Tips and Tricks for Image Processing and Computer Vision Code Generation

Image processing algorithms may require some special consideration when used with MATLAB Coder and computer vision algorithms. Image algorithms tend to be bound by data bandwidth (as opposed to computational bandwidth). In addition, data structures with the generated code may not match those in your existing code base.

This document is designed to offer simple but effective MATLAB Coder tips and tricks that address your specific goals for image processing algorithms. It is intended to complement the Quick Start Guide for MATLAB Coder, which includes:

- MATLAB Coder Tips and Tricks
- Preparing MATLAB Code for MATLAB Coder
- Generating C Code with MATLAB Coder
- Accelerating MATLAB Code with MATLAB Coder

Table of Contents

Common Problems

Before You Begin

Tips and Tricks

- Improving Performance
- Unit Test Framework with MATLAB Coder
- MATLAB Visualization from Visual Studio and Eclipse
- Converting nested functions into sub functions
- Passing Structures by Reference

Appendix A: Image Processing Toolbox Code Generation Details

Appendix B: Suggested Function Replacements for Unsupported Functions

Common Problems

Data Alignment

MATLAB structures its matrices (and thus, its images) in a column major format. Most external image processing vendors tend to favor a row major format.

Example of MATLAB organized image/pixel data as **column major:**

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

This data is then not ordered properly when interfacing with row major functions:

1	4	7	10	13	2	5	8	11	14	3	6	9	12	15

Example of image / pixel data after transposing to **row major**:

1	2	3
4	5	6
7	8	9
10	11	12
13	14	15

The data is correctly ordered when interfacing to row major functions.

This is something users need to be aware of if they are:

- Integrating with external libraries (e.g., OpenCV)
- Interfacing with external hardware (e.g., CMOS image sensor)

When developing software, careful consideration is needed in order to minimize the need for unnecessary transposes and data copies.

Data Structures

When using variable sized arrays, MATLAB Coder may use data structures as a container for an image. Understanding these data structures as you integrate generated code with your existing code/libraries will help you develop more efficient algorithms.

```
typedef struct emxArray_real_T
{
    double *data;
    int *size;
    int allocatedSize;
    int numDimensions;
    boolean_T canFreeData;
} emxArray_real_T;
```

Color Representation

Color images can be represented in a variety of formats in MATLAB, but those formats may not agree with external code and or devices, again requiring special care. For instance, OpenCV uses a BGR color format, while MATLAB opts for RGB. Minimizing the interaction between the two libraries should increase performance.

Before You Begin

Determine Code Generation Goals

The first step is to determine your code generation goals. Three possible choices exist:

- **Create C code combined with optimized shared libraries for x86/x64 platforms** running supported operating systems (Windows, Linux, and OSX). The code and libraries are performance optimized to take advantage of technologies like Intel's IPP and TBB libraries. The library components are specific for Intel and AMD architectures and the supported OS.
- **Create Standalone C/C++ code** that is capable of being compiled on any processor or platform. This code is typically single treaded and is not tied to any specific architecture.
- **Create and Compile MEX** (MATLAB executables) for acceleration in MATLAB simulations.

Prepare MATLAB Code for Code Generation

When working with image processing and computer vision algorithms for code generation, it is first necessary to prepare the MATLAB code. Here are a few general purpose steps, but for a more complete description, see the document, Preparing MATLAB Code for MATLAB Coder.

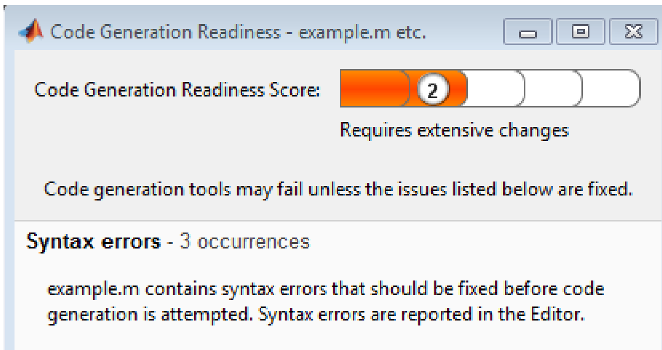
This section adds additional details specifically related to image processing and computer vision algorithms.

1. **Insert code generation pragma**

Add `%#codegen` comment in each MATLAB file that is meant for code generation to enable additional capabilities of the MATLAB Code Analyzer. Note: This comment can be place anywhere in the MATLAB file.

```
%#codegen
```

2. **Exercise code readiness tool**



3. **Verify functions for code generation**

   Verify MATLAB functions that support your code generation goal. (View full list of functions.) See "Remarks and Limitations" column for details.

Name	Product	Remarks and Limitations
imfilter	Image Processing Toolbox	The input image can be either 2-D or 3-D. The value of the input argument, `options`, must be a compile-time constant. Generated code for this function uses a precompiled platform-specific shared library.

   Functions that support code generation can do one of the following:

   - Generate ANSI C *source* code (platform *independent*). Most functions do this.
   - Generate precompiled platform-specific *shared library* (platform-*dependent*).

   If you see "platform-specific shared library" in the supported function list, it means it can generate a precompiled platform-specific shared library.

4. **Verify Image Processing Toolbox functions** that support your code generation goal. See Appendix A for code generation details.

5. **Verify Computer Vision System Toolbox functions** that support your code generation goal. View list of functions. Make note of the "Remarks and Limitations" column for details.

6. **Unsupported functions** require rewriting MATLAB code or calling external C libraries.
    - See section, "Leveraging External Libraries and Custom C Code."
    - See Appendix B for a list of suggested function replacements.

Tips & Tricks

Improving Performance

There are several techniques for improving the performance of your generated code. The following are a few recommendations:

1. Turn off Dynamic Memory Allocation (MALLOC) and avoid functions that require MALLOC in performance critical areas if possible (i.e. loops). To disable dynamic memory allocation in the Project Settings box:
    - On the MATLAB Coder project **Build** tab, click **More settings**.
    - In the **Project Settings** dialog box **Memory** tab, under **Enable variable-sizing**, set **Dynamic memory allocation** to **Never**.

2. Enable parallel processing on multicore machines with OpenMP. If your target compiler supports OpenMP then use `parfor` to run parallel threads on a multicore machines.

3. Use Code Generation Metrics Report to gather statistics on the generated code. View more information.

3. Function Information [hide]

View function metrics in a call tree format or table format.Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: Call Tree | Table

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
moving_average		8	8	4	13	1
moving_average_initialize		44	0	1	4	1
moving_average_terminate		0	0	0	4	1
rtGetInf		38	34	36	43	5
rtGetInfF	rtGetInf	4	4	3	6	1
rtGetMinusInf		38	34	36	43	5
rtGetMinusInfF	rtGetMinusInf	4	4	3	6	1
rtGetNaN		44	34	36	43	5
rtGetNaNF	rtGetNaN	10	10	19	24	4
rtIsInf		0	0	1	4	2
rtIsInfF		0	0	1	4	2
rtIsNaN		0	0	5	8	2
rtIsNaNF		0	0	5	8	2
rt_InitInfAndNaN		44	0	7	10	1

For more information on improving performance, see Accelerating MATLAB Code with MATLAB Coder.

4. Integrate existing C code and libraries. Often you will have existing code that you would like to leverage in MATLAB with code generation. This section shows how to integrate external code into MATLAB and to generate code that uses your existing code.
   - To integrate existing code or to preserve modifications to generated code, use `coder.target.`
     - Target options include `'MATLAB'`, `'MEX'`, `'Sfun'`, `'Rtw'`, `'HDL'`, and `'Custom'`.
     - Here is an example of how to use `coder.target`

```
if coder.target('MATLAB')  % runs in MATLAB
   y = algo_in_MATLAB(x);
else % used by MATLAB Coder
    y = zeros(size(x));  % Pre-define output
    coder.ceval('algo_in_c',x,y);
end
```

     View more information.

   - To integrate external libraries with header files, use `coder.ExternalDependency`. This allows external libraries to be accessed for MATLAB simulations and to be included with code generation. Here is an example using `coder.ExternalDependency`:

```
function c = adder(a, b)
    coder.cinclude('adder.h');
    c = 0;
    c = coder.ceval('adder', a, b);
end
```

```
y = AdderAPI.adder(x1, x2);
```

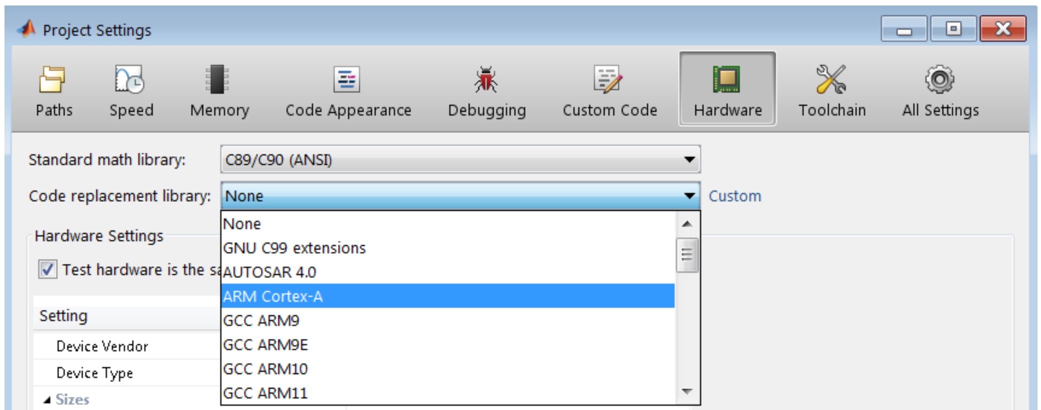
   Features of the `coder.ExternalDependency` class includes the following:

     - Easy to use as it uses `coder.ceval` to execute the custom code
     - Support for variable sized arrays
     - Most controls are on MATLAB side and less burden on external users
     - Creates build info with include paths, header file names, library names
     - Converts MATLAB variables to C types and calls C function API
     - Calls MATLAB function with `coder.ceval`
     - Converts MATLAB variable to C pointer using `coder.ref` or `coder.wref`
     - `Scalar` can be passed without any conversion

     View more information.

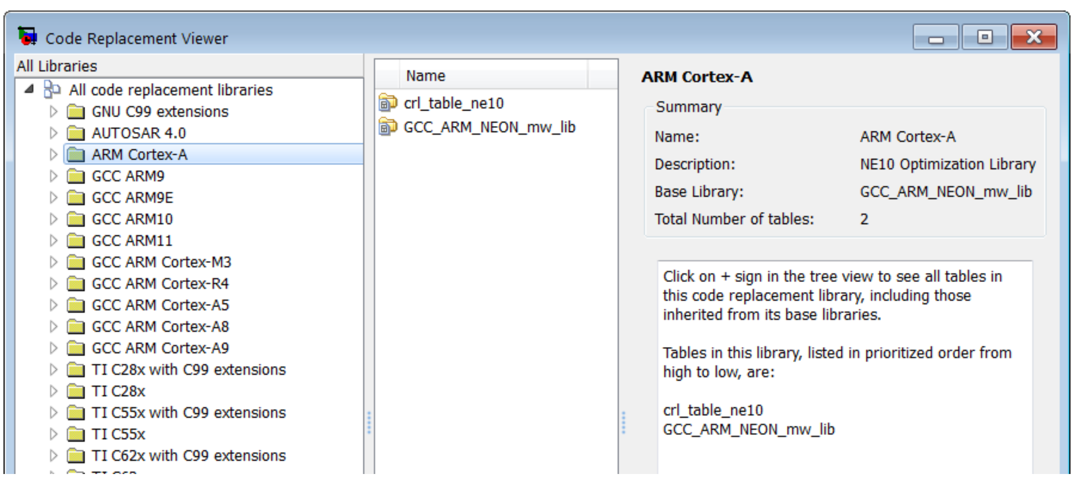
5. To take advantage of low level replacements of intrinsic operators, use Code Replacement Library (CRL). For a replacement to occur, the operator and data type arguments must match the table tale precisely. Here is the dialog tab showing the selection of some sample intrinsics.

   Note: CRL requires that the input types match and see `RTX.Tfl` file for more details. The CRL capability also requires a license for Embedded Coder.



   Note: Use the following command for a full list of library replacements:

```
> RTW.viewTfl
```



   Features of CRL include the following:

   - Replace low-level MATLAB implementations
   - Limitation: does not support variable size

- Needs dedicated table entry for library replacement (uses `coder.replace`)
- Requires Embedded Coder

CRL example:
When targeting the ARM Cortex-A and Cortex-M, the Ne10 and CMSIS Code Replacement Library (CRL) can be used to sSupport Ne10 (ARM Cortex-A):

```
ne10_add_float_neon()
ne10_sub_float_neon()
ne10_mul_float_neon()
ne10_divc_float_neon()
```

6. Interface with row-major code libraries. For image processing and computer vision (IPCV) applications, the fact that MATLAB is column-major while C/C++ code is row-major can cause issues. This can occur especially when dealing with code generation (pushing MATLAB algorithms to C code) or legacy code integration (bringing legacy C code into MATLAB). Here are two approaches to deal with this:
   - Modify the existing MATLAB algorithm in such a way as to process matrix data in a transposed manner.
   - At the input and output boundaries of the algorithms, transpose the input/output matrices.

Unit Test Framework with MATLAB Coder

Users can write an extensive set of test cases for a MATLAB function using the MATLAB Unit Test Framework, capturing expected behavior, edge cases, and exceptions thrown.

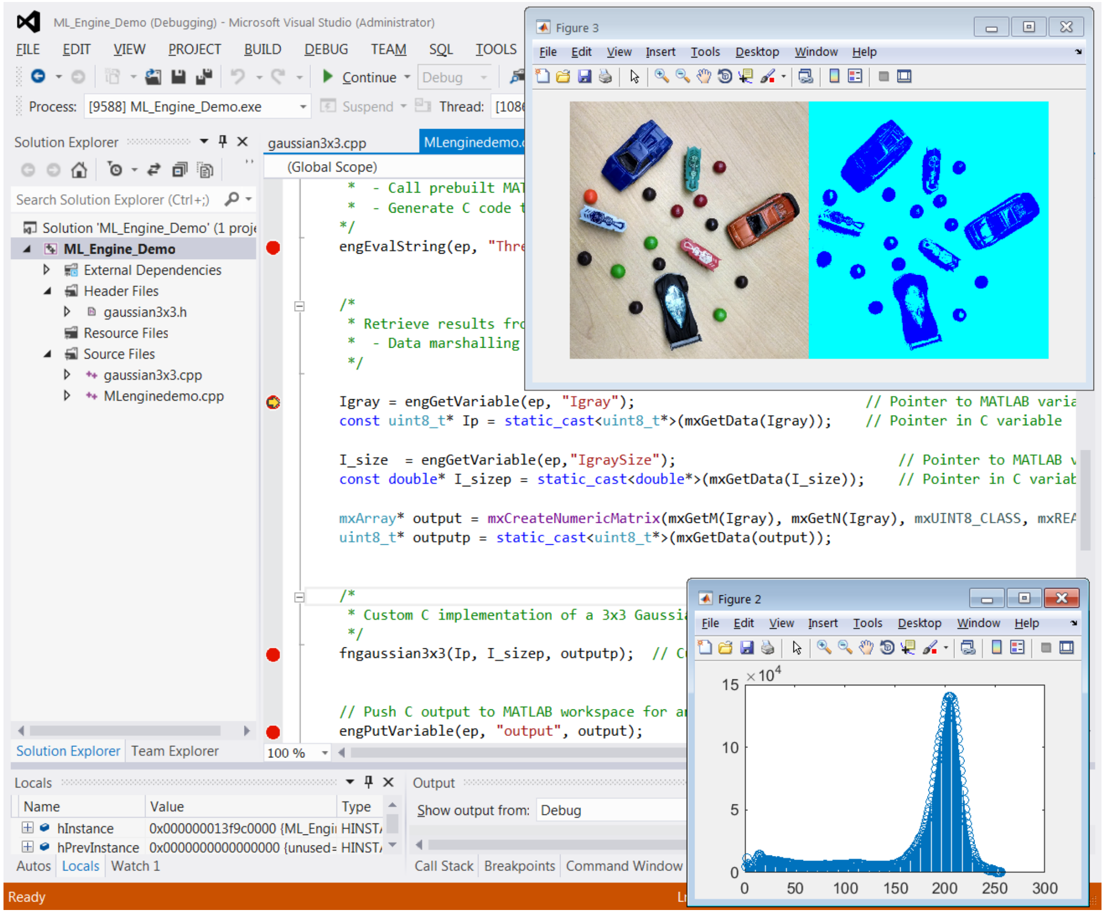
This capability can be combined when you generate code for the function using MATLAB Coder. Use the Unit Test Framework with MEX-files.

View more information.

Visualization and Verification from Visual Studio and Eclipse

When developing C/C++ applications in Visual Studio and Eclipse, it can be challenging to visualize and experiment with code changes, and test and verify results.

MATLAB Engine enables you to visualize, experiment, and test C/C++ code directly from Visual Studio and Eclipse by communicating with MATLAB. This connection enables you to access MATLAB plots, toolbox functions, and scripts directly from Visual Studio and Eclipse. With this connection, you can quickly explore and test results throughout the development phase to save time and effort.



For more information, see documentation for [MATLAB Engine API for C, C++, and Fortran]().

Converting Nested Functions into Sub-Functions

Nested functions are not yet supported in MATLAB Coder (as of R2014a). Here're a workaround to make nested functions codegen ready:

1. Move the nested function to its own sub-function
2. Pass any data used by the nested function as inputs to the new sub-function
3. Return any data needed by the main function back as an output of the sub-function

Passing Structures by Reference

Depending upon the MATLAB code and/or the configuration of MATLAB Coder, generated code will either pass by value or pass by reference. When a function has one or two simple scalar values, passing by value is usually more efficient. However, when function arguments are larger, passing by reference is more efficient.

MATLAB Coder will generate both idioms and the result will depend on when the MATLAB function includes:

- An input argument
- An output argument
- An input/output argument.

The code also differs for:

- MEX targets
- Standalone code

Here is the MATLAB code used to test each case.

Input only	Output only	Input and Output
`function y = strin(s)` `y = s.f;`	`function s = strout(x)` `s.f = x;`	`function [y,s] = strinout(x,s)` `y = x + sum(s.f);`

1. `PassStructByReference` option

Starting in R2013a, a "Pass Structure by Reference" option was introduced.

It only applied to input structures in R2013a, and was later extended to apply to output structures in R2013b.

This option is only applicable to standalone code. There is no option to alter the generated MEX code.

Note: The default from the user interface is `cfg.PassStructByReference=true` for R2013b and R2014a, but from the command line the default is `cfg.PassStructByReference=false`, as shown in the following tables by "default prj" and "default cfg" respectively.

2. Structure is an input

```
function y = strin(s)
y = s.f;
```

Stand-alone C code:

	cfg.PassStructByReference=false
R2013a (default)	void strin(const struct_T s, real_T y[4])
R2013b (default cfg)	void strin(const struct_T s, double y[4])
R2014a (default cfg)	void strin(const struct0_T s, double y[4])

	cfg.PassStructByReference=true;
R2013a	void strin(const struct_T *s, real_T y[4])
R2013b (default prj)	void strin(const struct_T *s, double y[4])
R2014a (default prj)	void strin(const struct0_T *s, double y[4])

MEX code:

	cfg.PassStructByReference not applicable
R2013a	void strin(const struct_T *s, real_T y[4])
R2013b	void strin(const emlrtStack *sp, const struct_T *s, real_T y[4])
R2014a	void strin(const struct0_T *s, real_T y[4])

3. Structure is an output

```
function s = strout(x)
s.f = x;
```

Stand-alone C code:

	cfg.PassStructByReference=false
R2013a (default)	struct_T strout(const real_T x[4])
R2013b (default cfg)	struct_T strout(const double x[4])
R2014a (default cfg)	struct0_T strout(const double x[4])

	cfg.PassStructByReference=true;
R2013a	struct_T strout(const real_T x[4])
R2013b (default prj)	void strout(const double x[4], struct_T *s)
R2014a (default prj)	void strout(const double x[4], struct0_T *s)

Default from the user interface is `cfg.PassStructByReference=true` for R2013b and R2014a, but from the command line the default is `cfg.PassStructByReference=false`.

MEX code:

	cfg.PassStructByReference not applicable
R2013a	struct_T strout(const real_T x[4])
R2013b	void strout(const emlrtStack *sp, const real_T x[4], struct_T *s)
R2014a	void strout(const real_T x[4], struct0_T *s)

4. Structure is an input and an output

```
function [y,s] = strinout(x,s)
y = x + sum(s.f);
```

Stand-alone C code:

	cfg.PassStructByReference=false
R2013a (default)	void strinout(const real_T x[4], const struct_T *s, real_T y[4])
R2013b (default cfg)	void strinout(const double x[4], const struct_T *s, double y[4])
R2014a (default cfg)	void strinout(const double x[4], const struct0_T *s, double y[4])

	cfg.PassStructByReference=true;
R2013a	Same as false
R2013b (default prj)	Same as false
R2014a (default prj)	Same as false

Default from the user interface is `cfg.PassStructByReference=true` for R2013b and R2014a, but from the command line the default is `cfg.PassStructByReference=false`.

MEX code:

	cfg.PassStructByReference not applicable
R2013a	void strinout(const real_T x[4], const struct_T *s, real_T y[4])
R2013b	void strinout(const emlrtStack *sp, const real_T x[4], const struct_T *s, real_T y[4])
R2014a	void strinout(const real_T x[4], const struct0_T *s, real_T y[4])

Appendix A:

Image Processing Toolbox Code Generation Details:

Function	Generates standalone C code (any target)	Generates standalone C code using platform-specific shared library (applies when hardware is set to "MATLAB Host Computer")	Requires dynamic memory allocation support	Requires enabling variable sizing support	Comments/limitations
`affine2d`	14a	NA	No	No	
`bwdist`	No	14b	No	No	
`bweuler`	15a	15a	No	Yes	
`bwlabel`	15a	NA	Yes	Yes	
`bwlookup`	14b	12b	No	Yes	
`bwmorph`	14b	12b	No	No	
`bwpack`	No	14a	No	No	
`bwperim`	15a	15a	No	Yes	
`bwselect`	15a	14a	No	Yes	
`bwtraceboundary`	14b	NA	Yes	Yes	
`bwunpack`	No	14a	No*	No*	* Dynamic memory allocation is not required provided M is a compile-time constants.
`conndef`	13a	NA	No	No	

`edge`	15a	14a	No*	Yes	* Dynamic memory allocation is not required provided THRESH and SIGMA are compile-time constants.
`fitgeotrans`	14b	NA	No	No	
`fspecial`	Pre-12b%	NA	No*	Yes	* Dynamic memory allocation is not required provided HSIZE, RADIUS, LEN and THETA are compile-time constants. Prior to 14b, this function generated constant-folded C code. In 14b, the function started generating C code.
`getrangefrom class`	14a	NA	No	No	
`histeq`	No	14b	No*	No*	* Dynamic memory allocation and variable-sizing support is not required provided N is a compile-time constant.
`hsv2rgb^`	14b	NA	No	No	
`im2uint8`	15a	14a	No	No	
`im2uint16`	No	14a	No	No	
`im2int16`	No	14a	No	No	
`im2single`	14a	NA	No	No	
`im2double^`	14a	NA	No	No	
`imadjust`	No	14b	No	No	
`imbothat`	14b	14a	No	Yes	
`imclearborde r`	15a	14b	No	No	
`imclose`	14b	14a	No	Yes	
`imcomplement`	13a	NA	No	No	
`imdilate`	14b	14a	No	Yes	
`imerode`	14b	14a	No	Yes	

imextendedmax	15a	14a	No	Yes	
imextendedmin	15a	14a	No	Yes	
imfill	15a	13a	No	Yes	
imfilter	14b	14a	No	Yes	
imhist	15a	14a	No*	No*	* Both dynamic memory allocation and variable-sizing support is not required provided N, the number of bins is a compile-time constant.
imhmax	15a	13a	No	Yes	
imhmin	15a	13a	No	Yes	
imlincomb	No	14b			
imopen	14b	14a	No	Yes	
imquantize	14b	NA	No	No	
imreconstruct	15a	13a	No	Yes	
imref2d	14a	NA	No	No	
imref3d	14a	NA	No	Yes	
imregionalmax	15a	13a	No	Yes	
imregionalmin	15a	13a	No	Yes	
imtophat	14b	14a	No	Yes	
imwarp	15a	14a	No*	Yes	* Dynamic memory allocation is not required provided TFORM is a compile-time constant.
intlut	No	14b	No	No	
iptcheckconn	13a	NA	No	No	
iptcheckmap	14b	NA	No	No	
label2rgb	Pre-12b	NA	No	No	

mean2	13b	NA	No	No	
medfilt2	15a	14b	No*	Yes	* Dynamic memory allocation is not required provided [M N], the neighborhood size is a compile-time constant.
multithresh	15a	14b	No*	Yes	* Dynamic memory allocation is not required provided the number of thresholds, N is a compile-time constant.
ordfilt2	15a	14b	No	Yes	
padarray	13a	NA	No*	No*	* Both dynamic memory allocation and variable-sizing support is not required provided PADSIZE is a compile-time constant.
projective2d	14a	NA	No	No	
rgb2gray^	14b	NA	No	No	
rgb2hsv^	14b	NA	No	No	
rgb2ycbcr	No	14b	No	No	
regionprops	15a	15a	Yes	Yes	
strel	14a	NA	No	No	
stretchlim	15a	14b	No	No	
watershed	15a	15a	Yes	Yes	
ycbcr2rgb	No	14b	No	No	
^Indicates that the function is in base MATLAB.					
* Indicates that the function supports a feature provided certain conditions in the Comments/Limitations section are met.					

% Indicates an enhancement to the generated code.					

Appendix B:

Suggested Function Replacements for Unsupported Functions

MATLAB function	**Suggested replacement**	**Comments**
`imcrop`	`vision.ImagePadder`	`imcrop` not supported for code generation
`imrotate`	`vision.GeometricRotator`	`imrotate` not supported for code generation

`graythresh`	`multithresh`	`thresh = graythresh(img);`
		
		can be replaced by:
		
		`thresh = ultithresh(img,1);`

`graythresh`	`vision.Autothresholder`	`bw = im2bw(img,graythresh(img))`
		
		can be replaced by
		
		`AT = vision.Autothresholder;`
		
		`im2bw` takes threshold input on [0,1]. You may need to recast `img` as type double or to re-scale `thresh` to match type of `img`.

`im2bw`		`im2bw(img,thresh)`
		
		can be replaced by
		
		`bw = img > thresh;`

		im2bw takes threshold input on [0,1]. You may need to recast img as type double or to re-scale thresh to match type of img.
im2bw		In R2014b and later, use imquantize in place of im2bw
imshow		Consider removing visualizations for codegen
regionprops	vision.BlobAnalysis	
labelmatrix	vision.ConnectedComponentLabeler	
imread		Consider passing image (matrix) in directly instead of using imread
imread	vision.VideoFileReader	Reads in images (.jpg, .bmp only), video, and audio
imread	Use OpenCV calls (i.e. cv::imread) in a C++ environment.	
bwlabel	vision.ConnectedComponentLabeler	
bwareaperim	vision.BlobAnalysis	
bwareaopen	vision.MorphologicalOpen	
bwareaopen	vision.ConnectedComponentLabeler in conjunction with vision.Autothresholder	
bwareaopen	vision.BlobAnalysis	