes to input argument type definitions, above the input arguments table, click ↶ or ↷.



mcadd.m

A	double(4 x 4)
B	double(4 x 4)

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



Global variables:

g	initialized(double(1 x 1))
g1	initialized(double(1 x 1))

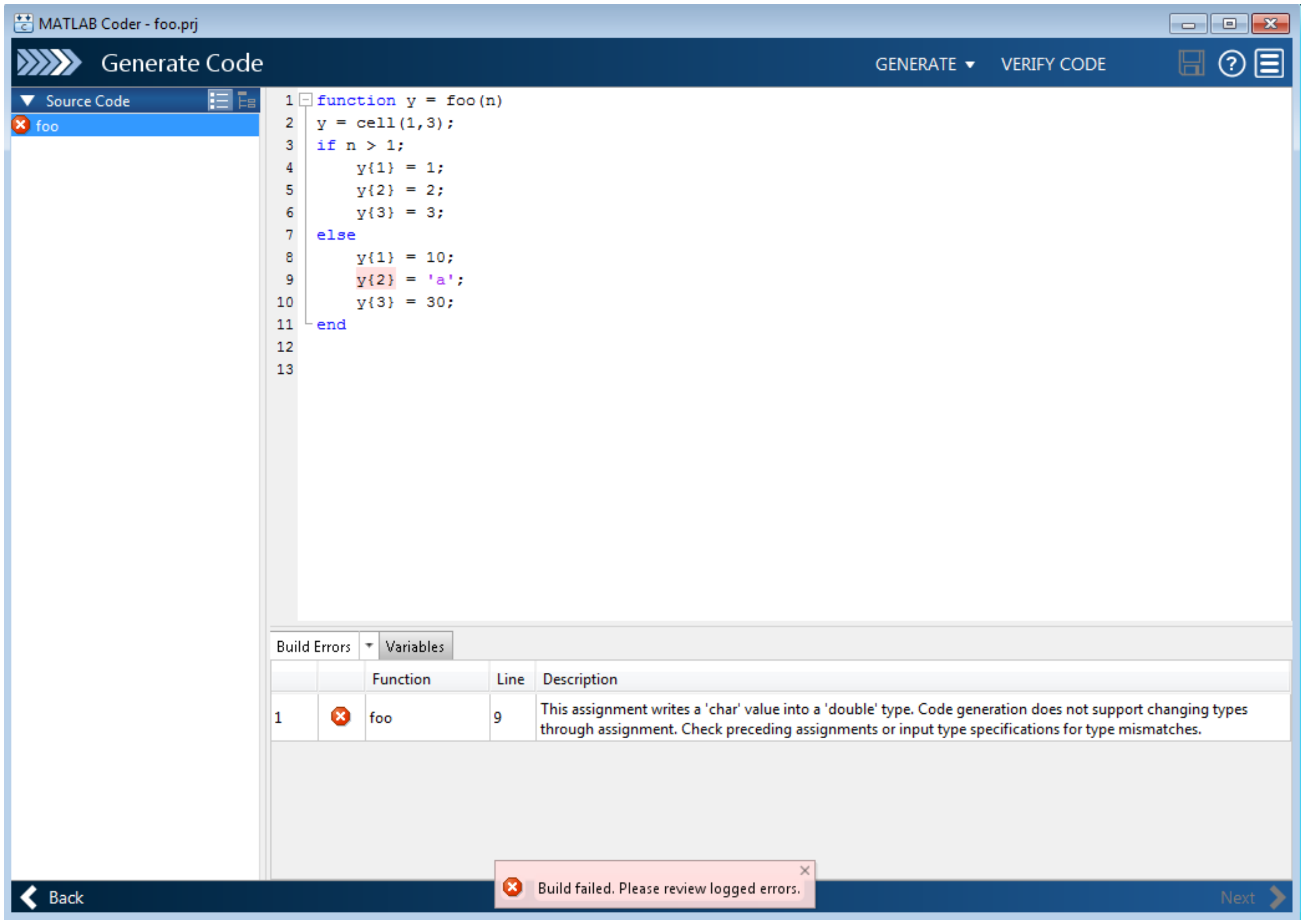
Alternatively, use the keyboard shortcuts for Undo and Redo. The keyboard shortcuts apply to the table that is selected. The shortcuts are defined in your MATLAB preferences. On a Windows platform, the default keyboard shortcuts for Undo and Redo are **Ctrl+Z** and **Ctrl+Y**.

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.



```
function y = foo(n)
y = cell(1,3);
if n > 1;
    y{1} = 1;
    y{2} = 2;
    y{3} = 3;
else
    y{1} = 10;
    y{2} = 'a';
    y{3} = 30;
end
```

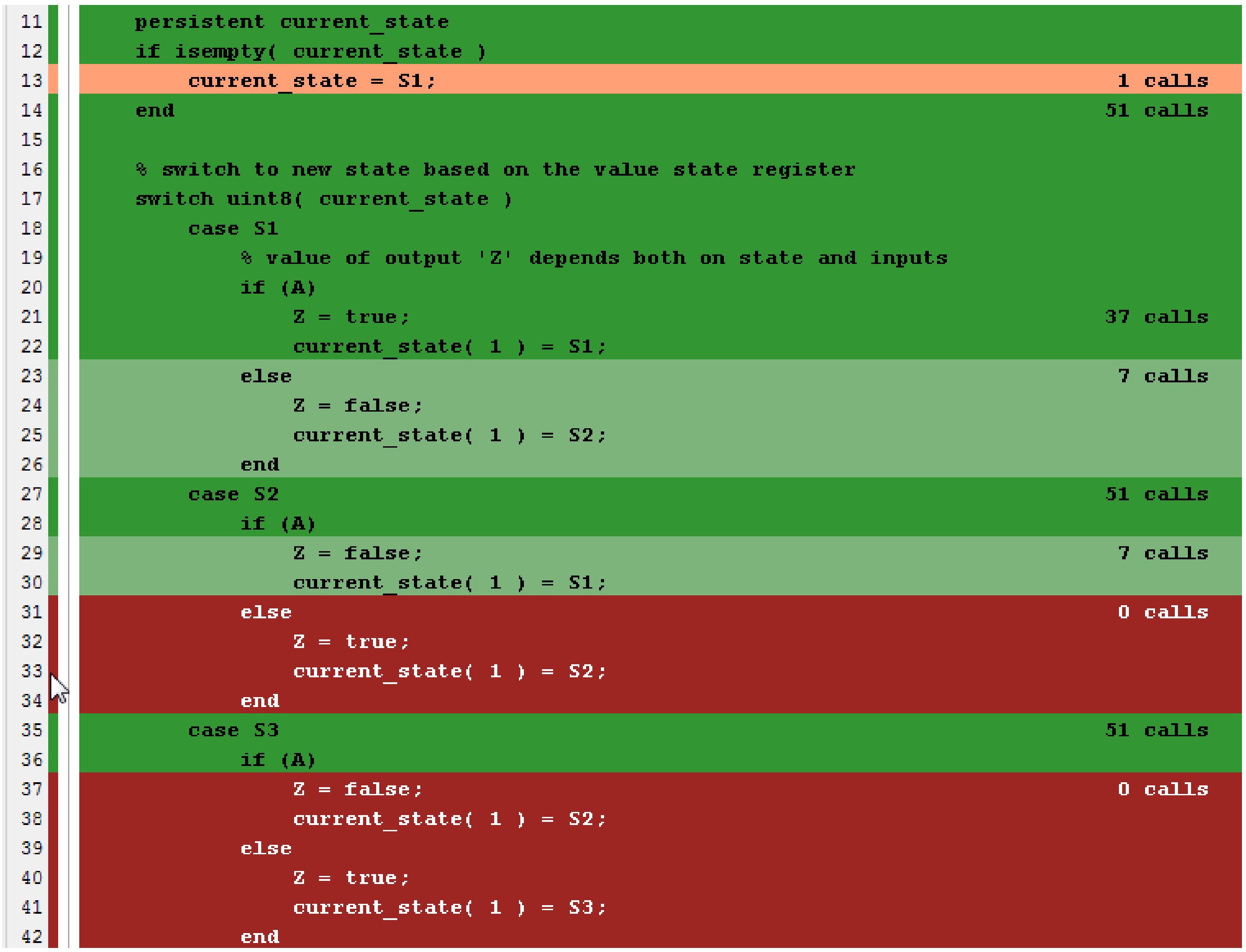
		Function	Line	Description
1	❌	foo	9	This assignment writes a 'char' value into a 'double' type. Code generation does not support changing types through assignment. Check preceding assignments or input type specifications for type mismatches.

Build failed. Please review logged errors.

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.



```
    persistent current_state
    if isempty( current_state )
        current_state = S1;                                    1 calls
    end                                                       51 calls

    % switch to new state based on the value state register
    switch uint8( current_state )
        case S1
            % value of output 'Z' depends both on state and inputs
            if (A)
                Z = true;                                     37 calls
                current_state( 1 ) = S1;
            else                                               7 calls
                Z = false;
                current_state( 1 ) = S2;
            end
        case S2                                               51 calls
            if (A)
                Z = false;                                     7 calls
                current_state( 1 ) = S1;
            else                                               0 calls
                Z = true;
                current_state( 1 ) = S2;
            end
        case S3                                               51 calls
            if (A)
                Z = false;                                     0 calls
                current_state( 1 ) = S2;
            else
                Z = true;
                current_state( 1 ) = S3;
            end
```

See Code Coverage in Automated Fixed-Point Conversion.

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

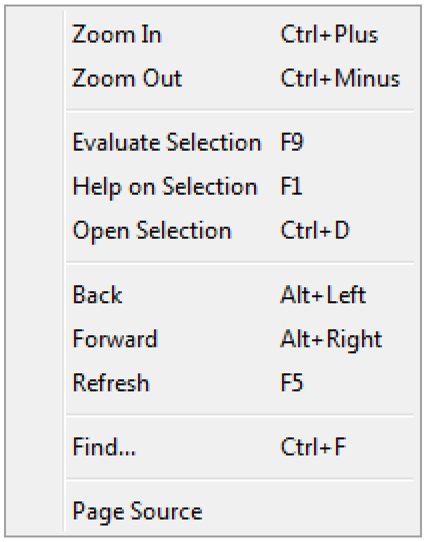
More Keyboard Shortcuts in Code Generation Report: Navigate the report more easily

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**

Action	Default Keyboard Shortcut for a Windows Platform
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.



See Define Keyboard Shortcuts and Code Generation Reports.

Performance

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 LAPACKE 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.

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

`for (i = 0; i < 100; i++) {`
`    Y1[i] = 0.0;`
`    Y2[i] = 0.0;`
`    Y3[i] = 0.0;`
` }`
	`memset(&Y1[0],0,100U*sizeof(double));`
	`memset(&Y2[0],0,100U*sizeof(double ));`
	`memset(&Y3[0],0,100U*sizeof(double ));`

Consecutive statements that define a continuous write.

Previous Releases	R2016a

`Y1[0] = 255;`
`Y1[1] = 255;`
`Y1[2] = 255;`
`...`
`Y1[99] = 255`
	`memset(&Y1[0], 255, 100U * sizeof(unsigned char));`

A structure that contains an array.

Previous Releases	R2016a

`for (i = 0; i < 100; i++) {`
`    S->f1[i] = 0.0;`
	`memset(&S>f1[0], 0, 100U * sizeof(double));`

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

Previous Releases	R2016a

`for (i = 0; i < 100; i++) {`
`    S[i].f1 = 255;`
`    S[i].f2 = 255;`
`    S[i].f3 = 255;`
`}`
	`memset(&S[0], 255, 100U * sizeof(struct0_T));`

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

`if (cond) {`
`    y = true;`
`  } else {`
`    y = val;`
`  }`
 	
`  return y;`
	`return cond || val;`

`y = x && !x;`	`y = false;`

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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 `https://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

bwareaopen	houghpeaks	immse	integralBoxFilter
grayconnected	imabsdiff	imresize	psnr

`hough`	`imcrop`	`imrotate`	
`houghlines`	`imgaborfilt`	`imtranslate`	

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

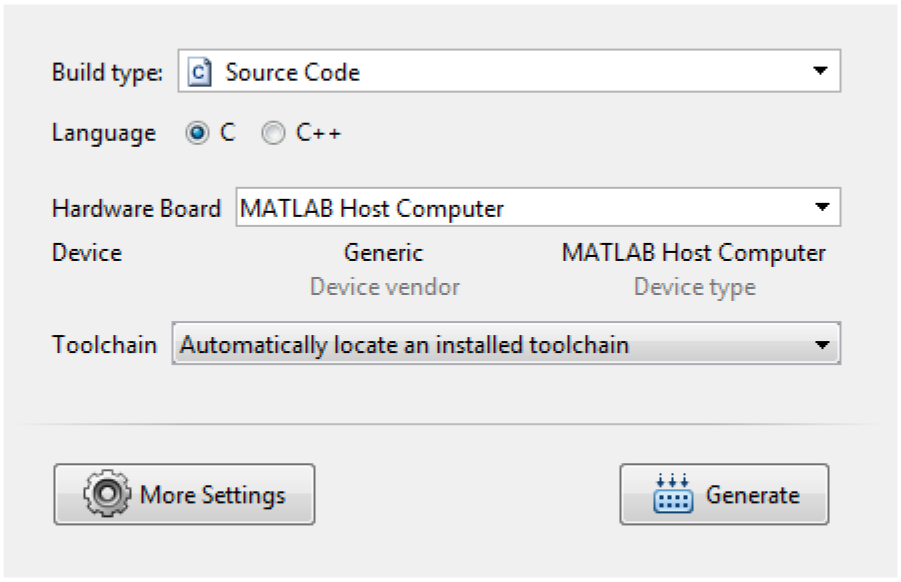
- `kmeans`
- `randsample`

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: Source Code

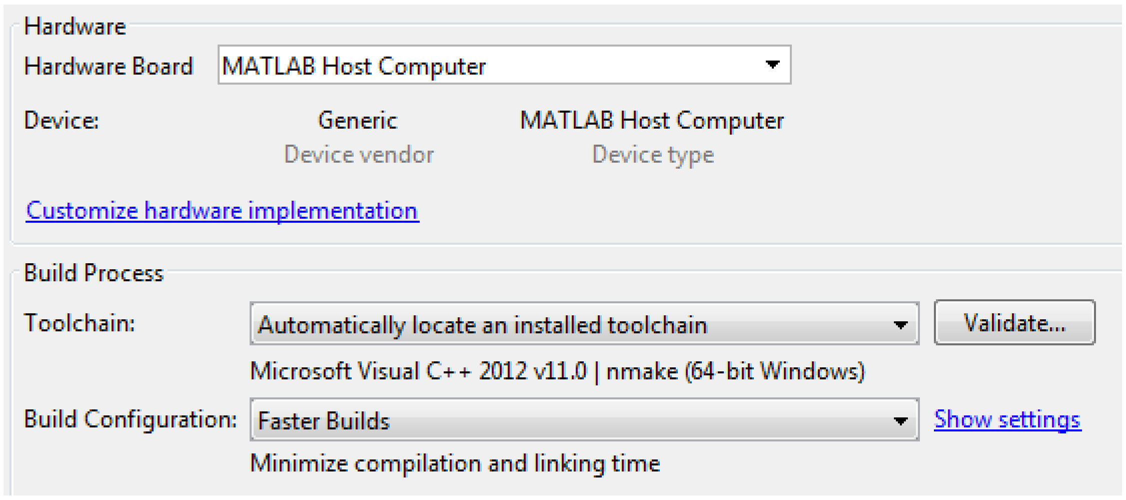
Language [✓ C] C++

Hardware Board: MATLAB Host Computer

Device: Generic (Device vendor) | MATLAB Host Computer (Device type)

Toolchain: Automatically locate an installed toolchain

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


Hardware

Hardware Board: MATLAB Host Computer

Device: Generic (Device vendor) | MATLAB Host Computer (Device type)

Customize hardware implementation

Build Process

Toolchain: Automatically locate an installed toolchain

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

Build Configuration: Faster Builds

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

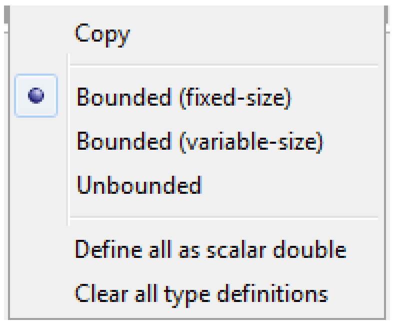
  For information about using hardware support packages with MATLAB Coder, see the Embedded Coder release notes.
- On the **Hardware** tab, the app hides the hardware implementation details. To see or modify the hardware implementation details, click **Customize hardware implementation**. By default, the 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

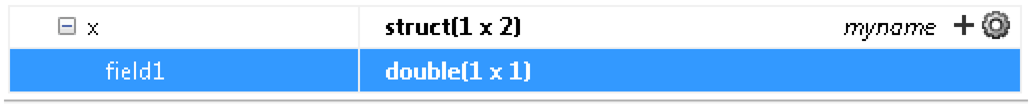
Improvements for manual type definition

Improvements for manual type definition include:

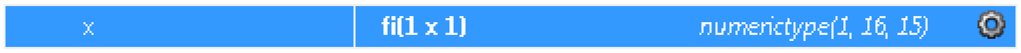
- Context menu options to specify array size.



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



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



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

Tab completion for specifying files

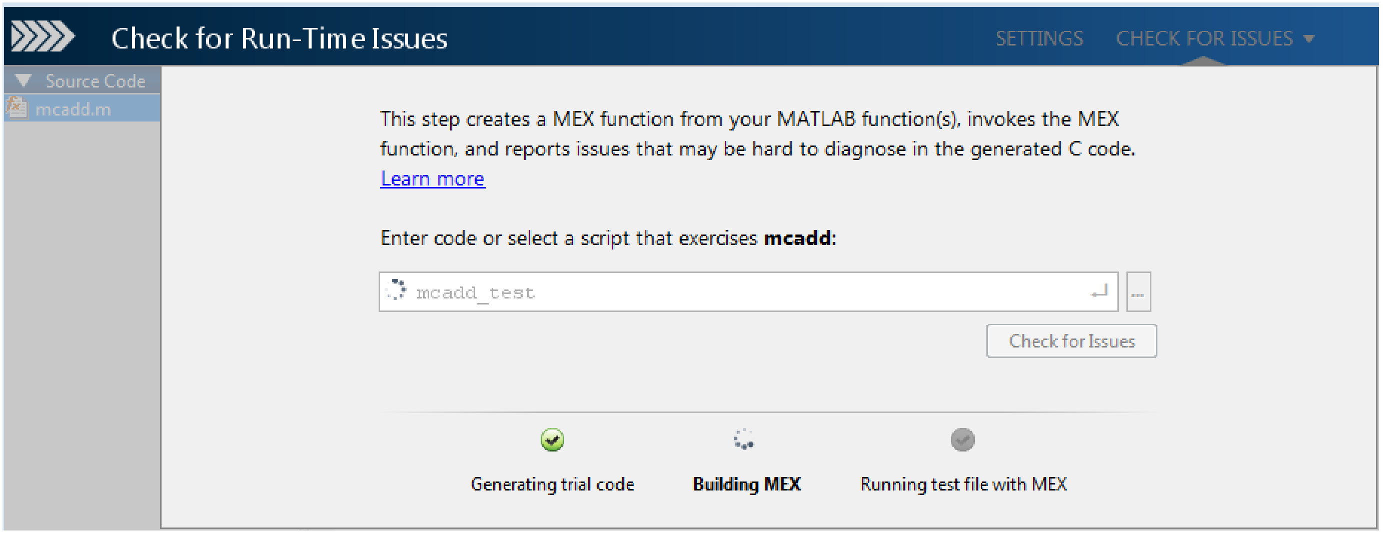
You can use tab completion to specify entry-point functions and test files.

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.



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.

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

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

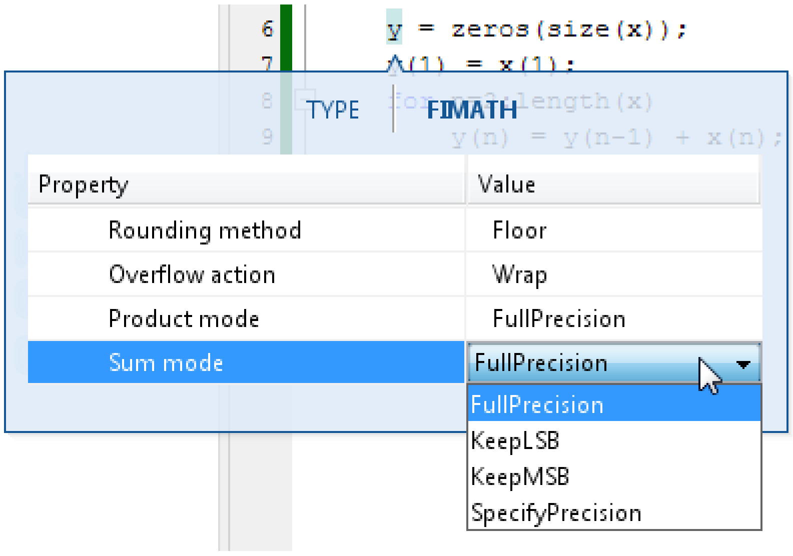
If you close a project before completing the fixed-point conversion process, the app saves your work. When you reopen the project, the app restores the state. You do not have to repeat the fixed-point conversion steps that you completed in a previous session. For example, suppose that you close the project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



```
y = zeros(size(x));
y(1) = x(1);
for n=2:length(x)
    y(n) = y(n-1) + x(n);
```

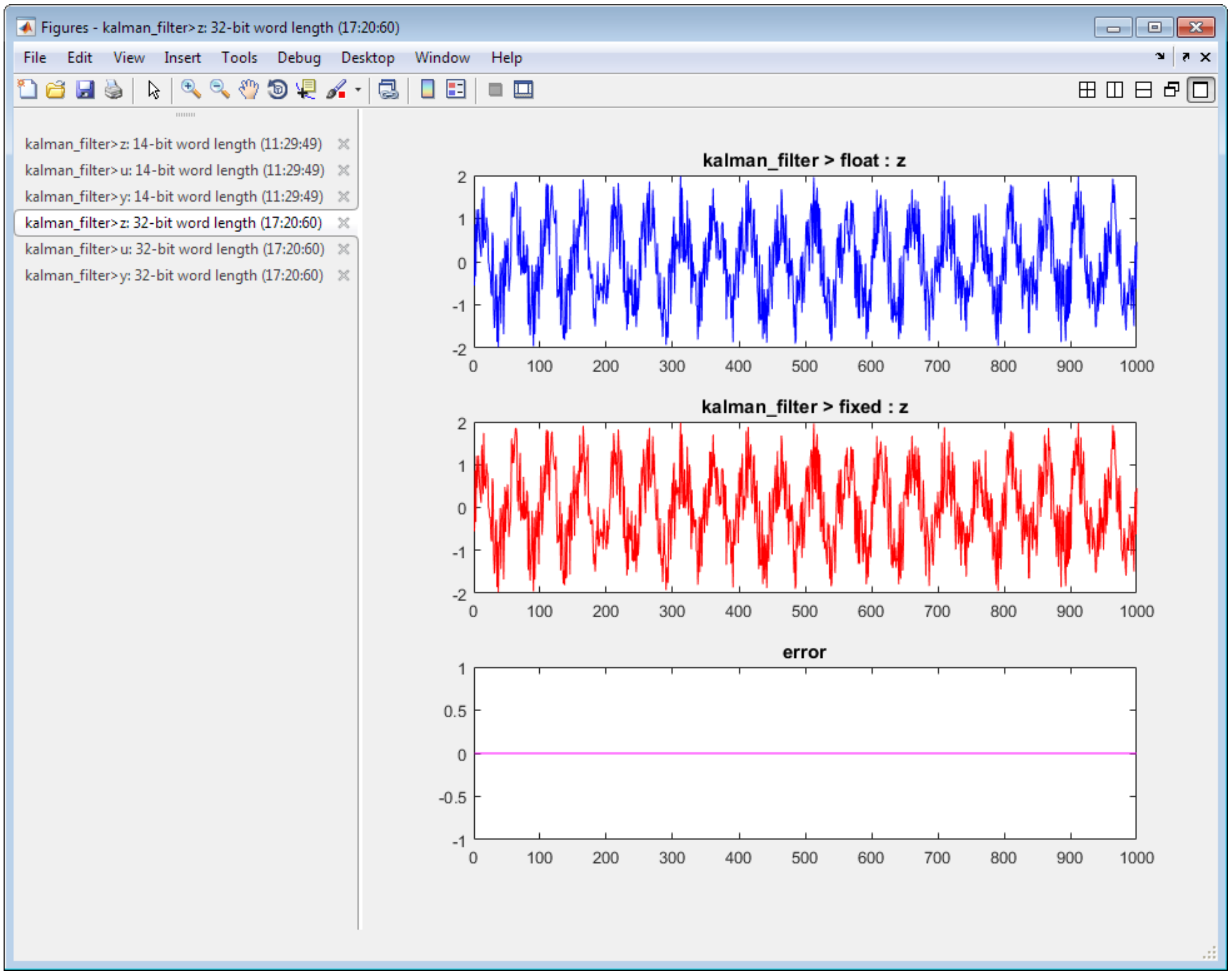
Property	Value
Rounding method	Floor
Overflow action	Wrap
Product mode	FullPrecision
Sum mode	FullPrecision

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

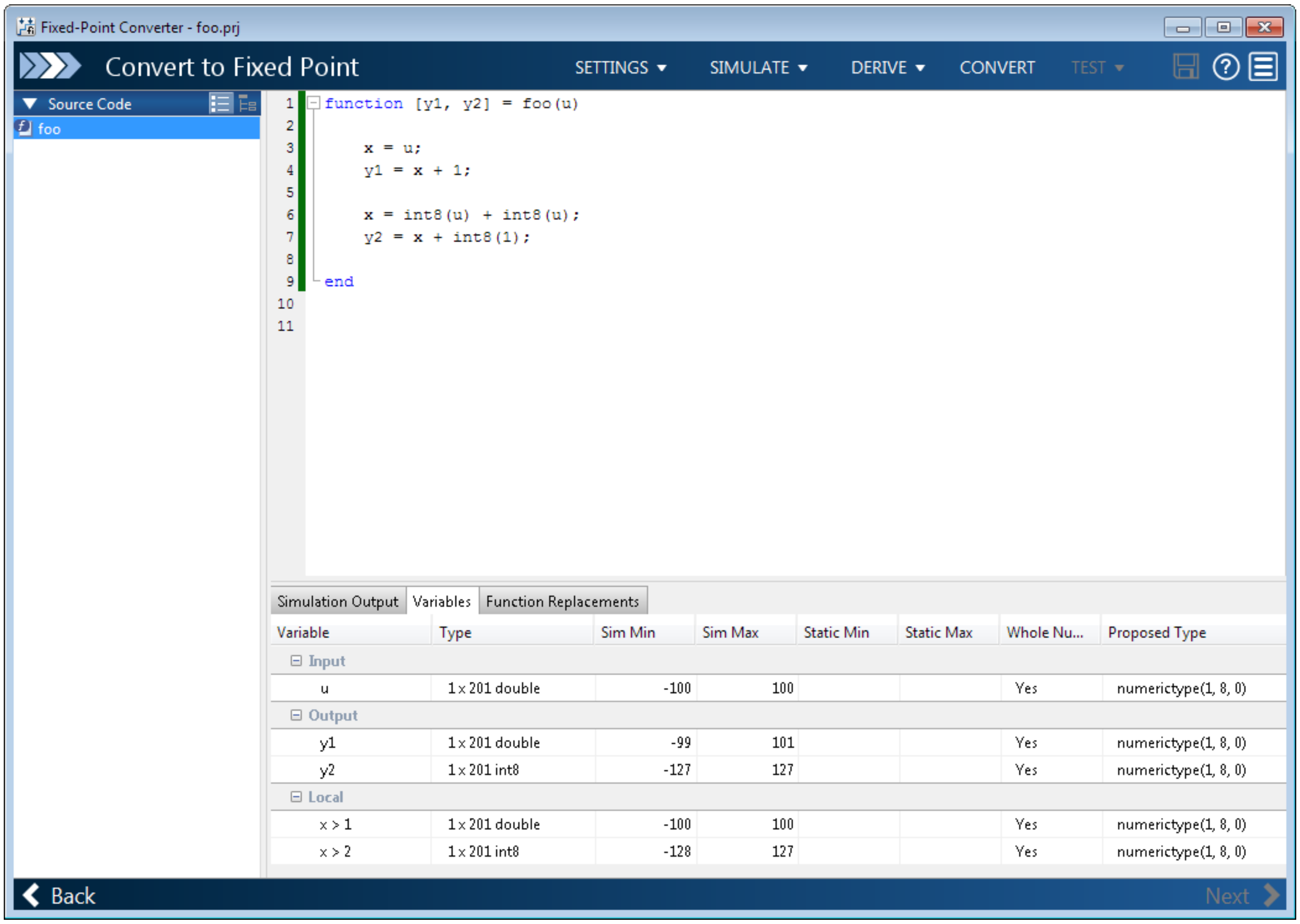
During fixed-point conversion, the app docks plots that are generated during the testing phase of your fixed-point code into separate tabs of one figure window. Each tabbed figure represents one input or output variable and is labeled with the function, variable, word length, and a timestamp. Each tab 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.


Figures - kalman_filter>z: 32-bit word length (17:20:60)
File Edit View Insert Tools Debug Desktop Window Help
kalman_filter>z: 14-bit word length (11:29:49)
kalman_filter>u: 14-bit word length (11:29:49)
kalman_filter>y: 14-bit word length (11:29:49)
kalman_filter>z: 32-bit word length (17:20:60)
kalman_filter>u: 32-bit word length (17:20:60)
kalman_filter>y: 32-bit word length (17:20:60)
kalman_filter > float : z
kalman_filter > fixed : z
error

Variable specializations

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



```
function [y1, y2] = foo(u)

    x = u;
    y1 = x + 1;

    x = int8(u) + int8(u);
    y2 = x + int8(1);

end
```

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

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

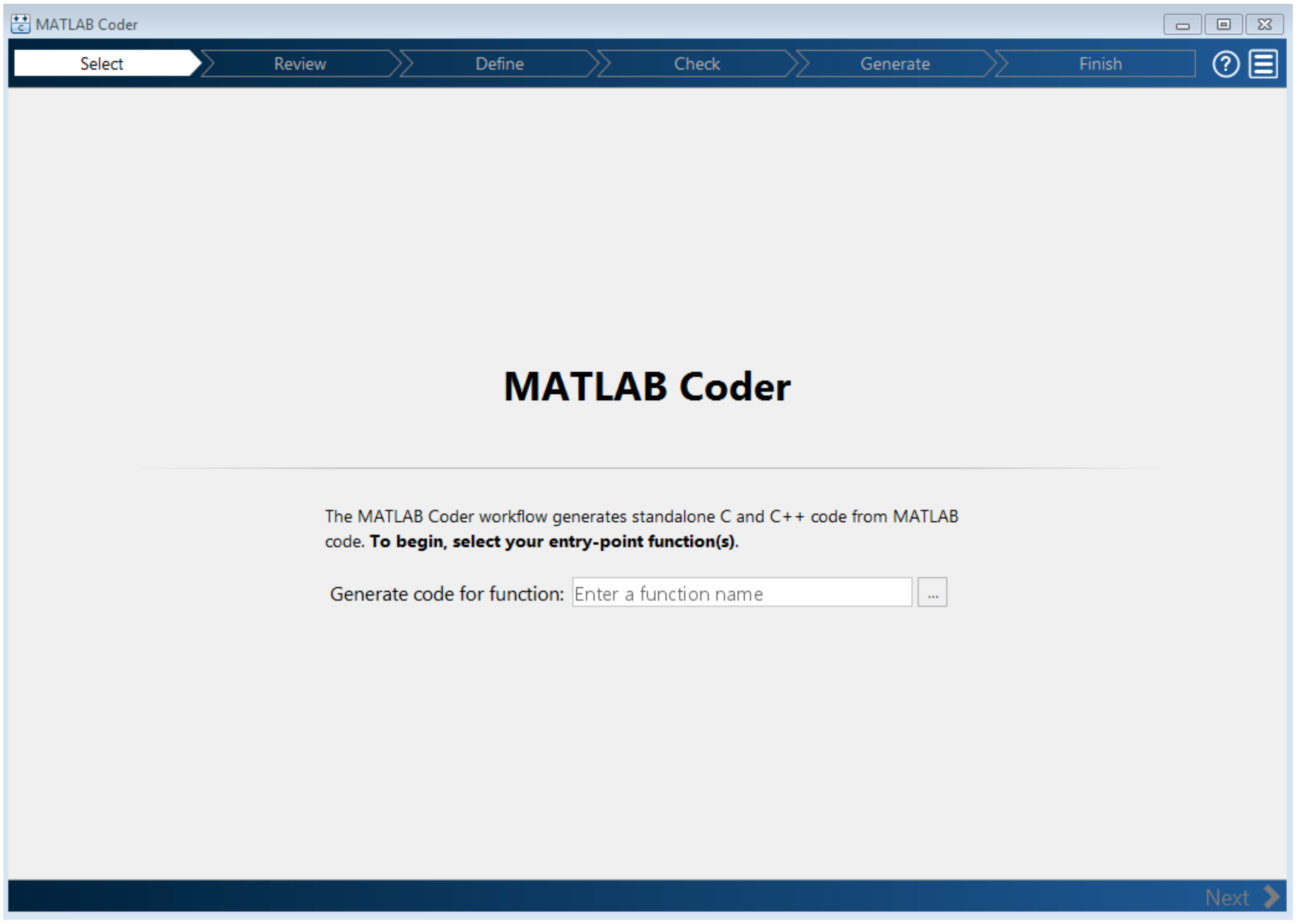
The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

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

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.



MATLAB Coder

The MATLAB Coder workflow generates standalone C and C++ code from MATLAB code. **To begin, select your entry-point function(s).**

Generate code for function: Enter a function name

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 Computer Vision System 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 `emxArray`s that are outputs or `emxArray`s 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 `emxArray`s 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 `emxArray`s. 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 `emxArray`s 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 `emxArray`s 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.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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

Image Processing Toolbox

`bwdist`	`imadjust`	`intlut`	`ordfilt2`
`bwtraceboundary`	`imclearborder`	`iptcheckmap`	`rgb2ycbcr`
`fitgeotrans`	`imlincomb`	`medfilt2`	`stretchlim`
`histeq`	`imquantize`	`multithresh`	`ycbcr2rgb`

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.

- `myscript`*`suffix`*`.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`.

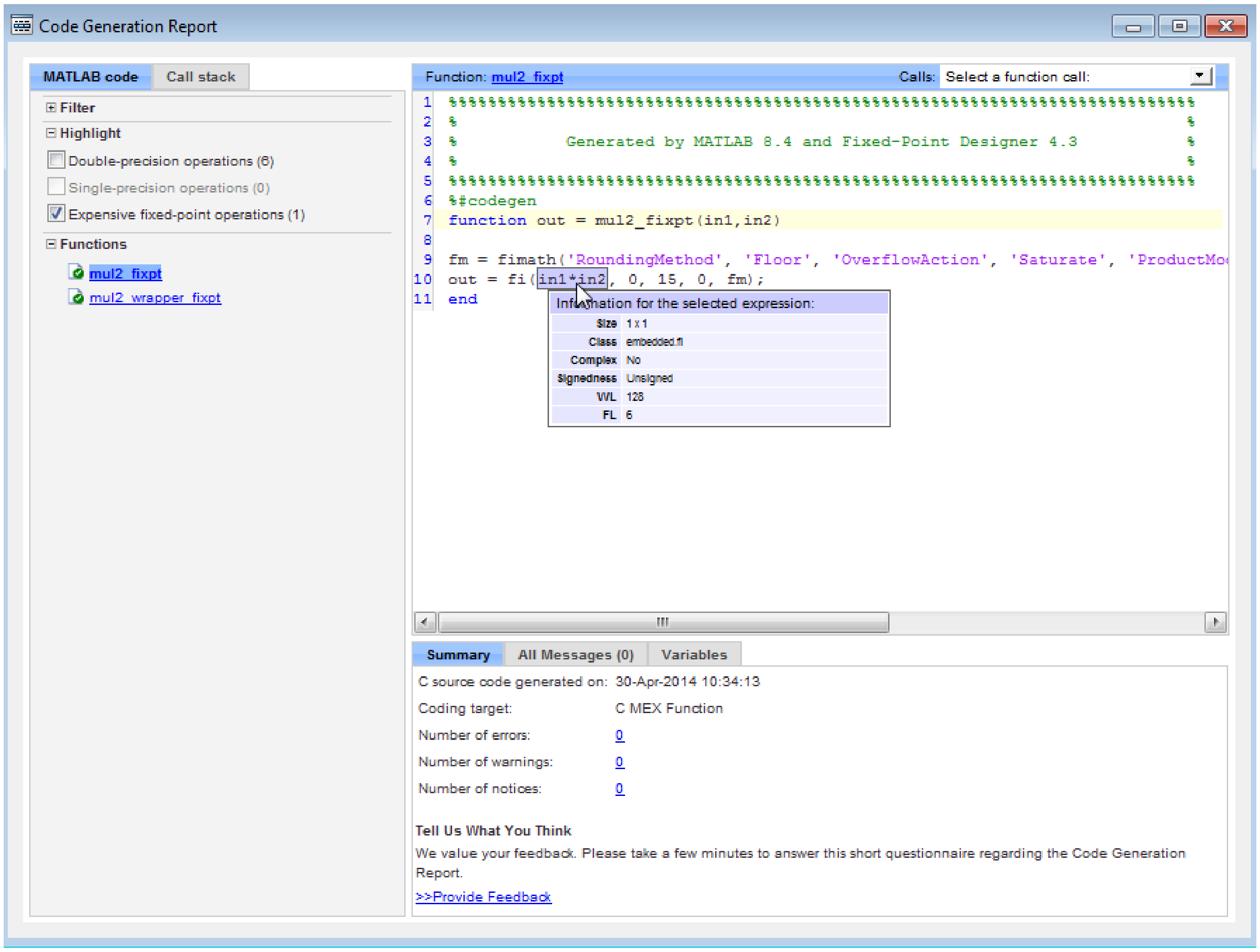
  If you concatenate matrices, the conversion tool uses the largest numerictype among the expressions of a row and casts the leftmost element to that type. This type is then used for the concatenated matrix to avoid loss of range.
- If the function that you are converting has a scalar input, and the `mpower` exponent input is not constant, the conversion tool sets `fimath` `ProductMode` to `SpecifyPrecision` 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.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%           Generated by MATLAB 8.4 and Fixed-Point Designer 4.3            %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function out = mul2_fixpt(in1,in2)

fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Saturate', 'ProductMo
out = fi(in1*in2, 0, 15, 0, fm);
end
```

Information for the selected expression:

Size	1 x 1
Class	embedded.fi
Complex	No
Signedness	Unsigned
WL	128
FL	6

C source code generated on:	30-Apr-2014 10:34:13
Coding target:	C MEX Function
Number of errors:	0
Number of warnings:	0
Number of notices:	0

Tell Us What You Think

We value your feedback. Please take a few minutes to answer this short questionnaire regarding the Code Generation Report.

>>Provide Feedback

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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

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`

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

`ca2tf`	`firhalfband`	`ifir`	`iirnotch`

`cl2tf`	`firlpnorm`	`iircomb`	`iirpeak`
`firceqrip`	`firminphase`	`iirgrpdelay`	`tf2ca`
`fireqint`	`firnyquist`	`iirlpnorm`	`tf2cl`
`firgr`	`firpr2chfb`	`iirlpnormc`	`dsp.DCBlocker`

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 (`TargetLangStandard`)
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 informations, 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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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`
- `MSERRegions`
- `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 `https://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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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);
  ...
```

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 `https://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?
`emlc`	Errors in R2013a.
`emlmex`	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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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`

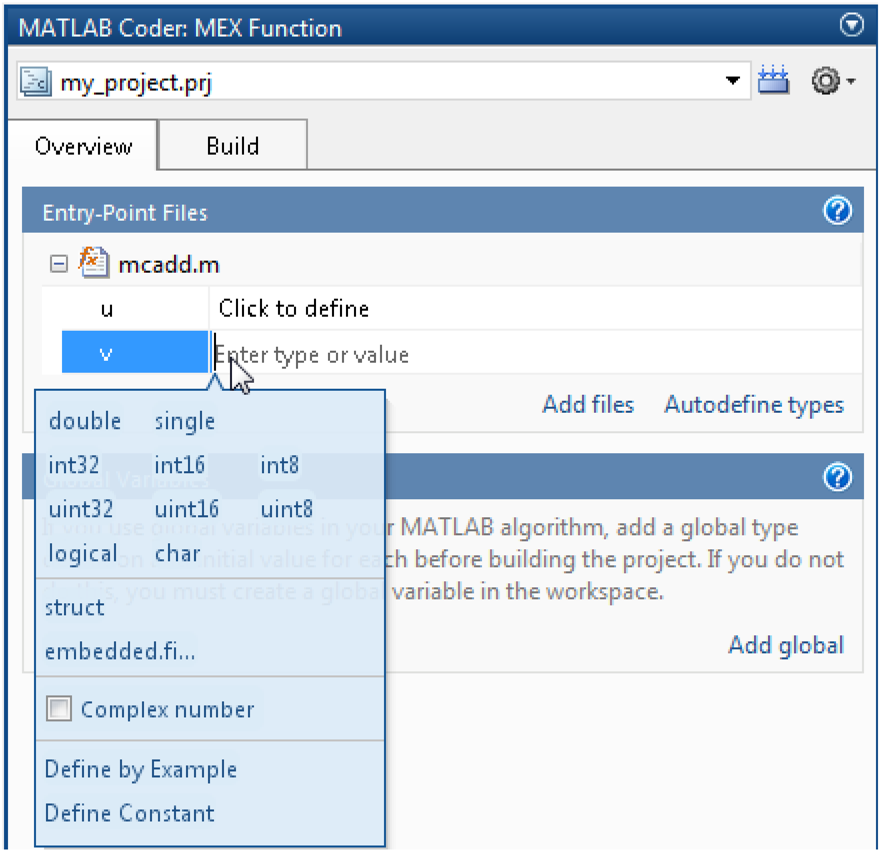
Visualization functions in generated MEX functions

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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...**
`coderdemo_reverb`	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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

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
`-eg`	`-args`
`emlcoder.egc`	`coder.Constant`
`emlcoder.egs`	`coder.typeof(a,b,1)` specifies a variable-size input with the same class and complexity as `a` and same size and upper bounds as the size vector `b`. Creates `coder.Type` objects for use with the `codegen -args` option. For more information, see `coder.typeof`.
`-F`	No`codegen` option available. Instead, use the default fimath. For more information, see the Fixed-Point Toolbox documentation.
`-global`	`-globals` **Note** `-global` continues to work with `codegen`

Old emlc Option	New codegen Option
`-N`	This option is no longer supported. Instead, set up `numerictype` in MATLAB.
`-s`	`-config` Use with the new configuration objects, see “New Code Generation Configuration Objects” on page 26-3.
`-T rtw:exe`	`-config:exe` Use this option to generate a C/C++ executable using default build options. Otherwise, use `-config` with a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` configuration object.
`-T mex`	`-config:mex` Use this option to generate a MEX function using default build options. Otherwise, use `-config` with a `coder.MexCodeConfig` configuration object.

`-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:
	• `RTWCompilerOptimization` is now `CCompilerOptimization`
	• `RTWCustomCompilerOptimization` is now `CCustomCompilerOptimization`
	• `RTWVerbose` is now `Verbose`

Old emlc Configuration Object	New codegen Configuration Object
emlcoder.RTWConfig('ert')	coder.EmbeddedCodeConfig The following property names have changed: • MultiInstanceERTCode is now MultiInstanceCode • RTWCompilerOptimization is now CCompilerOptimization • RTWCustomCompilerOptimization is now CCustomCompilerOptimization • RTWVerbose is now Verbose
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
`coder.config`	Create MATLAB Coder code generation configuration objects
`coder.newtype`	Create a new `coder.Type` object
`coder.resize`	Resize a `coder.Type` object
`coder.typeof`	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.

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 `codegen`
`Edge Detection on Images`	Generate a standalone C library from MATLAB code that implements a Sobel filter
`Read Text File`	Generate a standalone C library from MATLAB code that uses the `coder.ceval`, `coder.extrinsic` and `coder.opaque` 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`	• 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
`emlc`	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
`%#eml`	Still runs	`%#codegen`
`eml.allowpcode`	Still runs	`coder.allowpcode`
`eml.ceval`	Still runs	`coder.ceval`
`eml.cstructname`	Still runs	`coder.cstructname`
`eml.extrinsic`	Still runs	`coder.extrinsic`
`eml.inline`	Still runs	`coder.inline`
`eml.nullcopy`	Still runs	`coder.nullcopy`
`eml.opaque`	Still runs	`coder.opaque`
`eml.ref`	Still runs	`coder.ref`
`eml.rref`	Still runs	`coder.rref`
`eml.target`	Still runs	`coder.target`
`eml.unroll`	Still runs	`coder.unroll`
`eml.varsize`	Still runs	`coder.varsize`
`eml.wref`	Still runs	`coder.wref`

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.