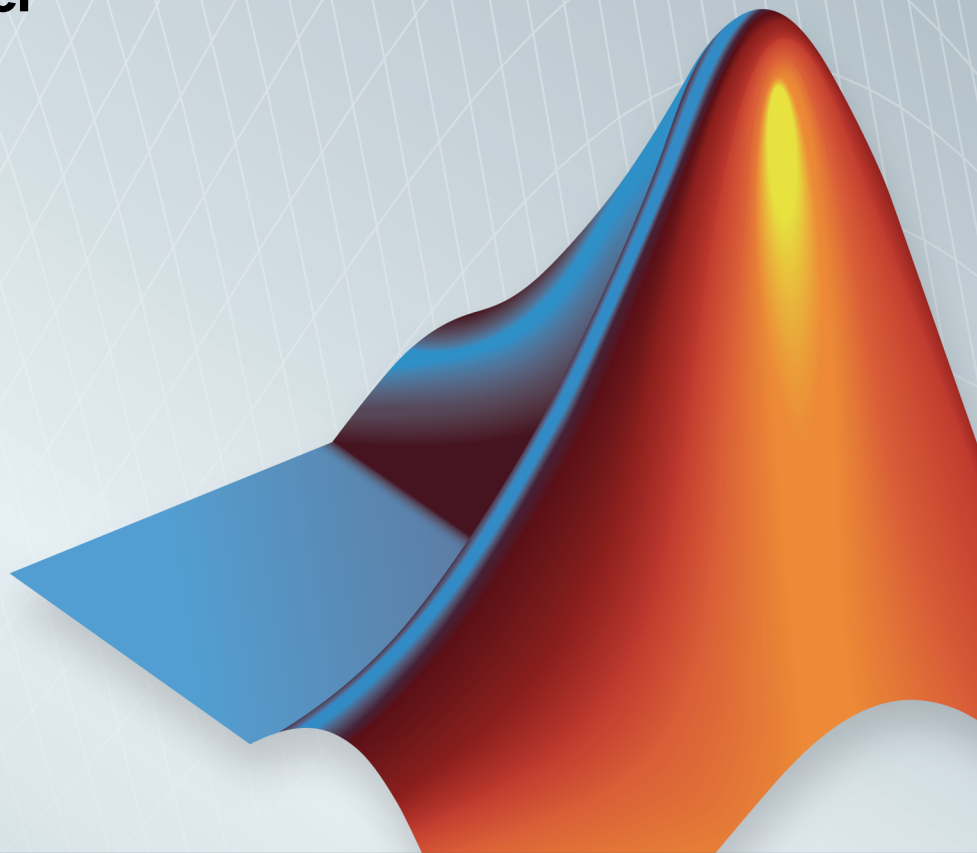


**MATLAB<sup>®</sup> Coder<sup>™</sup>**

User's Guide

R2014b



**MATLAB<sup>®</sup>**



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*MATLAB® Coder™ User's Guide*

© COPYRIGHT 2011–2014 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### Revision History

April 2011	Online only	New for Version 2 (R2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.4 (Release 2013a)
September 2013	Online only	Revised for Version 2.5 (Release 2013b)
March 2014	Online only	Revised for Version 2.6 (Release 2014a)
October 2014	Online only	Revised for Version 2.7 (Release 2014b)

## 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/](http://www.mathworks.com/support/bugreports/). Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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



## About MATLAB Coder

1

<b>MATLAB Coder Product Description</b> .....	1-2
Key Features .....	1-2
<b>Product Overview</b> .....	1-3
When to Use MATLAB Coder .....	1-3
Code Generation for Embedded Software Applications .....	1-3
Code Generation for Fixed-Point Algorithms .....	1-3
<b>Code Generation Workflow</b> .....	1-5
See Also .....	1-5

## Design Considerations for C/C++ Code Generation

2

<b>When to Generate Code from MATLAB Algorithms</b> .....	2-2
When Not to Generate Code from MATLAB Algorithms .....	2-2
<b>Which Code Generation Feature to Use</b> .....	2-4
<b>Prerequisites for C/C++ Code Generation from MATLAB</b> ...	2-5
<b>MATLAB Code Design Considerations for Code Generation</b>	2-6
See Also .....	2-7
<b>Differences in Behavior After Compiling MATLAB Code</b> ...	2-8
Why Are There Differences? .....	2-8
Character Size .....	2-8
Order of Evaluation in Expressions .....	2-8
Termination Behavior .....	2-9

Size of Variable-Size N-D Arrays .....	2-9
Size of Empty Arrays .....	2-9
Floating-Point Numerical Results .....	2-10
NaN and Infinity Patterns .....	2-10
Code Generation Target .....	2-11
MATLAB Class Initial Values .....	2-11
Variable-Size Support for Code Generation .....	2-11
<b>MATLAB Language Features Supported for C/C++ Code Generation .....</b>	<b>2-12</b>
MATLAB Language Features Not Supported for C/C++ Code Generation .....	2-12

### 3 **System Objects Supported for Code Generation**

Code Generation for System Objects .....	3-2
--	-----

### **Functions, Classes, and System Objects Supported for Code Generation**

## 4

<b>Functions and Objects Supported for C and C++ Code Generation — Alphabetical List .....</b>	<b>4-2</b>
--	------------

<b>Functions and Objects Supported for C and C++ Code Generation — Category List .....</b>	<b>4-134</b>
Aerospace Toolbox .....	4-136
Arithmetic Operations in MATLAB .....	4-136
Bit-Wise Operations MATLAB .....	4-137
Casting in MATLAB .....	4-138
Communications System Toolbox .....	4-138
Complex Numbers in MATLAB .....	4-144
Computer Vision System Toolbox .....	4-144
Control Flow in MATLAB .....	4-153
Data and File Management in MATLAB .....	4-154
Data Types in MATLAB .....	4-157

Desktop Environment in MATLAB .....	4-158
Discrete Math in MATLAB .....	4-158
DSP System Toolbox .....	4-159
Error Handling in MATLAB .....	4-166
Exponents in MATLAB .....	4-167
Filtering and Convolution in MATLAB .....	4-167
Fixed-Point Designer .....	4-168
HDL Coder .....	4-178
Histograms in MATLAB .....	4-178
Image Acquisition Toolbox .....	4-178
Image Processing in MATLAB .....	4-178
Image Processing Toolbox .....	4-179
Input and Output Arguments in MATLAB .....	4-186
Interpolation and Computational Geometry in MATLAB ..	4-187
Linear Algebra in MATLAB .....	4-190
Logical and Bit-Wise Operations in MATLAB .....	4-191
MATLAB Compiler .....	4-191
Matrices and Arrays in MATLAB .....	4-192
Neural Network Toolbox .....	4-199
Nonlinear Numerical Methods in MATLAB .....	4-199
Numerical Integration and Differentiation in MATLAB ...	4-199
Optimization Functions in MATLAB .....	4-200
Phased Array System Toolbox .....	4-201
Polynomials in MATLAB .....	4-209
Programming Utilities in MATLAB .....	4-209
Relational Operators in MATLAB .....	4-209
Rounding and Remainder Functions in MATLAB .....	4-210
Set Operations in MATLAB .....	4-210
Signal Processing in MATLAB .....	4-215
Signal Processing Toolbox .....	4-216
Special Values in MATLAB .....	4-221
Specialized Math in MATLAB .....	4-221
Statistics in MATLAB .....	4-222
Statistics Toolbox .....	4-222
String Functions in MATLAB .....	4-231
Structures in MATLAB .....	4-233
Trigonometry in MATLAB .....	4-233

## Defining MATLAB Variables for C/C++ Code Generation

### 5

<b>Variables Definition for Code Generation</b> .....	5-2
<b>Best Practices for Defining Variables for C/C++ Code Generation</b> .....	5-3
Define Variables By Assignment Before Using Them .....	5-3
Use Caution When Reassigning Variables .....	5-5
Use Type Cast Operators in Variable Definitions .....	5-5
Define Matrices Before Assigning Indexed Variables .....	5-6
<b>Eliminate Redundant Copies of Variables in Generated Code</b> .....	5-7
When Redundant Copies Occur .....	5-7
How to Eliminate Redundant Copies by Defining Uninitialized Variables .....	5-7
Defining Uninitialized Variables .....	5-8
<b>Reassignment of Variable Properties</b> .....	5-9
<b>Define and Initialize Persistent Variables</b> .....	5-10
<b>Reuse the Same Variable with Different Properties</b> .....	5-11
When You Can Reuse the Same Variable with Different Properties .....	5-11
When You Cannot Reuse Variables .....	5-11
Limitations of Variable Reuse .....	5-14
<b>Avoid Overflows in for-Loops</b> .....	5-15
<b>Supported Variable Types</b> .....	5-17

## Defining Data for Code Generation

### 6

<b>Data Definition for Code Generation</b> .....	6-2
--	-----



<b>Code Generation for Complex Data</b> .....	<b>6-4</b>
Restrictions When Defining Complex Variables .....	<b>6-4</b>
Expressions With Complex Operands Yield Complex Results .	<b>6-4</b>
<b>Code Generation for Characters</b> .....	<b>6-6</b>
<b>Array Size Restrictions for Code Generation</b> .....	<b>6-7</b>
See Also .....	<b>6-7</b>

## Code Generation for Variable-Size Data

# 7

<b>What Is Variable-Size Data?</b> .....	<b>7-2</b>
<b>Variable-Size Data Definition for Code Generation</b> .....	<b>7-3</b>
<b>Bounded Versus Unbounded Variable-Size Data</b> .....	<b>7-4</b>
<b>Control Memory Allocation of Variable-Size Data</b> .....	<b>7-5</b>
<b>Specify Variable-Size Data Without Dynamic Memory</b>	
<b>Allocation</b> .....	<b>7-6</b>
Fixing Upper Bounds Errors .....	<b>7-6</b>
Specifying Upper Bounds for Variable-Size Data .....	<b>7-6</b>
<b>Variable-Size Data in Code Generation Reports</b> .....	<b>7-9</b>
What Reports Tell You About Size .....	<b>7-9</b>
How Size Appears in Code Generation Reports .....	<b>7-10</b>
How to Generate a Code Generation Report .....	<b>7-10</b>
<b>Define Variable-Size Data for Code Generation</b> .....	<b>7-11</b>
When to Define Variable-Size Data Explicitly .....	<b>7-11</b>
Using a Matrix Constructor with Nonconstant Dimensions .	<b>7-11</b>
Inferring Variable Size from Multiple Assignments .....	<b>7-12</b>
Defining Variable-Size Data Explicitly Using <code>coder.varsize</code> .	<b>7-13</b>
<b>C Code Interface for Arrays</b> .....	<b>7-17</b>
C Code Interface for Statically Allocated Arrays .....	<b>7-17</b>
C Code Interface for Dynamically Allocated Arrays .....	<b>7-18</b>
Utility Functions for Creating <code>emxArray</code> Data Structures . .	<b>7-19</b>

<b>Diagnose and Fix Variable-Size Data Errors</b> .....	<b>7-21</b>
Diagnosing and Fixing Size Mismatch Errors .....	<b>7-21</b>
Diagnosing and Fixing Errors in Detecting Upper Bounds ..	<b>7-23</b>
<b>Incompatibilities with MATLAB in Variable-Size Support for Code Generation</b> .....	<b>7-25</b>
Incompatibility with MATLAB for Scalar Expansion .....	<b>7-25</b>
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays .....	<b>7-27</b>
Incompatibility with MATLAB in Determining Size of Empty Arrays .....	<b>7-28</b>
Incompatibility with MATLAB in Determining Class of Empty Arrays .....	<b>7-29</b>
Incompatibility with MATLAB in Vector-Vector Indexing ..	<b>7-30</b>
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation .....	<b>7-30</b>
Incompatibility with MATLAB in Concatenating Variable-Size Matrices .....	<b>7-31</b>
Dynamic Memory Allocation Not Supported for MATLAB Function Blocks .....	<b>7-32</b>
<b>Variable-Sizing Restrictions for Code Generation of Toolbox Functions</b> .....	<b>7-33</b>
Common Restrictions .....	<b>7-33</b>
Toolbox Functions with Variable Sizing Restrictions .....	<b>7-34</b>

## Code Generation for MATLAB Structures

# 8

<b>Structure Definition for Code Generation</b> .....	<b>8-2</b>
<b>Structure Operations Allowed for Code Generation</b> .....	<b>8-3</b>
<b>Define Scalar Structures for Code Generation</b> .....	<b>8-4</b>
Restriction When Using struct .....	<b>8-4</b>
Restrictions When Defining Scalar Structures by Assignment .....	<b>8-4</b>
Adding Fields in Consistent Order on Each Control Flow Path .....	<b>8-4</b>
Restriction on Adding New Fields After First Use .....	<b>8-5</b>

<b>Define Arrays of Structures for Code Generation</b> .....	8-7
Ensuring Consistency of Fields .....	8-7
Using repmat to Define an Array of Structures with Consistent Field Properties .....	8-7
Defining an Array of Structures Using Concatenation .....	8-8
<b>Make Structures Persistent</b> .....	8-9
<b>Index Substructures and Fields</b> .....	8-10
<b>Assign Values to Structures and Fields</b> .....	8-12
<b>Pass Structure Arguments by Reference or by Value</b> .....	8-14
Specify Pass by Reference or by Value Using a Project .....	8-14
Specify Pass by Reference or by Value Using the Command- Line Interface .....	8-15
Pass Input Structure Argument by Reference .....	8-15
Pass Input Structure Argument by Value .....	8-16
Pass Output Structure Argument by Reference .....	8-16
Pass Output Structure Argument by Value .....	8-17
Pass Input and Output Structure Argument by Reference ..	8-18

## Code Generation for Enumerated Data

# 9

<b>Enumerated Data Definition for Code Generation</b> .....	9-2
<b>Enumerated Types Supported for Code Generation</b> .....	9-3
Enumeration Class Base Types for Code Generation .....	9-3
C Code Representation for Base Type int32 .....	9-4
C Code Representation for Base Type Other Than int32 ...	9-4
<b>When to Use Enumerated Data for Code Generation</b> .....	9-6
<b>Generate Code for Enumerated Data from MATLAB</b>	
<b>Algorithms</b> .....	9-7
See Also .....	9-7
<b>Define Enumerated Data for Code Generation</b> .....	9-8
Naming Enumerated Types for Code Generation .....	9-9

<b>Operations on Enumerated Data for Code Generation . . . .</b>	<b>9-10</b>
Assignment Operator, = . . . . .	9-10
Relational Operators, < > <= >= == ~= . . . . .	9-10
Cast Operation . . . . .	9-10
Indexing Operation . . . . .	9-11
Control Flow Statements: if, switch, while . . . . .	9-11
<b>Include Enumerated Data in Control Flow Statements . . .</b>	<b>9-13</b>
if Statement with Enumerated Data Types . . . . .	9-13
switch Statement with Enumerated Data Types . . . . .	9-14
while Statement with Enumerated Data Types . . . . .	9-16
<b>Customize Enumerated Types for Code Generation . . . . .</b>	<b>9-19</b>
Customizing Enumerated Types . . . . .	9-19
Specify a Default Enumerated Value . . . . .	9-20
Specify a Header File . . . . .	9-21
<b>Use Enumerated Types in LED Control Function . . . . .</b>	<b>9-23</b>
<b>Control Names of Enumerated Type Values in Generated Code . . . . .</b>	<b>9-26</b>
<b>Change and Reload Enumerated Data Types . . . . .</b>	<b>9-29</b>
<b>Restrictions on Use of Enumerated Data in for-Loops . . . .</b>	<b>9-30</b>
<b>Toolbox Functions That Support Enumerated Types for Code Generation . . . . .</b>	<b>9-31</b>

## Code Generation for MATLAB Classes

# 10

<b>MATLAB Classes Definition for Code Generation . . . . .</b>	<b>10-2</b>
Language Limitations . . . . .	10-2
Code Generation Features Not Compatible with Classes . . .	10-3
Defining Class Properties for Code Generation . . . . .	10-4
Calls to Base Class Constructor . . . . .	10-5
Inheritance from Built-In MATLAB Classes Not Supported .	10-6
<b>Classes That Support Code Generation . . . . .</b>	<b>10-7</b>

<b>Generate Code for MATLAB Value Classes</b> .....	<b>10-8</b>
<b>Generate Code for MATLAB Handle Classes and System Objects</b> .....	<b>10-13</b>
<b>MATLAB Classes in Code Generation Reports</b> .....	<b>10-15</b>
What Reports Tell You About Classes .....	<b>10-15</b>
How Classes Appear in Code Generation Reports .....	<b>10-15</b>
How to Generate a Code Generation Report .....	<b>10-17</b>
<b>Troubleshooting Issues with MATLAB Classes</b> .....	<b>10-18</b>
Class class does not have a property with name name .....	<b>10-18</b>

## Code Generation for Function Handles

# 11

<b>Function Handle Definition for Code Generation</b> .....	<b>11-2</b>
<b>Define and Pass Function Handles for Code Generation</b> ..	<b>11-3</b>
<b>Function Handle Limitations for Code Generation</b> .....	<b>11-5</b>

## Defining Functions for Code Generation

# 12

<b>Specify Variable Numbers of Arguments</b> .....	<b>12-2</b>
<b>Supported Index Expressions</b> .....	<b>12-3</b>
<b>Apply Operations to a Variable Number of Arguments</b> .....	<b>12-4</b>
When to Force Loop Unrolling .....	<b>12-4</b>
Using Variable Numbers of Arguments in a for-Loop .....	<b>12-5</b>
<b>Implement Wrapper Functions</b> .....	<b>12-6</b>
Passing Variable Numbers of Arguments from One Function to Another .....	<b>12-6</b>

Pass Property/Value Pairs .....	12-7
Variable Length Argument Lists for Code Generation ....	12-9

## Calling Functions for Code Generation

# 13

<b>Resolution of Function Calls for Code Generation .....</b>	<b>13-2</b>
Key Points About Resolving Function Calls .....	13-4
Compile Path Search Order .....	13-4
When to Use the Code Generation Path .....	13-5
<b>Resolution of File Types on Code Generation Path .....</b>	<b>13-6</b>
<b>Compilation Directive %#codegen .....</b>	<b>13-8</b>
<b>Call Local Functions .....</b>	<b>13-9</b>
<b>Call Supported Toolbox Functions .....</b>	<b>13-10</b>
<b>Call MATLAB Functions .....</b>	<b>13-11</b>
Declaring MATLAB Functions as Extrinsic Functions ....	13-12
Calling MATLAB Functions Using feval .....	13-16
How MATLAB Resolves Extrinsic Functions During Simulation .....	13-16
Working with mxArrayArrays .....	13-17
Restrictions on Extrinsic Functions for Code Generation ..	13-19
Limit on Function Arguments .....	13-19

## Fixed-Point Conversion

# 14

<b>Convert MATLAB Code to Fixed-Point C Code .....</b>	<b>14-3</b>
<b>Propose Fixed-Point Data Types Based on Simulation     Ranges .....</b>	<b>14-4</b>

<b>Propose Fixed-Point Data Types Based on Derived Ranges</b> .....	14-18
<b>Type Proposal Settings</b> .....	14-34
<b>Detect Overflows</b> .....	14-38
<b>Replace the exp Function with a Lookup Table</b> .....	14-45
<b>Replace a Custom Function with a Lookup Table</b> .....	14-51
<b>Enable Plotting Using the Simulation Data Inspector</b> ...	14-58
<b>Log Data for Histogram</b> .....	14-59
<b>View and Modify Variable Information</b> .....	14-62
View Variable Information .....	14-62
Modify Variable Information .....	14-62
Revert Changes .....	14-64
Promote Sim Min and Sim Max Values .....	14-65
<b>Build Instrumented MEX Function</b> .....	14-66
<b>Propose Fixed-Point Data Types</b> .....	14-67
<b>Apply Fixed-Point Data Types</b> .....	14-77
<b>Modify Data Type Proposal Settings</b> .....	14-82
<b>Modify Instrumentation Report Settings</b> .....	14-85
<b>Automated Fixed-Point Conversion</b> .....	14-86
License Requirements .....	14-86
Automated Fixed-Point Conversion Capabilities .....	14-86
Code Coverage .....	14-88
Proposing Data Types .....	14-91
Locking Proposed Data Types .....	14-93
Viewing Functions .....	14-93
Viewing Variables .....	14-94
Histogram .....	14-100
Function Replacements .....	14-102
Validating Types .....	14-103
Testing Numerics .....	14-103

Detecting Overflows .....	14-104
<b>Instrumented MEX Functions .....</b>	<b>14-105</b>
Generating Instrumented MEX Functions .....	14-105
Merging Instrumentation Results .....	14-105
Clearing Instrumentation Results .....	14-106
Redirecting Entry-Point Calls to MEX Function .....	14-106
Proposing Fraction Lengths .....	14-106
Proposing Word Lengths .....	14-106
<b>Convert Fixed-Point Conversion Project to MATLAB Scripts .....</b>	<b>14-107</b>
<b>Generated Fixed-Point Code .....</b>	<b>14-110</b>
Location of Generated Fixed-Point Files .....	14-110
Minimizing <code>fi</code> -casts to Improve Code Readability .....	14-111
Avoiding Overflows in the Generated Fixed-Point Code ..	14-111
Controlling Bit Growth .....	14-112
Avoiding Loss of Range or Precision .....	14-112
Handling Non-Constant <code>mpower</code> Exponents .....	14-114
<b>Fixed-Point Code for MATLAB Classes .....</b>	<b>14-116</b>
Automated Conversion Support for MATLAB Classes ...	14-116
Unsupported Constructs .....	14-116
Coding Style Best Practices .....	14-117
<b>Automated Fixed-Point Conversion Best Practices .....</b>	<b>14-119</b>
Create a Test File .....	14-119
Prepare Your Algorithm for Code Acceleration or Code Generation .....	14-120
Check for Fixed-Point Support for Functions Used in Your Algorithm .....	14-121
Manage Data Types and Control Bit Growth .....	14-121
Convert to Fixed Point .....	14-122
Use the Histogram to Fine-Tune Data Type Settings ....	14-122
Optimize Your Algorithm .....	14-124
Avoid Explicit Double and Single Casts .....	14-126
<b>Replacing Functions Using Lookup Table Approximations .....</b>	<b>14-128</b>
<b>MATLAB Language Features Supported for Automated Fixed-Point Conversion .....</b>	<b>14-129</b>



<b>Inspecting Data Using the Simulation Data Inspector . . .</b>	<b>14-131</b>
What Is the Simulation Data Inspector? . . . . .	14-131
Import Logged Data . . . . .	14-131
Export Logged Data . . . . .	14-131
Group Signals . . . . .	14-131
Run Options . . . . .	14-132
Create Report . . . . .	14-132
Comparison Options . . . . .	14-132
Enabling Plotting Using the Simulation Data Inspector . .	14-132
Save and Load Simulation Data Inspector Sessions . . . . .	14-132
<b>Custom Plot Functions . . . . .</b>	<b>14-134</b>
<b>Data Type Issues in Generated Code . . . . .</b>	<b>14-136</b>
Enable the Highlight Option in a MATLAB Coder Project	14-136
Enable the Highlight Option at the Command Line . . . . .	14-136
Stowaway Doubles . . . . .	14-136
Stowaway Singles . . . . .	14-136
Expensive Fixed-Point Operations . . . . .	14-136

## Automated Fixed-Point Conversion Using Programmatic Workflow

# 15

<b>Convert MATLAB Code to Fixed-Point C Code . . . . .</b>	<b>15-2</b>
<b>Propose Fixed-Point Data Types Based on Simulation     Ranges . . . . .</b>	<b>15-5</b>
<b>Propose Fixed-Point Data Types Based on Derived     Ranges . . . . .</b>	<b>15-11</b>
<b>Detect Overflows . . . . .</b>	<b>15-19</b>
<b>Replace the exp Function with a Lookup Table . . . . .</b>	<b>15-23</b>
<b>Replace a Custom Function with a Lookup Table . . . . .</b>	<b>15-25</b>
<b>Enable Plotting Using the Simulation Data Inspector . . .</b>	<b>15-28</b>

<b>Visualize Differences Between Floating-Point and Fixed-Point Results</b> .....	<b>15-29</b>
---	--------------

# 16

## Setting Up a MATLAB Coder Project

<b>MATLAB Coder Project Set Up Workflow</b> .....	<b>16-2</b>
<b>Creating a New Project</b> .....	<b>16-3</b>
From the MATLAB APPS Tab .....	<b>16-3</b>
At the Command Line .....	<b>16-3</b>
From a MATLAB Coder Project .....	<b>16-4</b>
<b>Opening an Existing Project</b> .....	<b>16-5</b>
From the MATLAB APPS Tab .....	<b>16-5</b>
At the Command Line .....	<b>16-5</b>
From a MATLAB Coder Project .....	<b>16-5</b>
<b>Adding Files to the Project</b> .....	<b>16-6</b>
<b>Specifying Properties of Primary Function Inputs in a Project</b> .....	<b>16-7</b>
Why You Must Specify Input Properties .....	<b>16-7</b>
How to Specify an Input Definition in a Project .....	<b>16-7</b>
<b>Autodefine Input Types</b> .....	<b>16-8</b>
How MATLAB Coder Autodefines Input Types .....	<b>16-8</b>
Prerequisites for Autodefining Input Types .....	<b>16-8</b>
How to Autodefine Input Types .....	<b>16-8</b>
<b>Define Input Parameters by Example in a Project</b> .....	<b>16-12</b>
How to Define an Input Parameter by Example .....	<b>16-12</b>
Specifying Input Parameters by Example .....	<b>16-13</b>
Specifying an Enumerated Type Input Parameter by Example .....	<b>16-15</b>
Specifying a Fixed-Point Input Parameter by Example ...	<b>16-16</b>
<b>Define or Edit Input Parameter Type in a Project</b> .....	<b>16-19</b>
How to Define or Edit an Input Parameter Type .....	<b>16-19</b>
Specifying an Enumerated Type Input Parameter by Type .....	<b>16-21</b>

Specifying a Fixed-Point Input Parameter by Type . . . . .	16-22
Specifying Structures . . . . .	16-23
<b>Define Constant Input Parameters in a Project . . . . .</b>	<b>16-29</b>
<b>Define Inputs Programmatically in the MATLAB File . . . . .</b>	<b>16-30</b>
<b>Adding Global Variables in a Project . . . . .</b>	<b>16-31</b>
<b>Specifying Global Variable Type and Initial Value in a Project . . . . .</b>	<b>16-32</b>
Why Specify a Type Definition for Global Variables? . . . . .	16-32
How to Specify a Global Variable Type . . . . .	16-32
Defining a Global Variable by Example . . . . .	16-33
Defining or Editing Global Variable Type . . . . .	16-34
Defining Global Variable Initial Value . . . . .	16-36
Defining Global Variable Constant Value . . . . .	16-38
Removing Global Variables . . . . .	16-39
<b>Specify Output File Name . . . . .</b>	<b>16-40</b>
Command Line Alternative . . . . .	16-40
<b>Specify Output File Locations . . . . .</b>	<b>16-41</b>
Command Line Alternative . . . . .	16-41
<b>Selecting Output Type . . . . .</b>	<b>16-42</b>
Command Line Alternative . . . . .	16-42
Changing Output Type . . . . .	16-42

## 17 Preparing MATLAB Code for C/C++ Code Generation

<b>Workflow for Preparing MATLAB Code for Code Generation . . . . .</b>	<b>17-2</b>
See Also . . . . .	17-3
<b>Fixing Errors Detected at Design Time . . . . .</b>	<b>17-4</b>
See Also . . . . .	17-4
<b>Using the Code Analyzer . . . . .</b>	<b>17-5</b>

<b>Check Code With the Code Analyzer</b> .....	<b>17-6</b>
<b>Check Code Using the Code Generation Readiness Tool</b> ..	<b>17-8</b>
Run Code Generation Readiness Tool at the Command Line	17-8
Run Code Generation Readiness Tool from the Current Folder	
Browser .....	17-8
Run the Code Generation Readiness Tool in a Project .....	17-8
See Also .....	17-9
<b>Code Generation Readiness Tool</b> .....	<b>17-10</b>
What Information Does the Code Generation Readiness Tool	
Provide? .....	17-10
Summary Tab .....	17-11
Code Structure Tab .....	17-13
See Also .....	17-16
<b>Unable to Determine Code Generation Readiness</b> .....	<b>17-17</b>
<b>Generate MEX Functions Using the MATLAB Coder Project</b>	
<b>Interface</b> .....	<b>17-18</b>
Project Workflow for Generating MEX Functions .....	17-18
Generate MEX Functions Using the Project Interface .....	17-18
Configure Project Settings .....	17-23
Build a MATLAB Coder Project .....	17-24
See Also .....	17-25
<b>Generate MEX Functions at the Command Line</b> .....	<b>17-26</b>
Command-line Workflow for Generating MEX Functions ..	17-26
Generate MEX Functions at the Command Line .....	17-26
Generating MEX Functions at the Command Line Using	
codegen .....	17-27
See Also .....	17-27
<b>Fix Errors Detected at Code Generation Time</b> .....	<b>17-28</b>
See Also .....	17-28
<b>Design Considerations When Writing MATLAB Code for Code</b>	
<b>Generation</b> .....	<b>17-29</b>
See Also .....	17-30
<b>Running MEX Functions</b> .....	<b>17-31</b>
Debugging MEX Functions .....	17-31

Debugging Strategies .....	17-32
----------------------------	-------

## Testing MEX Functions in MATLAB

# 18

<b>Workflow for Testing MEX Functions in MATLAB</b> .....	18-2
See Also .....	18-2
<b>Why Test MEX Functions in MATLAB?</b> .....	18-4
<b>Running MEX Functions</b> .....	18-5
Debugging MEX Functions .....	18-5
<b>Verify MEX Functions in a Project</b> .....	18-6
Using Test Files That Call Only MATLAB Functions .....	18-6
Using Test Files That Call MEX Functions .....	18-7
<b>Verify MEX Functions at the Command Line</b> .....	18-8
<b>Debug Run-Time Errors</b> .....	18-9
Viewing Errors in the Run-Time Stack .....	18-9
Handling Run-Time Errors .....	18-10

## Generating C/C++ Code from MATLAB Code

# 19

<b>Code Generation Workflow</b> .....	19-3
See Also .....	19-3
<b>C/C++ Code Generation</b> .....	19-5
Specify Custom Files to Build .....	19-5
<b>Generating C/C++ Static Libraries from MATLAB Code</b> ...	19-6
Generate a C Static Library Using the Project Interface ...	19-6
Generate a C Static Library at the Command Line .....	19-9

<b>Generating C/C++ Dynamically Linked Libraries from MATLAB Code</b> .....	<b>19-10</b>
Dynamic Libraries Generated by MATLAB Coder .....	19-10
Generate a C Dynamically Linked Library (DLL) Using the Project Interface .....	19-10
Generate a C Dynamic Library at the Command Line .....	19-12
<b>Generating Standalone C/C++ Executables from MATLAB Code</b> .....	<b>19-13</b>
Generate a C Executable Using the Project Interface .....	19-13
Generate a C Executable at the Command Line .....	19-15
Specifying main Functions for C/C++ Executables .....	19-16
Specify main Functions .....	19-17
<b>Build Setting Configuration</b> .....	<b>19-19</b>
Specify Output Type .....	19-19
Specify a Language for Code Generation .....	19-21
Specify Data Type Used in Generated Code .....	19-22
Specify Output File Name .....	19-23
Specify Output File Locations .....	19-24
Parameter Specification Methods .....	19-25
Specify Build Configuration Parameters .....	19-25
<b>Standard Math Libraries</b> .....	<b>19-32</b>
<b>Change the Standard Math Library</b> .....	<b>19-33</b>
See Also .....	19-33
<b>Share Build Configuration Settings</b> .....	<b>19-34</b>
Export Settings .....	19-34
Import Settings .....	19-35
See Also .....	19-36
<b>Convert MATLAB Coder Project to MATLAB Script</b> .....	<b>19-37</b>
<b>Primary Function Input Specification</b> .....	<b>19-39</b>
Why You Must Specify Input Properties .....	19-39
Properties to Specify .....	19-39
Rules for Specifying Properties of Primary Inputs .....	19-42
Methods for Defining Properties of Primary Inputs .....	19-43
Define Input Properties by Example at the Command Line .....	19-44
Specify Constant Inputs at the Command Line .....	19-46
Specify Variable-Size Inputs at the Command Line .....	19-47

<b>Control Constant Inputs in MEX Function Signatures . . .</b>	<b>19-49</b>
Control MEX Function Signature Using the Project Interface . . . . .	19-49
Control MEX Function Signature at the Command-Line Interface . . . . .	19-49
Options for Controlling Constant Inputs in MEX Function Signatures . . . . .	19-49
Call MEX Function with a Constant Input . . . . .	19-51
See Also . . . . .	19-52
<b>Define Input Properties Programmatically in the MATLAB File . . . . .</b>	<b>19-53</b>
How to Use assert with MATLAB Coder . . . . .	19-53
Rules for Using assert Function . . . . .	19-59
Specifying General Properties of Primary Inputs . . . . .	19-59
Specifying Properties of Primary Fixed-Point Inputs . . . . .	19-60
Specifying Class and Size of Scalar Structure . . . . .	19-61
Specifying Class and Size of Structure Array . . . . .	19-62
<b>Speed Up Compilation . . . . .</b>	<b>19-63</b>
Generate Code Only . . . . .	19-63
Disable Compiler Optimization . . . . .	19-63
<b>Paths and File Infrastructure Setup . . . . .</b>	<b>19-65</b>
Compile Path Search Order . . . . .	19-65
Specifying Folders to Search for Custom Code . . . . .	19-65
Naming Conventions . . . . .	19-66
<b>Generate Code for Multiple Entry-Point Functions . . . . .</b>	<b>19-70</b>
Advantages of Generating Code for More Than One Entry-Point Function . . . . .	19-70
Generating Code for More Than One Entry-Point Function Using the Project Interface . . . . .	19-70
Generating Code for More Than One Entry-Point Function at the Command Line . . . . .	19-72
How to Call an Entry-Point Function in a MEX Function . . . . .	19-74
How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code . . . . .	19-74
<b>Generate Code for Global Data . . . . .</b>	<b>19-75</b>
Workflow . . . . .	19-75
Declare Global Variables . . . . .	19-75
Define Global Data . . . . .	19-76

Synchronizing Global Data with MATLAB .....	19-77
Define Constant Global Data .....	19-80
Limitations of Using Global Data .....	19-83
<b>Generation of Traceable Code .....</b>	<b>19-84</b>
About Code Traceability .....	19-84
Generate Traceable Code .....	19-84
Format of Traceability Tags .....	19-87
Location of Comments in Generated Code .....	19-87
Traceability Limitations .....	19-91
<b>Generate Code for Enumerated Types .....</b>	<b>19-93</b>
See Also .....	19-93
<b>Generate Code for Variable-Size Data .....</b>	<b>19-94</b>
Disable Support for Variable-Size Data .....	19-94
Control Dynamic Memory Allocation .....	19-95
Generating Code for MATLAB Functions with Variable-Size Data .....	19-97
Generate Code for a MATLAB Function That Expands a Vector in a Loop .....	19-98
Using Dynamic Memory Allocation for an "Atoms" Simulation .....	19-104
<b>Code Generation for MATLAB Classes .....</b>	<b>19-113</b>
<b>How MATLAB Coder Partitions Generated Code .....</b>	<b>19-114</b>
Partitioning Generated Files .....	19-114
How to Select the File Partitioning Method .....	19-114
Partitioning Generated Files with One C/C++ File Per MATLAB File .....	19-115
Generated Files and Locations .....	19-120
File Partitioning and Inlining .....	19-122
<b>Requirements for Signed Integer Representation .....</b>	<b>19-126</b>
<b>Customize the Post-Code-Generation Build Process .....</b>	<b>19-127</b>
Customize Build Using coder.updateBuildInfo .....	19-127
Customize Build Using Post-Code-Generation Command .	19-127
Build Information Object .....	19-128
Build Information Methods .....	19-128
Write Post-Code-Generation Command .....	19-164
Use Post-Code-Generation Command to Customize Build	19-165



Write and Use Post-Code-Generation Command at the Command Line .....	19-165
<b>Code Generation Reports</b> .....	19-167
About Code Generation Reports .....	19-167
Enable Code Generation Reports .....	19-170
View Your MATLAB Code in a Report .....	19-170
Viewing Call Stack Information .....	19-172
View Generated C and C++ Code in a Report .....	19-174
View the Build Summary Information .....	19-174
View Errors and Warnings in a Report .....	19-175
Viewing Variables in Your MATLAB Code .....	19-176
View Target Build Information .....	19-182
Keyboard Shortcuts for the Code Generation Report ....	19-183
Report Limitations .....	19-184
<b>Troubleshooting</b> .....	19-186
Run-time Stack Overflow .....	19-186
<b>Package Code For Other Development Environments</b> ..	19-187
When to Package Code .....	19-187
Package Generated Code in a Project .....	19-187
Package Generated Code at the Command Line .....	19-188
Specify packNGo options .....	19-189

## Code Replacement for MATLAB Code

# 20

<b>What Is Code Replacement?</b> .....	20-2
<b>Code Replacement Libraries</b> .....	20-4
<b>Code Replacement Terminology</b> .....	20-6
<b>Code Replacement Limitations</b> .....	20-9
<b>Replace Code Generated from MATLAB Code</b> .....	20-10
<b>Choose a Code Replacement Library</b> .....	20-12
About Choosing a Code Replacement Library .....	20-12

Explore Available Code Replacement Libraries .....	20-12
Explore Code Replacement Library Contents .....	20-12

## Custom Toolchain Registration

# 21

<b>Custom Toolchain Registration</b> .....	21-2
What Is a Custom Toolchain? .....	21-2
What Is a Factory Toolchain? .....	21-2
What is a Toolchain Definition? .....	21-3
Key Terms .....	21-4
Typical Workflow .....	21-4
 <b>About <code>codemake.ToolchainInfo</code></b> .....	21-6
 <b>Create and Edit Toolchain Definition File</b> .....	21-8
 <b>Toolchain Definition File with Commentary</b> .....	21-10
Steps Involved in Writing a Toolchain Definition File ....	21-10
Write a Function That Creates a ToolchainInfo Object ...	21-10
Setup .....	21-11
Macros .....	21-11
C Compiler .....	21-12
C++ Compiler .....	21-12
Linker .....	21-13
Archiver .....	21-13
Builder .....	21-14
Build Configurations .....	21-14
 <b>Create and Validate ToolchainInfo Object</b> .....	21-16
 <b>Register the Custom Toolchain</b> .....	21-17
 <b>Use the Custom Toolchain</b> .....	21-19
 <b>Troubleshooting Custom Toolchain Validation</b> .....	21-20
Build Tool Command Path Incorrect .....	21-20
Build Tool Not in System Path .....	21-20
Tool Path Does Not Exist .....	21-21
Unsupported Platform .....	21-21

Toolchain is Not installed . . . . .	21-22
Project or Configuration is Using the Template Makefile . .	21-22
Skipped Validation of Build Tool “Download” or “Execute” .	21-23
<b>Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers . . . . .</b>	<b>21-24</b>

## Deploying Generated Code

# 22

<b>Call a C Static Library Function from C Code . . . . .</b>	<b>22-2</b>
<b>Call a C/C++ Static Library Function from MATLAB Code .</b>	<b>22-4</b>
<b>Call Generated C/C++ Functions . . . . .</b>	<b>22-6</b>
Conventions for Calling Functions in Generated Code . . . . .	22-6
How to Call C/C++ Functions from MATLAB Code . . . . .	22-6
Calling Initialize and Terminate Functions . . . . .	22-7
Calling C/C++ Functions with Multiple Outputs . . . . .	22-8
Calling C/C++ Functions that Return Arrays . . . . .	22-8
<b>Use a MATLAB Coder Dynamic Library in a Simple Microsoft Visual Studio Project . . . . .</b>	<b>22-9</b>
<b>Specify External File Locations . . . . .</b>	<b>22-12</b>
External File Locations for External Code Integration . . . .	22-12
Specify External Files in a Class Derived from coder.ExternalDependency . . . . .	22-12
Specify External Files in MATLAB Code Using coder.updateBuildInfo . . . . .	22-12
Specify External Files in the Project Settings Dialog Box . .	22-13
Specify External Files at the Command Line . . . . .	22-13
Specify External Files with Configuration Objects . . . . .	22-14

<b>Workflow for Accelerating MATLAB Algorithms</b> .....	23-2
See Also .....	23-3
<b>Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms</b> .....	23-4
Accelerate Code That Dominates Execution Time .....	23-4
Include Loops Inside MEX Function .....	23-4
Avoid Generating MEX Functions from Unsupported Functions .....	23-5
Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time .....	23-6
Minimize MEX Function Calls .....	23-6
<b>Edge Detection on Images</b> .....	23-7
<b>Accelerate MATLAB Algorithms</b> .....	23-14
<b>Modifying MATLAB Code for Acceleration</b> .....	23-15
How to Modify Your MATLAB Code for Acceleration .....	23-15
<b>Control Run-Time Checks</b> .....	23-16
Types of Run-Time Checks .....	23-16
When to Disable Run-Time Checks .....	23-16
How to Disable Run-Time Checks .....	23-17
<b>Algorithm Acceleration Using Parallel for-Loops (parfor)</b> .....	23-18
Parallel for-Loops (parfor) in Generated Code .....	23-18
How parfor-Loops Improve Execution Speed .....	23-19
When to Use parfor-Loops .....	23-19
When Not to Use parfor-Loops .....	23-19
parfor-Loop Syntax .....	23-20
parfor Restrictions .....	23-20
<b>Control Compilation of parfor-Loops</b> .....	23-24
When to Disable parfor .....	23-24
<b>Reduction Assignments in parfor-Loops</b> .....	23-25
What are Reduction Assignments? .....	23-25
Multiple Reductions in a parfor-Loop .....	23-25

<b>Classification of Variables in parfor-Loops</b> .....	23-26
Overview .....	23-26
Sliced Variables .....	23-27
Broadcast Variables .....	23-28
Reduction Variables .....	23-28
Temporary Variables .....	23-33
<b>Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)</b> .....	23-35
<b>Specify Maximum Number of Threads in parfor-Loops</b> ..	23-36
<b>Troubleshooting parfor-Loops</b> .....	23-37
What Causes Errors With Global Structures in Parallel Regions? .....	23-37
Compiler Does Not Support OpenMP .....	23-37
<b>Accelerating Simulation of Bouncing Balls</b> .....	23-38

## 24

### Calling C/C++ Functions from Generated Code

<b>External Function Calls from Generated Code</b> .....	24-2
Calling External Functions from Generated Code .....	24-2
Why Call External Functions from Generated Code? .....	24-2
How To Call External Functions .....	24-2
Pass Arguments by Reference to External Functions .....	24-3
Manipulate C Data .....	24-4
<b>Call External Functions Using coder.ceval</b> .....	24-6
Workflow for Calling External Functions .....	24-6
Best Practices for Calling External Code from Generated Code .....	24-7
<b>Return Multiple Values from C Functions</b> .....	24-8
<b>How MATLAB Coder Infers C/C++ Data Types</b> .....	24-9
Mapping MATLAB Types to C/C++ Types .....	24-9
Mapping 64-Bit Integer Types to C/C++ .....	24-10
Mapping Fixed-Point Types to C/C++ .....	24-11

Mapping Arrays to C/C++ .....	24-11
Mapping Complex Values to C/C++ .....	24-12
Mapping Structures to C/C++ Structures .....	24-13
Mapping Strings to C/C++ .....	24-13
Mapping Multiword Types to C/C++ .....	24-14

## External Code Integration

# 25

<b>External Code Integration for Code Generation .....</b>	<b>25-2</b>
<b>Encapsulating the Interface to External Code .....</b>	<b>25-3</b>
<b>Best Practices for Using <code>coder.ExternalDependency</code> .....</b>	<b>25-4</b>
Terminate Code Generation for Unsupported External Dependency .....	25-4
Parameterize Methods for MATLAB and Generated Code ..	25-4
Parameterize <code>updateBuildInfo</code> for Multiple Platforms .....	25-5
<b>Encapsulate Interface to an External C Library .....</b>	<b>25-6</b>
<b>Update Build Information from MATLAB code .....</b>	<b>25-9</b>
<b>Call External Functions Encapsulated by     <code>coder.ExternalDependency</code> .....</b>	<b>25-10</b>

## Generate Efficient and Reusable Code

# 26

<b>Optimization Strategies .....</b>	<b>26-2</b>
<b>Modularize MATLAB Code .....</b>	<b>26-5</b>
<b>Eliminate Redundant Copies of Function Inputs .....</b>	<b>26-6</b>
<b>Inline Code .....</b>	<b>26-8</b>
Prevent Function Inlining .....	26-8

Use Inlining in Control Flow Statements .....	26-8
<b>Control Inlining Using Configuration Object .....</b>	<b>26-10</b>
Control Size of Functions Inlined .....	26-10
Control Size of Functions After Inlining .....	26-11
Control Stack Size Limit on Inlined Functions .....	26-11
<b>Fold Function Calls into Constants .....</b>	<b>26-13</b>
<b>Control Stack Space Usage .....</b>	<b>26-15</b>
<b>Stack Allocation and Performance .....</b>	<b>26-16</b>
<b>Rewrite Logical Array Indexing as a Loop .....</b>	<b>26-17</b>
<b>Dynamic Memory Allocation and Performance .....</b>	<b>26-18</b>
When Dynamic Memory Allocation Occurs .....	26-18
<b>Minimize Dynamic Memory Allocation .....</b>	<b>26-19</b>
<b>Provide Maximum Size for Variable-Size Arrays .....</b>	<b>26-20</b>
<b>Disable Dynamic Memory Allocation During Code     Generation .....</b>	<b>26-26</b>
<b>Set Dynamic Memory Allocation Threshold .....</b>	<b>26-27</b>
Set Dynamic Memory Allocation Threshold Using Project Interface .....	26-27
Set Dynamic Memory Allocation Threshold from Command Line .....	26-29
<b>Excluding Unused Paths from Generated Code .....</b>	<b>26-30</b>
<b>Prevent Code Generation for Unused Execution Paths ..</b>	<b>26-31</b>
Prevent Code Generation When Local Variable Controls Flow .....	26-31
Prevent Code Generation When Input Variable Controls Flow .....	26-32
<b>Generate Code with Parallel for-Loops (parfor) .....</b>	<b>26-33</b>
<b>Minimize Redundant Operations in Loops .....</b>	<b>26-35</b>

<b>Unroll for-Loops</b> .....	<b>26-37</b>
Limit Copying the for-loop Body in Generated Code .....	<b>26-37</b>
<b>Support for Integer Overflow and Non-Finites</b> .....	<b>26-40</b>
Disable Support for Integer Overflow .....	<b>26-40</b>
Disable Support for Non-Finites .....	<b>26-41</b>
<b>Integrate Custom Code</b> .....	<b>26-42</b>
<b>MATLAB Coder Optimizations in Generated Code</b> .....	<b>26-48</b>
Constant Folding .....	<b>26-48</b>
Loop Fusion .....	<b>26-49</b>
Successive Matrix Operations Combined .....	<b>26-49</b>
Unreachable Code Elimination .....	<b>26-50</b>
<b>Generate Reusable Code</b> .....	<b>26-51</b>



# About MATLAB Coder

---

- “MATLAB Coder Product Description” on page 1-2
- “Product Overview” on page 1-3
- “Code Generation Workflow” on page 1-5

# MATLAB Coder Product Description

## Generate C and C++ code from MATLAB code

MATLAB<sup>®</sup> Coder<sup>™</sup> generates standalone C and C++ code from MATLAB code. The generated source code is portable and readable. MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. It can generate MEX functions that let you accelerate computationally intensive portions of MATLAB code and verify the behavior of the generated code.

## Key Features

- ANSI<sup>®</sup>/ISO<sup>®</sup> compliant C and C++ code generation
- MEX function generation for fixed-point and floating-point math
- Project management tool for specifying entry points, input data properties, and other code-generation configuration options
- Static or dynamic memory allocation for variable-size data
- Code generation support for many functions and System objects in Communications System Toolbox<sup>™</sup>, DSP System Toolbox<sup>™</sup>, Computer Vision System Toolbox<sup>™</sup>, and Phased Array System Toolbox<sup>™</sup>
- Support for common MATLAB language features, including matrix operations, subscripting, program controls statements (if, switch, for, while), and structures

# Product Overview

**In this section...**

“When to Use MATLAB Coder” on page 1-3

“Code Generation for Embedded Software Applications” on page 1-3

“Code Generation for Fixed-Point Algorithms” on page 1-3

## When to Use MATLAB Coder

Use MATLAB Coder to:

- Generate readable, efficient, standalone C/C++ code from MATLAB code.
- Generate MEX functions from MATLAB code to:
  - Accelerate your MATLAB algorithms.
  - Verify generated C code within MATLAB.
- Integrate custom C/C++ code into MATLAB.

## Code Generation for Embedded Software Applications

The Embedded Coder<sup>®</sup> product extends the MATLAB Coder product with features that are important for embedded software development. Using the Embedded Coder add-on product, you can generate code that has the clarity and efficiency of professional handwritten code. For example, you can:

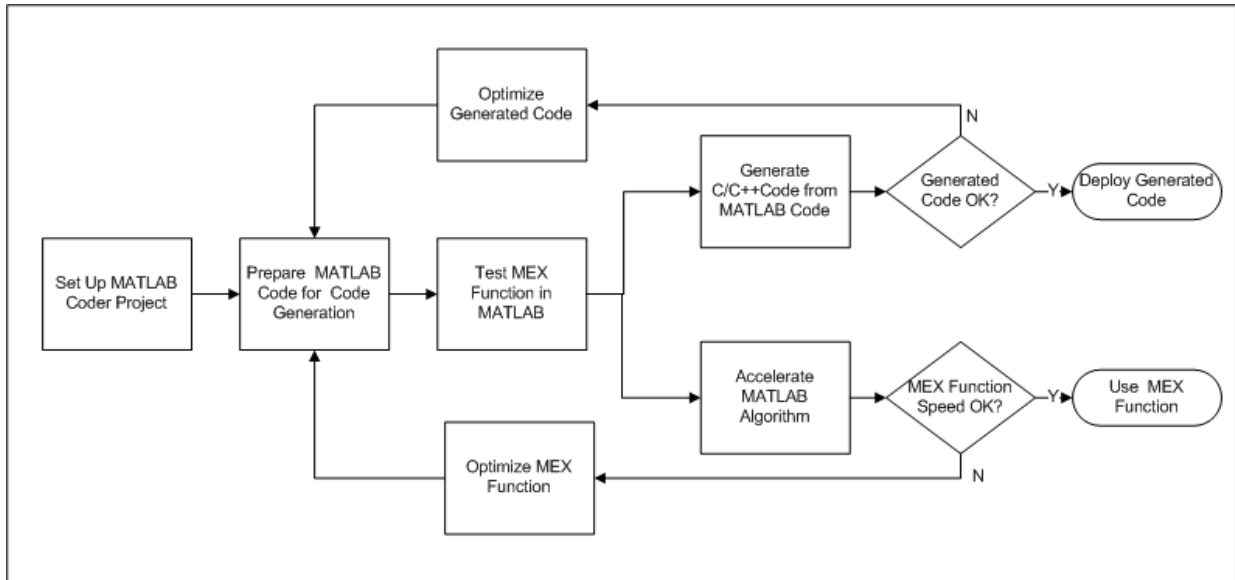
- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize the generated code for a specific target environment.
- Enable tracing options that help you to verify the generated code.
- Generate reusable, reentrant code.

## Code Generation for Fixed-Point Algorithms

Using the Fixed-Point Designer<sup>™</sup> product, you can generate:

- MEX functions to accelerate fixed-point algorithms.
- Fixed-point code that provides a bit-wise match to MEX function results.

## Code Generation Workflow



### See Also

- “MATLAB Coder Project Set Up Workflow”
- “Workflow for Preparing MATLAB Code for Code Generation”
- “Workflow for Testing MEX Functions in MATLAB”
- “Code Generation Workflow”
- “Workflow for Accelerating MATLAB Algorithms”



# Design Considerations for C/C++ Code Generation

---

- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Which Code Generation Feature to Use” on page 2-4
- “Prerequisites for C/C++ Code Generation from MATLAB” on page 2-5
- “MATLAB Code Design Considerations for Code Generation” on page 2-6
- “Differences in Behavior After Compiling MATLAB Code” on page 2-8
- “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-12

## When to Generate Code from MATLAB Algorithms

Generating code from MATLAB algorithms for desktop and embedded systems allows you to perform your software design, implementation, and testing completely within the MATLAB workspace. You can:

- Verify that your algorithms are suitable for code generation
- Generate efficient, readable, and compact C/C++ code automatically, which eliminates the need to manually translate your MATLAB algorithms and minimizes the risk of introducing errors in the code.
- Modify your design in MATLAB code to take into account the specific requirements of desktop and embedded applications, such as data type management, memory use, and speed.
- Test the generated code and easily verify that your modified algorithms are functionally equivalent to your original MATLAB algorithms.
- Generate MEX functions to:
  - Accelerate MATLAB algorithms in certain applications.
  - Speed up fixed-point MATLAB code.
- Generate hardware description language (HDL) from MATLAB code.

## When Not to Generate Code from MATLAB Algorithms

Do not generate code from MATLAB algorithms for the following applications. Use the recommended MathWorks product instead.

To:	Use:
Deploy an application that uses handle graphics	MATLAB Compiler™
Use Java®	MATLAB Builder™ JA
Use toolbox functions that do not support code generation	Toolbox functions that you rewrite for desktop and embedded applications
Deploy MATLAB based GUI applications on a supported MATLAB host	MATLAB Compiler
Deploy web-based or Windows® applications	<ul style="list-style-type: none"> <li>• MATLAB Builder NE</li> <li>• MATLAB Builder JA</li> </ul>



<b>To:</b>	<b>Use:</b>
Interface C code with MATLAB	MATLAB <code>mex</code> function

## Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	<code>codegen</code> function	MATLAB Coder	Try this in “MEX Function Generation at the Command Line”.
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder user interface	MATLAB Coder	Try this in “C Code Generation Using the Project Interface”.
	<code>codegen</code> function	MATLAB Coder	Try this in “C Code Generation at the Command Line”.
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder user interface	MATLAB Coder	See “Accelerate MATLAB Algorithms”.
	<code>codegen</code> function	MATLAB Coder	
Integrate MATLAB code into Simulink®	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code”.
Speed up fixed-point MATLAB code	<code>fiaccel</code> function	Fixed-Point Designer	Learn more in “Code Acceleration and Code Generation from MATLAB”.
Integrate custom C code into MATLAB and generate efficient, readable code	<code>codegen</code> function	MATLAB Coder	Learn more in “Specify External File Locations”.
Integrate custom C code into code generated from MATLAB	<code>coder.ceval</code> function	MATLAB Coder	Learn more in <code>coder.ceval</code> .
Generate HDL from MATLAB code	MATLAB Function block	Simulink and HDL Coder™	Learn more at <a href="http://www.mathworks.com/products/slhdlcoder">www.mathworks.com/products/slhdlcoder</a> .

## Prerequisites for C/C++ Code Generation from MATLAB

To generate C/C++ or MEX code from MATLAB algorithms, you must install the following software:

- MATLAB Coder product
- C/C++ compiler

# MATLAB Code Design Considerations for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

You can choose whether the generated code uses static or dynamic memory allocation.

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get better speed, but with higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- Speed

Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

To improve the speed of the generated code:

- Choose a suitable C/C++ compiler. Do not use the default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms.
- Consider disabling run-time checks.

By default, for safety, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower simulation. Disabling run-time checks usually results in streamlined generated code and faster simulation. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

## **See Also**

- “Data Definition Basics”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data”
- “Control Dynamic Memory Allocation”
- “Control Run-Time Checks”

## Differences in Behavior After Compiling MATLAB Code

In this section...
“Why Are There Differences?” on page 2-8
“Character Size” on page 2-8
“Order of Evaluation in Expressions” on page 2-8
“Termination Behavior” on page 2-9
“Size of Variable-Size N-D Arrays” on page 2-9
“Size of Empty Arrays” on page 2-9
“Floating-Point Numerical Results” on page 2-10
“NaN and Infinity Patterns” on page 2-10
“Code Generation Target” on page 2-11
“MATLAB Class Initial Values” on page 2-11
“Variable-Size Support for Code Generation” on page 2-11

### Why Are There Differences?

To convert MATLAB code to C/C++ code that works efficiently, the code generation process introduces optimizations that intentionally cause the generated code to behave differently — and sometimes produce different results — from the original source code. This section describes these differences.

### Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Code Generation for Characters” on page 6-6.

### Order of Evaluation in Expressions

Generated code does not enforce order of evaluation in expressions. For most expressions, order of evaluation is not significant. However, for expressions with side effects, the generated code may produce the side effects in different order from the original MATLAB code. Expressions that produce side effects include those that:

- Modify persistent or global variables

- Display data to the screen
- Write data to files
- Modify the properties of handle class objects

In addition, the generated code does not enforce order of evaluation of logical operators that do not short circuit.

For more predictable results, it is good coding practice to split expressions that depend on the order of evaluation into multiple statements. For example, rewrite:

```
A = f1() + f2();
```

as

```
A = f1();  
A = A + f2();
```

so that the generated code calls `f1` before `f2`.

## Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, optimizations remove infinite loops from generated code if they do not have side effects. As a result, the generated code may terminate even though the corresponding MATLAB code does not.

## Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function might return a different result in generated code than in MATLAB source code. The `size` function sometimes returns trailing ones (singleton dimensions) in generated code, but always drops trailing ones in MATLAB. For example, for an N-D array `X` with dimensions `[4 2 1 1]`, `size(X)` might return `[4 2 1 1]` in generated code, but always returns `[4 2]` in MATLAB. See “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-27.

## Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. See “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 7-28.

### **Floating-Point Numerical Results**

The generated code might not produce the same floating-point numerical results as MATLAB in the following situations:

#### **When computer hardware uses extended precision registers**

Results vary depending on how the C/C++ compiler allocates extended precision floating-point registers. Computation results might not match MATLAB calculations because of different compiler optimization settings or different code surrounding the floating-point calculations.

#### **For certain advanced library functions**

The generated code might use different algorithms to implement certain advanced library functions, such as `fft`, `svd`, `eig`, `mldivide`, and `mrdivide`.

For example, the generated code uses a simpler algorithm to implement `svd` to accommodate a smaller footprint. Results might also vary according to matrix properties. For example, MATLAB might detect symmetric or Hermitian matrices at run time and switch to specialized algorithms that perform computations faster than implementations in the generated code.

#### **For implementation of BLAS library functions**

For implementations of BLAS library functions. Generated C/C++ code uses reference implementations of BLAS functions, which may produce different results from platform-specific BLAS implementations in MATLAB.

#### **NaN and Infinity Patterns**

The generated code might not produce exactly the same pattern of NaN and `inf` values as MATLAB code when these values are mathematically meaningless. For example, if MATLAB output contains a NaN, output from the generated code should also contain a NaN, but not necessarily in the same place.



## Code Generation Target

The `coder.target` function returns different values in MATLAB than in the generated code. The intent is to help you determine whether your function is executing in MATLAB or has been compiled for a simulation or code generation target. See `coder.target`.

## MATLAB Class Initial Values

MATLAB computes class initial values at class loading time before code generation. The code generation software uses the value that MATLAB computed, it does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generation software does not execute this function. If the function modifies a global state, for example, a persistent variable, code generation software might provide a different initial value than MATLAB. For more information, see “Defining Class Properties for Code Generation”.

## Variable-Size Support for Code Generation

For incompatibilities with MATLAB in variable-size support for code generation, see:

- “Incompatibility with MATLAB for Scalar Expansion”
- “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays”
- “Incompatibility with MATLAB in Determining Size of Empty Arrays”
- “Incompatibility with MATLAB in Vector-Vector Indexing”
- “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation”

# MATLAB Language Features Supported for C/C++ Code Generation

MATLAB supports the following language features in generated code:

- N-dimensional arrays (see “Array Size Restrictions for Code Generation” on page 6-7)
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “Variable-Size Data Definition for Code Generation” on page 7-3)
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-30)
- Complex numbers (see “Code Generation for Complex Data” on page 6-4)
- Numeric classes (see “Supported Variable Types” on page 5-17)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB”)
- Program control statements `if`, `switch`, `for`, `while`, and `break`
- Arithmetic, relational, and logical operators
- Local functions
- Persistent variables (see “Define and Initialize Persistent Variables” on page 5-10)
- Global variables (see “Specifying Global Variable Type and Initial Value in a Project”).
- Structures
- Characters (see “Code Generation for Characters” on page 6-6)
- Function handles
- Frames
- Variable length input and output argument lists
- Subset of MATLAB toolbox functions
- MATLAB classes
- Ability to call functions (see “Resolution of Function Calls for Code Generation” on page 13-2)

# MATLAB Language Features Not Supported for C/C++ Code Generation

MATLAB does not support the following features in generated code:

- Anonymous functions
- Cell arrays
- Java
- Nested functions
- Recursion
- Sparse matrices
- try/catch statements



# System Objects Supported for Code Generation

---

## Code Generation for System Objects

You can generate C and C++ code for a subset of System objects provided by the following toolboxes.

Toolbox Name	See
Communications System Toolbox	“System Objects in MATLAB Code Generation” in the DSP System Toolbox documentation.
Computer Vision System Toolbox	“System Objects in MATLAB Code Generation” in the Computer Vision System Toolbox documentation.
DSP System Toolbox	“System Objects in MATLAB Code Generation” in the DSP System Toolbox documentation.
Image Acquisition Toolbox™	<ul style="list-style-type: none"><li>• <code>imaq.VideoDevice</code>.</li><li>• “Code Generation with VideoDevice System Object” in the Image Acquisition Toolbox documentation.</li></ul>
Phased Array System Toolbox	“Code Generation” in the Phased Array System Toolbox documentation.

To use these System objects, you need to install the requisite toolbox. For a list of System objects supported for C and C++ 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 — Category List”.

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets. For general information about MATLAB objects, see “Begin Using Object-Oriented Programming”.

# Functions, Classes, and System Objects Supported for Code Generation

---

- “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List” on page 4-2
- “Functions and Objects Supported for C and C++ Code Generation — Category List” on page 4-134

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

You can generate efficient C and C++ code for a subset of MATLAB built-in functions and toolbox functions, classes, and System objects that you call from MATLAB code. These function, classes, and System objects appear in alphabetical order in the following table.

To find supported functions, classes, and System objects by MATLAB category or toolbox, see “Functions and Objects Supported for C and C++ Code Generation — Category List”.

---

**Note:** For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB”.

---

Name	Product	Remarks and Limitations
abs	MATLAB	—
abs	Fixed-Point Designer	—
accumneg	Fixed-Point Designer	—
accumpos	Fixed-Point Designer	—
acos	MATLAB	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
acosd	MATLAB	—
acosh	MATLAB	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>



Name	Product	Remarks and Limitations
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
add	Fixed-Point Designer	Code generation in MATLAB does not support the syntax <code>F.add(a,b)</code> . You must use the syntax <code>add(F,a,b)</code> .
affine2d	Image Processing Toolbox™	When generating code, you can only specify single objects—arrays of objects are not supported.
aicctest	Phased Array System Toolbox	Does not support variable-size inputs.
albersheim	Phased Array System Toolbox	Does not support variable-size inputs.
all	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
all	Fixed-Point Designer	—
ambgfun	Phased Array System Toolbox	Does not support variable-size inputs.
and	MATLAB	—
any	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
any	Fixed-Point Designer	—
aperture2gain	Phased Array System Toolbox	Does not support variable-size inputs.
asec	MATLAB	—
asecd	MATLAB	—

Name	Product	Remarks and Limitations
asech	MATLAB	—
asin	MATLAB	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
asind	MATLAB	—
asinh	MATLAB	—
assert	MATLAB	<ul style="list-style-type: none"> <li>Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time.</li> <li>For standalone code generation, excluded from the generated code.</li> <li>See “Rules for Using assert Function”.</li> </ul>
assignDetections-ToTracks	Computer Vision System Toolbox	Compile-time constant input: No restriction. Supports MATLAB Function block: Yes
atan	MATLAB	—
atan2	MATLAB	—
atan2	Fixed-Point Designer	—
atan2d	MATLAB	—
atand	MATLAB	—
atanh	MATLAB	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
az2broadside	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
azel2phitheta	Phased Array System Toolbox	Does not support variable-size inputs.
azel2phithetapat	Phased Array System Toolbox	Does not support variable-size inputs.
azel2uv	Phased Array System Toolbox	Does not support variable-size inputs.
azel2uvpat	Phased Array System Toolbox	Does not support variable-size inputs.
azelaxes	Phased Array System Toolbox	Does not support variable-size inputs.
barthannwin	Signal Processing Toolbox™	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
bartlett	Signal Processing Toolbox	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
bboxOverlapRatio	Computer Vision System Toolbox	<p>Compile-time constant input: No restriction</p> <p>Supports MATLAB Function block: No</p>
beat2range	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
besselap	Signal Processing Toolbox	<p>Filter order must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
beta	MATLAB	—
betacdf	Statistics Toolbox™	—
betainc	MATLAB	Always returns a complex result.
betaincinv	MATLAB	Always returns a complex result.
betainv	Statistics Toolbox	—
betaln	MATLAB	—
betapdf	Statistics Toolbox	—
betarnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
betastat	Statistics Toolbox	—
bi2de	Communications System Toolbox	—
billingsleyicm	Phased Array System Toolbox	Does not support variable-size inputs.
bin2dec	MATLAB	<ul style="list-style-type: none"> <li>• Does not match MATLAB when the input is empty.</li> </ul>
binaryFeatures	Computer Vision System Toolbox	—

Name	Product	Remarks and Limitations
binocdf	Statistics Toolbox	—
binoinv	Statistics Toolbox	—
binopdf	Statistics Toolbox	—
binornd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
binostat	Statistics Toolbox	—
bitand	MATLAB	—
bitand	Fixed-Point Designer	• Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	Fixed-Point Designer	—
bitcmp	MATLAB	—
bitcmp	Fixed-Point Designer	—
bitconcat	Fixed-Point Designer	—
bitget	MATLAB	—
bitget	Fixed-Point Designer	—
bitmax	MATLAB	—
bitor	MATLAB	—
bitor	Fixed-Point Designer	• Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	Fixed-Point Designer	—
bitreplicate	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
bitrevorder	Signal Processing Toolbox	—
bitrol	Fixed-Point Designer	—
bitror	Fixed-Point Designer	—
bitset	MATLAB	—
bitset	Fixed-Point Designer	—
bitshift	MATLAB	—
bitshift	Fixed-Point Designer	—
bitsliceget	Fixed-Point Designer	—
bitsll	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Generated code may not handle out of range shifting.</li> </ul>
bitsra	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Generated code may not handle out of range shifting.</li> </ul>
bitsrl	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Generated code may not handle out of range shifting.</li> </ul>
bitxor	MATLAB	—
bitxor	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Not supported for slope-bias scaled <code>fi</code> objects.</li> </ul>
bitxorreduce	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
blackman	Signal Processing Toolbox	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
blackmanharris	Signal Processing Toolbox	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
blanks	MATLAB	—
blkdiag	MATLAB	—
bohmanwin	Signal Processing Toolbox	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
break	MATLAB	—

Name	Product	Remarks and Limitations
BRISKPoints	Computer Vision System Toolbox	<p>Compile-time constant inputs: No restriction</p> <p>Supports MATLAB Function block: No</p> <p>To index locations with this object, use the syntax: <code>points.Location(idx, :)</code>, for <code>points</code> object. See <code>visionRecoverFromCodeGeneration_kernel.m</code>, which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.</p>
broadside2az	Phased Array System Toolbox	Does not support variable-size inputs.
bsxfun	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
buttap	Signal Processing Toolbox	<p>Filter order must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
butter	Signal Processing Toolbox	<p>Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>



Name	Product	Remarks and Limitations
buttord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
bwdist	Image Processing Toolbox	<p>The <code>method</code> argument must be a compile-time constant. Input images must have fewer than <math>2^{32}</math> pixels.</p> <p>Generated code for this function uses a precompiled, “platform-specific shared library”.</p>
bwlookup	Image Processing Toolbox	<ul style="list-style-type: none"> <li>For best results, specify an input image of class <code>logical</code>.</li> </ul>
bwmorph	Image Processing Toolbox	<ul style="list-style-type: none"> <li>The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code>.</li> </ul>
bwpack	Image Processing Toolbox	<p>Generated code for this function uses a precompiled platform-specific shared library.</p>
bwselect	Image Processing Toolbox	<p>Supports only the 3 and 4 input argument syntaxes: <code>BW2 = bwselect(BW,c,r)</code> and <code>BW2 = bwselect(BW,c,r,n)</code>. The optional fourth input argument, <code>n</code>, must be a compile-time constant. In addition, with code generation, <code>bwselect</code> only supports only the 1 and 2 output argument syntaxes: <code>BW2 = bwselect(___)</code> or <code>[BW2, idx] = bwselect(___)</code>.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Name	Product	Remarks and Limitations
bwtraceboundary	Image Processing Toolbox	The <code>dir</code> , <code>fstep</code> , and <code>conn</code> arguments must be compile-time constants.
bwunpack	Image Processing Toolbox	Generated code for this function uses a precompiled platform-specific shared library.
ca2tf	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
cart2pol	MATLAB	—
cart2sph	MATLAB	—
cart2sphvec	Phased Array System Toolbox	Does not support variable-size inputs.
cast	MATLAB	—
cat	MATLAB	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
cbfweights	Phased Array System Toolbox	Does not support variable-size inputs.
cdf	Statistics Toolbox	—
ceil	MATLAB	—
ceil	Fixed-Point Designer	—
cfirpm	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
char	MATLAB	—

Name	Product	Remarks and Limitations
cheb1ap	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
cheb1ord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
cheb2ap	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Name	Product	Remarks and Limitations
cheb2ord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
chebwin	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
cheby1	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Name	Product	Remarks and Limitations
cheby2	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
chi2cdf	Statistics Toolbox	—
chi2inv	Statistics Toolbox	—
chi2pdf	Statistics Toolbox	—
chi2rnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
chi2stat	Statistics Toolbox	—
chol	MATLAB	—
circpol2pol	Phased Array System Toolbox	Does not support variable-size inputs.
circshift	MATLAB	—
cl2tf	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
class	MATLAB	—

Name	Product	Remarks and Limitations
colon	MATLAB	<ul style="list-style-type: none"> <li>• Does not accept complex inputs.</li> <li>• The input <code>i</code> cannot have a logical value.</li> <li>• Does not accept vector inputs.</li> <li>• Inputs must be constants.</li> <li>• Uses single-precision arithmetic to produce single-precision results.</li> </ul>
comm.ACPR	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.AGC	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Algebraic-Deinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.APPDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.AWGNChannel	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BarkerCode	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BCHDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BCHEncoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Binary-SymmetricChannel	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BlockDeinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BlockInterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BPSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.BPSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"

Name	Product	Remarks and Limitations
comm.CCDF	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Constellation-Diagram	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Convolutional-Deinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Convolutional-Encoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Convolutional-Interleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CPFSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CPFSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CPMCarrier-PhaseSynchronizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CPMDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CPModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CRCDetector	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.CRCGenerator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DBPSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DBPSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Descrambler	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Differential-Decoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"

Name	Product	Remarks and Limitations
comm.Differential-Encoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DiscreteTimeVCO	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DPSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DPSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DQPSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DQPSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.EarlyLateGate-TimingSynchronizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.ErrorRate	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.EVM	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.FSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.FSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GardnerTiming-Synchronizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GeneralQAM-Demodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GeneralQAM-Modulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GeneralQAMTCM-Demodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GeneralQAMTCM-Modulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"



Name	Product	Remarks and Limitations
comm.GMSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GMSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GMSKTiming-Synchronizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.GoldSequence	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.HadamardCode	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.HDLCRCDetector	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.HDLCRCGenerator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.HDLRSDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.HDLRSEncoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Helical-Deinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.HelicalInterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.IntegrateAnd-DumpFilter	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.IQImbalance-Compensator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.KasamiSequence	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.LDPCDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.LDPCEncoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"

Name	Product	Remarks and Limitations
comm.LTEMIMOChannel	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Matrix-Deinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MatrixHelical-ScanDeinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MatrixHelical-ScanInterLeaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MatrixInterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Memoryless-Nonlinearity	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MER	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MIMOChannel	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MLSEEqualizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MSKTiming-Synchronizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.MuellerMuller-TimingSynchronizer	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Multiplexed-Deinterleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Multiplexed-Interleaver	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.OFDMDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"

Name	Product	Remarks and Limitations
comm.OFDMModulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.OSTBCCombiner	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.OSTBCEncoder	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.OQPSKDemodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.OQPSKModulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PAMDemodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PAMModulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PhaseRequency-Offset	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PhaseNoise	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PNSequence	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PSKCoarseFrequencyEstimator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PSKCoarseFrequencyEstimator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PSKDemodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PSKModulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PSKTCMDemodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.PSKTCMModulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
comm.QAMCoarseFrequencyEstimator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.QPSKDemodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.QPSKModulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RaisedCosine-ReceiveFilter	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RaisedCosine-TransmitFilter	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RayleighChannel	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RectangularQAM-Demodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Rectangular-Modulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RectangularQAMTCM-Demodulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RectangularQAMTCM-Modulator	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RicianChannel	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RSDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.RSEncoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.Scrambler	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.SphereDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.ThermalNoise	Communications System Toolbox	"System Objects in MATLAB Code Generation"

Name	Product	Remarks and Limitations
comm.TurboDecoder	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.TurboEncoder	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.ViterbiDecoder	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.WalshCode	Communications System Toolbox	“System Objects in MATLAB Code Generation”
compan	MATLAB	—
complex	MATLAB	—
complex	Fixed-Point Designer	—
computer	MATLAB	<ul style="list-style-type: none"> <li>• Information about the computer on which the code generation software is running.</li> <li>• Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Designer	—
conndef	Image Processing Toolbox	All input arguments must be compile-time constants.
continue	MATLAB	—
conv	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”

Name	Product	Remarks and Limitations
conv	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.</li> <li>• For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> <li>• In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>.</li> <li>• In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.</li> </ul> </li> </ul>
conv2	MATLAB	—
convergent	Fixed-Point Designer	—
convn	MATLAB	—
cordicabs	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-size signals are not supported.</li> </ul>
cordicangle	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-size signals are not supported.</li> </ul>
cordicatan2	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-size signals are not supported.</li> </ul>
cordiccart2pol	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-size signals are not supported.</li> </ul>
cordicexp	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-size signals are not supported.</li> </ul>

Name	Product	Remarks and Limitations
cordiccos	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Variable-size signals are not supported.</li> </ul>
cordicpol2cart	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Variable-size signals are not supported.</li> </ul>
cordicrotate	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Variable-size signals are not supported.</li> </ul>
cordicsin	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Variable-size signals are not supported.</li> </ul>
cordicsincos	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Variable-size signals are not supported.</li> </ul>
cornerPoints	Computer Vision System Toolbox	<p>Compile-time constant input: No restriction</p> <p>Supports MATLAB Function block: No</p> <p>To index locations with this object, use the syntax: <code>points.Location(idx, :)</code>, for <code>points</code> object. See <code>visionRecovertransformCodeGeneration_kernel.m</code>, which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.</p>
corrcoef	MATLAB	<ul style="list-style-type: none"> <li>Row-vector input is only supported when the first two inputs are vectors and nonscalar.</li> </ul>
cos	MATLAB	—
cos	Fixed-Point Designer	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	<ul style="list-style-type: none"> <li>In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
coth	MATLAB	—

Name	Product	Remarks and Limitations
cov	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
cross	MATLAB	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
csc	MATLAB	—
cscd	MATLAB	<ul style="list-style-type: none"> <li>• In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>• In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
csch	MATLAB	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Designer	—
cumprod	MATLAB	<ul style="list-style-type: none"> <li>• Does not support logical inputs. Cast input to <code>double</code> first.</li> <li>• Does not support the <code>direction</code> argument.</li> </ul>
cumsum	MATLAB	<ul style="list-style-type: none"> <li>• Does not support logical inputs. Cast input to <code>double</code> first.</li> <li>• Does not support the <code>direction</code> argument.</li> </ul>
cumtrapz	MATLAB	—
db2pow	Signal Processing Toolbox	—



Name	Product	Remarks and Limitations
dct	Signal Processing Toolbox	<ul style="list-style-type: none"> <li>• Code generation for this function requires the DSP System Toolbox software.</li> <li>• Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.</li> </ul> <p style="text-align: center;"><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
de2bi	Communications System Toolbox	—
deal	MATLAB	—
deblank	MATLAB	<ul style="list-style-type: none"> <li>• Supports only inputs from the <code>char</code> class.</li> <li>• Input values must be in the range 0-127.</li> </ul>
dec2bin	MATLAB	<ul style="list-style-type: none"> <li>• If input <code>d</code> is <code>double</code>, <code>d</code> must be less than <math>2^{52}</math>.</li> <li>• If input <code>d</code> is <code>single</code>, <code>d</code> must be less than <math>2^{23}</math>.</li> <li>• Unless you specify input <code>n</code> to be constant and <code>n</code> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <code>n</code> must be at least 52 for <code>double</code>, 23 for <code>single</code>, 16 for <code>char</code>, 32 for <code>int32</code>, 16 for <code>int16</code>, and so on.</li> </ul>

Name	Product	Remarks and Limitations
dec2hex	MATLAB	<ul style="list-style-type: none"> <li>• If input <code>d</code> is <code>double</code>, <code>d</code> must be less than <math>2^{52}</math>.</li> <li>• If input <code>d</code> is <code>single</code>, <code>d</code> must be less than <math>2^{23}</math>.</li> <li>• Unless you specify input <code>n</code> to be constant and <code>n</code> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <code>n</code> must be at least 13 for <code>double</code>, 6 for <code>single</code>, 4 for <code>char</code>, 8 for <code>int32</code>, 4 for <code>int16</code>, and so on.</li> </ul>
dechirp	Phased Array System Toolbox	Does not support variable-size inputs.
deconv	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
del2	MATLAB	—
delayseq	Phased Array System Toolbox	Does not support variable-size inputs.
depressionang	Phased Array System Toolbox	Does not support variable-size inputs.
det	MATLAB	—
detectBRISKFeatures	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectFASTFeatures	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectHarrisFeatures	Computer Vision System Toolbox	Compile-time constant input: <code>FilterSize</code> Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.

Name	Product	Remarks and Limitations
detectMinEigenFeatures	Computer Vision System Toolbox	Compile-time constant input: FilterSize Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectMSERFeatures	Computer Vision System Toolbox	Compile-time constant input: No restriction Supports MATLAB Function block: No For code generation, the function outputs regions.PixelList as an array. The region sizes are defined in regions.Lengths.
detectSURFFeatures	Computer Vision System Toolbox	Compile-time constant input: No restrictions Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detrend	MATLAB	<ul style="list-style-type: none"> <li>• If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> <li>• Be real.</li> <li>• Be sorted in ascending order.</li> <li>• Restrict elements to integers in the interval <math>[1, n-2]</math>. <math>n</math> is the number of elements in a column of input argument <math>X</math>, or the number of elements in <math>X</math> when <math>X</math> is a row vector.</li> <li>• Contain all unique values.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul> </li> </ul>

Name	Product	Remarks and Limitations
diag	MATLAB	<ul style="list-style-type: none"> <li>• If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.</li> <li>• For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), <b>diag</b>:               <ul style="list-style-type: none"> <li>• Treats the input as a vector input.</li> <li>• Returns a matrix with the given vector along the specified diagonal.</li> </ul> </li> <li>• For variable-size inputs that are not variable-length vectors, <b>diag</b>:               <ul style="list-style-type: none"> <li>• Treats the input as a matrix.</li> <li>• Does not support inputs that are vectors at run time.</li> <li>• Returns a variable-length vector.</li> </ul> <p>If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, the output is 0-by-1 not 0-by-0. However, if the input is a constant size 0-by-0, the output is [ ].</p> </li> <li>• For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), <b>diag</b> treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force <b>diag</b> to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use:               <ul style="list-style-type: none"> <li>• <b>diag(x(:))</b> instead of <b>diag(x)</b></li> <li>• <b>diag(x(:),k)</b> instead of <b>diag(x,k)</b></li> </ul> </li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
diag	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• If supplied, the index, <i>k</i>, must be a real and scalar integer value that is not a <b>fi</b> object.</li> </ul>

Name	Product	Remarks and Limitations
diff	MATLAB	<ul style="list-style-type: none"> <li>• If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
disparity	Computer Vision System Toolbox	Compile-time constant input: Method. Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
divide	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Any non-<code>fi</code> input must be constant. Its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> <li>• Complex and imaginary divisors are not supported.</li> <li>• The syntax <code>T.divide(a,b)</code> is not supported.</li> </ul>
dop2speed	Phased Array System Toolbox	Does not support variable-size inputs.
dopsteeringvec	Phased Array System Toolbox	Does not support variable-size inputs.
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Designer	—
downsample	Signal Processing Toolbox	—

Name	Product	Remarks and Limitations
dpss	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
dsp.AdaptiveLattice-Filter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AffineProjection-Filter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AllpoleFilter	DSP System Toolbox	<ul style="list-style-type: none"> <li>• “System Objects in MATLAB Code Generation”</li> <li>• Only the <code>Denominator</code> property is tunable for code generation.</li> </ul>
dsp.AnalyticSignal	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ArrayPlot	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ArrayVectorAdder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ArrayVectorDivider	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ArrayVector-Multiplier	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ArrayVector-Subtractor	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AudioFileReader	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.AudioRecorder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AudioFileWriter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AudioPlayer	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Autocorrelator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.BiquadFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.BurgAREstimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.BurgSpectrum-Estimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CepstralToLPC	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CICCompensation-Decimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CICCompensation-Interpolator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CICDecimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CICInterpolator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Convolver	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Counter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Crosscorrelator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CrossSpectrum-Estimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.CumulativeProduct	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CumulativeSum	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.DCBlocker	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.DCT	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Delay	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.DelayLine	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.DigitalDown-Converter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.DigitalUpConverter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.DigitalFilter	DSP System Toolbox	<ul style="list-style-type: none"> <li>• “System Objects in MATLAB Code Generation”</li> <li>• The <code>SOSMatrix</code> and <code>Scalevalues</code> properties are not supported for code generation.</li> </ul>
dsp.FarrowRateConverter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FastTransversal-Filter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FFT	DSP System Toolbox	“System Objects in MATLAB Code Generation”



Name	Product	Remarks and Limitations
dsp.FilterCascade	DSP System Toolbox	<ul style="list-style-type: none"> <li>• You cannot generate code directly from <code>dsp.FilterCascade</code>. You can use the <code>generateFilteringCode</code> method to generate a MATLAB function. You can generate C/C++ code from this MATLAB function.</li> </ul> <p>“System Objects in MATLAB Code Generation”</p>
dsp.FilteredXLMSFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FIRDecimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FIRFilter	DSP System Toolbox	<ul style="list-style-type: none"> <li>• “System Objects in MATLAB Code Generation”</li> <li>• Only the <code>Numerator</code> property is tunable for code generation.</li> </ul>
dsp.FIRHalfband-Decimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FIRHalfband-Interpolator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FIRInterpolator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FIRRateConverter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FrequencyDomain-AdaptiveFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Histogram	DSP System Toolbox	<ul style="list-style-type: none"> <li>• This object has no tunable properties for code generation.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.IDCT	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.IFFT	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.IIRFilter	DSP System Toolbox	<ul style="list-style-type: none"> <li>• Only the Numerator and Denominator properties are tunable for code generation.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.Interpolator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.KalmanFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LDLFactor	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LevinsonSolver	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LMSFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LowerTriangular-Solver	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LPCToAuto-correlation	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LPCToCepstral	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LPCToLSF	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LPCToLSP	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LPCToRC	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LSFToLPC	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LSPToLPC	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.LUFactor	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Maximum	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Mean	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Median	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Minimum	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.NCO	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Normalizer	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.PeakFinder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.PeakToPeak	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.PeakToRMS	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.PhaseExtractor	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.PhaseUnwrapper	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.RCToAutocorrelation	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.RCToLPC	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.RMS	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.RLSFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
<code>dsp.SampleRateConverter</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.ScalarQuantizer-Decoder</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.ScalarQuantizer-Encoder</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.SignalSource</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.SineWave</code>	DSP System Toolbox	<ul style="list-style-type: none"> <li>This object has no tunable properties for code generation.</li> <li>“System Objects in MATLAB Code Generation”</li> </ul>
<code>dsp.SpectrumAnalyzer</code>	DSP System Toolbox	This System object™ does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
<code>dsp.SpectrumEstimator</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.StandardDeviation</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.StateLevels</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.TimeScope</code>	DSP System Toolbox	This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
<code>dsp.TransferFunction-Estimator</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.UDPReceiver</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.UDPSender</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”
<code>dsp.UpperTriangular-Solver</code>	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.VariableFraction-Delay	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VariableInteger-Delay	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Variance	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VectorQuantizer-Decoder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VectorQuantizer-Encoder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Window	DSP System Toolbox	<ul style="list-style-type: none"> <li>• This object has no tunable properties for code generation.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.ZeroCrossing-Detector	DSP System Toolbox	“System Objects in MATLAB Code Generation”
edge	Image Processing Toolbox	<p>The <code>method</code>, <code>direction</code>, and <code>sigma</code> arguments must be a compile-time constants. In addition, nonprogrammatic syntaxes are not supported. For example, the syntax <code>edge(im)</code>, where <code>edge</code> does not return a value but displays an image instead, is not supported.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
effearthradius	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
eig	MATLAB	<ul style="list-style-type: none"> <li>• For code generation, QZ algorithm is used in all cases. MATLAB can use different algorithms for different inputs. Consequently, <math>V</math> might represent a different basis of eigenvectors. The eigenvalues in <math>D</math> might not be in the same order as in MATLAB.</li> <li>• With one input, <math>[V,D] = \text{eig}(A)</math>, the results are similar to those obtained using <math>[V,D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')</math> in MATLAB, except that for code generation, the columns of <math>V</math> are normalized.</li> <li>• Options 'balance', and 'nobalance' are not supported for the standard eigenvalue problem. 'chol' is not supported for the symmetric generalized eigenvalue problem.</li> <li>• Outputs are of complex type.</li> <li>• Does not support the option to calculate left eigenvectors.</li> </ul>
ellip	Signal Processing Toolbox	<p>Inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Name	Product	Remarks and Limitations
ellipap	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
ellipke	MATLAB	—
ellipord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
end	MATLAB	—
end	Fixed-Point Designer	—
epipolarLine	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes</p>
eps	MATLAB	—
eps	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Supported for scalar fixed-point signals only.</li> <li>Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.</li> </ul>
eq	MATLAB	—
eq	Fixed-Point Designer	Not supported for fixed-point signals with different biases.

Name	Product	Remarks and Limitations
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—
error	MATLAB	For standalone code generation, excluded from the generated code.
espritdoa	Phased Array System Toolbox	Does not support variable-size inputs.
estimateFundamental-Matrix	Computer Vision System Toolbox	Compile-time constant input: Method, OutputClass, DistanceType, and ReportRuntimeError. Supports MATLAB Function block: Yes
estimateGeometric-Transform	Computer Vision System Toolbox	Compile-time constant input: transformType Supports MATLAB Function block: No
estimateUncalibrated-Rectification	Computer Vision System Toolbox	Compile-time constant input: transformType Supports MATLAB Function block: No
extractFeatures	Computer Vision System Toolbox	Generates platform-dependent library: Yes for BRISK, FREAK, and SURF methods only. Compile-time constant input: Method Supports MATLAB Function block: Yes for Block method only. Generated code for this function uses a precompiled platform-specific shared library.
extractHOGFeatures	Computer Vision System Toolbox	Compile-time constant input: No Supports MATLAB Function block: No
evcdf	Statistics Toolbox	—
evinv	Statistics Toolbox	—
evpdf	Statistics Toolbox	—



Name	Product	Remarks and Limitations
evrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
evstat	Statistics Toolbox	—
exp	MATLAB	—
expcdf	Statistics Toolbox	—
expint	MATLAB	—
expinv	Statistics Toolbox	—
expm	MATLAB	—
expm1	MATLAB	—
exppdf	Statistics Toolbox	—
exprnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
expstat	Statistics Toolbox	—
eye	MATLAB	<code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>eye</code> method for other classes. For example, <code>eye(m, n, 'myclass')</code> does not invoke <code>myclass.eye(m,n)</code> .
factor	MATLAB	<ul style="list-style-type: none"> <li>• The maximum double precision input is <math>2^{33}</math>.</li> <li>• The maximum single precision input is <math>2^{25}</math>.</li> <li>• The input <code>n</code> cannot have type <code>int64</code> or <code>uint64</code>.</li> </ul>
factorial	MATLAB	—

Name	Product	Remarks and Limitations
false	MATLAB	<ul style="list-style-type: none"> <li>• Dimensions must be real, nonnegative, integers.</li> </ul>
fcdf	Statistics Toolbox	—
fclose	MATLAB	—
feof	MATLAB	—
fft	MATLAB	<ul style="list-style-type: none"> <li>• Length of input vector must be a power of 2.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
fft2	MATLAB	<ul style="list-style-type: none"> <li>• Length of input matrix dimensions must each be a power of 2.</li> </ul>
fftn	MATLAB	<ul style="list-style-type: none"> <li>• Length of input matrix dimensions must each be a power of 2.</li> </ul>
fftshift	MATLAB	—
fi	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Use to create a fixed-point constant or variable.</li> <li>• The default constructor syntax without input arguments is not supported.</li> <li>• The rand <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v,'PropertyName',PropertyValue...)</code>.</li> <li>• If the input value is not known at compile time, you must provide complete <code>numericType</code> information.</li> <li>• All properties related to data type must be constant for code generation.</li> <li>• <code>numericType</code> object information must be available for non-fixed-point Simulink inputs.</li> </ul>

Name	Product	Remarks and Limitations
filter	MATLAB	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
filter	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.</li> </ul>
filter2	MATLAB	—
filtfilt	Signal Processing Toolbox	<p>Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
fimath	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned the <code>fimath</code> object defined in the MATLAB Function dialog in the Model Explorer.</li> <li>• Use to create <code>fimath</code> objects in generated code.</li> <li>• If the <code>ProductMode</code> property of the <code>fimath</code> object is set to anything other than <code>FullPrecision</code>, the <code>ProductWordLength</code> and <code>ProductFractionLength</code> properties must be constant.</li> <li>• If the <code>SumMode</code> property of the <code>fimath</code> object is set to anything other than <code>FullPrecision</code>, the <code>SumWordLength</code> and <code>SumFractionLength</code> properties must be constant.</li> </ul>

Name	Product	Remarks and Limitations
find	MATLAB	<ul style="list-style-type: none"> <li>Issues an error if a variable-sized input becomes a row vector at run time.</li> </ul> <hr/> <p><b>Note:</b> This limitation does not apply when the input is scalar or a variable-length row vector.</p> <hr/> <ul style="list-style-type: none"> <li>For variable-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or [] at run time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.</li> </ul>
findpeaks	Signal Processing Toolbox	—
finv	Statistics Toolbox	—
fir1	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Name	Product	Remarks and Limitations
fir2	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
firceqrip	DSP System Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p>
fircls	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
fircls1	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Name	Product	Remarks and Limitations
fireqint	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firgr	DSP System Toolbox	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
firhalfband	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firlpnorm	DSP System Toolbox	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
firls	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
firminphase	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firnyquist	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Product	Remarks and Limitations
firpr2chfb	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firpm	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
firpmord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
fitgeotrans	Image Processing Toolbox	The <code>transformtype</code> argument must be a compile-time constant. The function supports the following transformation types: 'nonreflectivesimilarity', 'similarity', 'affine', or 'projective'.
fix	MATLAB	—
fix	Fixed-Point Designer	—
fixed.Quantizer	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
flattopwin	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
flintmax	MATLAB	—
flip	MATLAB	—
flip	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
flipdim	MATLAB	<b>Note:</b> <code>flipdim</code> will be removed in a future release. Use <code>flip</code> instead.
fliplr	MATLAB	—
fliplr	Fixed-Point Designer	—
flipud	MATLAB	—
flipud	Fixed-Point Designer	—
floor	MATLAB	—
floor	Fixed-Point Designer	—
fminsearch	MATLAB	<ul style="list-style-type: none"> <li>• Ignores the <code>Display</code> option. Does not print status information during execution. Test the <code>exitflag</code> output for the exit condition.</li> <li>• The output structure does not include the <code>algorithm</code> or <code>message</code> fields.</li> <li>• Ignores the <code>OutputFcn</code> and <code>PlotFcns</code> options.</li> </ul>



Name	Product	Remarks and Limitations
fopen	MATLAB	<ul style="list-style-type: none"> <li>• Does not support: <ul style="list-style-type: none"> <li>• machineformat, encoding, or fileID inputs</li> <li>• message output</li> <li>• fopen('all')</li> </ul> </li> <li>• If you disable extrinsic calls, you cannot return fileIDs created with fopen to MATLAB or extrinsic functions. You can use such fileIDs only internally.</li> <li>• When generating C/C++ executables, static libraries, or dynamic libraries, you can open up to 20 files.</li> <li>• The generated code does not report errors from invalid file identifiers. Write your own file open error handling in your MATLAB code. Test whether fopen returns -1, which indicates that the file open failed. For example: <pre style="margin-left: 20px;"> ... fid = fopen(filename, 'r'); if fid == -1     % fopen failed  else     % fopen successful, okay to call fread A = fread(fid); ... </pre> </li> <li>• The behavior of the generated code for fread is compiler-dependent if you: <ol style="list-style-type: none"> <li>1 Open a file using fopen with a permission of a+.</li> <li>2 Read the file using fread before calling an I/O function, such as fseek or</li> </ol> </li> </ul>

Name	Product	Remarks and Limitations
		frewind, that sets the file position indicator.
for	MATLAB	—
for	Fixed-Point Designer	—
fpdf	Statistics Toolbox	—

Name	Product	Remarks and Limitations
fprintf	MATLAB	<ul style="list-style-type: none"> <li>• Does not support: <ul style="list-style-type: none"> <li>• <code>b</code> and <code>t</code> subtypes on <code>%u</code>, <code>%o</code> <code>%x</code>, and <code>%X</code> formats.</li> <li>• <code>\$</code> flag for reusing input arguments.</li> <li>• printing arrays.</li> </ul> </li> <li>• There is no automatic casting. Input arguments must match their format types for predictable results.</li> <li>• Escaped characters are limited to the decimal range of 0–127.</li> <li>• A call to <code>fprintf</code> with <code>fileID</code> equal to 1 or 2 becomes <code>printf</code> in the generated C/C++ code in the following cases: <ul style="list-style-type: none"> <li>• The <code>fprintf</code> call is inside a <code>parfor</code> loop.</li> <li>• Extrinsic calls are disabled.</li> </ul> </li> <li>• When the MATLAB behavior differs from the C compiler behavior, <code>fprintf</code> matches the C compiler behavior in the following cases: <ul style="list-style-type: none"> <li>• The format specifier has a corresponding C format specifier, for example, <code>%e</code> or <code>%E</code>.</li> <li>• The <code>printf</code> call is inside a <code>parfor</code> loop.</li> <li>• Extrinsic calls are disabled.</li> </ul> </li> <li>• When you call <code>fprintf</code> with the format specifier <code>%s</code>, do not put a null character in the middle of the input string. Use <code>fprintf(fid, '%c', char(0))</code> to write a null character.</li> <li>• When you call <code>fprintf</code> with an integer format specifier, the type of the integer argument must be a type that the target hardware can represent as a native C type. For example, if you call <code>fprintf('%d',</code></li> </ul>

Name	Product	Remarks and Limitations
		<code>int64(n)</code> , the target hardware must have a native C type that supports a 64-bit integer.

Name	Product	Remarks and Limitations
fread	MATLAB	<ul style="list-style-type: none"> <li>• precision must be a constant.</li> <li>• The source and output that precision specifies cannot have values long, ulong, unsigned long, bitN, or ubitN.</li> <li>• You cannot use the machineformat input.</li> <li>• If the source or output that precision specifies is a C type, for example, int, the target and production sizes for that type must: <ul style="list-style-type: none"> <li>• Match.</li> <li>• Map directly to a MATLAB type.</li> </ul> </li> <li>• The source type that precision specifies must map directly to a C type on the target hardware.</li> <li>• If the fread call reads the entire file, all of the data must fit in the largest array available for code generation.</li> <li>• If sizeA is not constant or contains a nonfinite element, then dynamic memory allocation is required.</li> <li>• Treats a char value for source or output as a signed 8-bit integer. Use values between 0 and 127 only.</li> <li>• The generated code does not report file read errors. Write your own file read error handling in your MATLAB code. Test that the number of bytes read matches the number of bytes that you requested. For example: <pre style="margin-left: 20px;"> ... N = 100; [vals, numRead] = fread(fid, N, '*double'); if numRead ~= N     % fewer elements read than expected </pre> </li> </ul>

Name	Product	Remarks and Limitations
		<p><code>end</code></p> <p>...</p>
<code>freqspace</code>	MATLAB	—
<code>freqz</code>	Signal Processing Toolbox	<p>When called with no output arguments, and without a semicolon at the end, <code>freqz</code> returns the complex frequency response of the input filter, evaluated at 512 points.</p> <p>If the semicolon is added, the function produces a plot of the magnitude and phase response of the filter.</p> <p>See “<code>freqz</code> With No Output Arguments”.</p>
<code>frewind</code>	MATLAB	—
<code>frnd</code>	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
<code>fspecial</code>	Image Processing Toolbox	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
<code>fspl</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>fstat</code>	Statistics Toolbox	—
<code>full</code>	MATLAB	—
<code>fzero</code>	MATLAB	<ul style="list-style-type: none"> <li>• The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument.</li> <li>• Supports up to three output arguments. Does not support the fourth output argument (the <code>output</code> structure).</li> </ul>

Name	Product	Remarks and Limitations
gain2aperture	Phased Array System Toolbox	Does not support variable-size inputs.
gamcdf	Statistics Toolbox	—
gaminv	Statistics Toolbox	—
gamma	MATLAB	—
gammainc	MATLAB	Output is always complex.
gammaincinv	MATLAB	Output is always complex.
gammaln	MATLAB	—
gampdf	Statistics Toolbox	—
gamrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
gamstat	Statistics Toolbox	—
gausswin	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change. <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Not supported for fixed-point signals with different biases.</li> </ul>
geocdf	Statistics Toolbox	—
geoinv	Statistics Toolbox	—

Name	Product	Remarks and Limitations
geomean	Statistics Toolbox	—
geopdf	Statistics Toolbox	—
geornd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
geostat	Statistics Toolbox	—
get	Fixed-Point Designer	• The syntax <code>structure = get(0)</code> is not supported.
getlsb	Fixed-Point Designer	—
getmsb	Fixed-Point Designer	—
getrangefromclass	Image Processing Toolbox	—
gevcdf	Statistics Toolbox	—
gevinv	Statistics Toolbox	—
gevpdf	Statistics Toolbox	—
gevrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
gevstat	Statistics Toolbox	—
global2localcoord	Phased Array System Toolbox	Does not support variable-size inputs.
gpcdf	Statistics Toolbox	—
gpinv	Statistics Toolbox	—
gppdf	Statistics Toolbox	—



Name	Product	Remarks and Limitations
gprnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
gpstat	Statistics Toolbox	—
gradient	MATLAB	—
grazingang	Phased Array System Toolbox	Does not support variable-size inputs.
gt	MATLAB	—
gt	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Not supported for fixed-point signals with different biases.</li> </ul>
hadamard	MATLAB	—
hamming	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.  <h3 style="text-align: center;">Specifying constants</h3> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
hankel	MATLAB	—

Name	Product	Remarks and Limitations
hann	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
harmmean	Statistics Toolbox	—
hdl.RAM	MATLAB	—
hex2dec	MATLAB	—
hex2num	MATLAB	<ul style="list-style-type: none"> <li>• For <math>n = \text{hex2num}(S)</math>, <math>\text{size}(S,2) \leq \text{length}(\text{num2hex}(0))</math></li> </ul>
hilb	MATLAB	—
hist	MATLAB	<ul style="list-style-type: none"> <li>• Histogram bar plotting not supported; call with at least one output argument.</li> <li>• If supplied, the second argument <code>x</code> must be a scalar constant.</li> <li>• Inputs must be real.</li> </ul>
histc	MATLAB	<ul style="list-style-type: none"> <li>• The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.</li> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
histeq	Image Processing Toolbox	<p>All the syntaxes that include indexed images are not supported. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Name	Product	Remarks and Limitations
horizonrange	Phased Array System Toolbox	Does not support variable-size inputs.
horzcat	Fixed-Point Designer	—
hygecdf	Statistics Toolbox	—
hygeinv	Statistics Toolbox	—
hygepdf	Statistics Toolbox	—
hygernd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
hygestat	Statistics Toolbox	—
hypot	MATLAB	—
icdf	Statistics Toolbox	—
idct	Signal Processing Toolbox	<ul style="list-style-type: none"> <li>• Code generation for this function requires the DSP System Toolbox software.</li> <li>• Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.</li> </ul> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
if, elseif, else	MATLAB	—

Name	Product	Remarks and Limitations
<code>idivide</code>	MATLAB	<ul style="list-style-type: none"> <li>For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.</li> </ul>
<code>ifft</code>	MATLAB	<ul style="list-style-type: none"> <li>Length of input vector must be a power of 2.</li> <li>Output of <code>ifft</code> block is complex.</li> <li>Does not support the 'symmetric' option.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
<code>ifft2</code>	MATLAB	<ul style="list-style-type: none"> <li>Length of input matrix dimensions must each be a power of 2.</li> <li>Does not support the 'symmetric' option.</li> </ul>
<code>ifftn</code>	MATLAB	<ul style="list-style-type: none"> <li>Length of input matrix dimensions must each be a power of 2.</li> <li>Does not support the 'symmetric' option.</li> </ul>
<code>ifftshift</code>	MATLAB	—
<code>ifir</code>	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>iircomb</code>	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>iirgrpdelay</code>	DSP System Toolbox	<ul style="list-style-type: none"> <li>All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>Does not support syntaxes that have cell array input.</li> </ul>
<code>iirlpnorm</code>	DSP System Toolbox	<ul style="list-style-type: none"> <li>All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>Does not support syntaxes that have cell array input.</li> </ul>

Name	Product	Remarks and Limitations
iirlpnormc	DSP System Toolbox	<ul style="list-style-type: none"> <li>All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>Does not support syntaxes that have cell array input.</li> </ul>
iirnotch	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
iirpeak	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
im2double	MATLAB	—
im2int16	Image Processing Toolbox	Generated code for this function uses a precompiled platform-specific shared library.
im2single	Image Processing Toolbox	—
im2uint8	Image Processing Toolbox	Generated code for this function uses a precompiled platform-specific shared library.
im2uint16	Image Processing Toolbox	Generated code for this function uses a precompiled platform-specific shared library.
imadjust	Image Processing Toolbox	<p>Does not support syntaxes that include indexed images. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imag	MATLAB	—
imag	Fixed-Point Designer	—
imaq.VideoDevice	Image Acquisition Toolbox	“Code Generation with VideoDevice System Object”

Name	Product	Remarks and Limitations
imbothat	Image Processing Toolbox	<p>The input image <b>IM</b> must be either 2-D or 3-D image. The structuring element input argument <b>SE</b> must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imclearborder	Image Processing Toolbox	<p>The optional second input argument, <b>conn</b>, must be a compile-time constant. Supports only up to 3-D inputs.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imclose	Image Processing Toolbox	<p>The input image <b>IM</b> must be either 2-D or 3-D image. The structuring element input argument <b>SE</b> must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imcomplement	Image Processing Toolbox	<p>Does not support <code>int64</code> and <code>uint64</code> data types.</p>
imdilate	Image Processing Toolbox	<p>The input image <b>IM</b> must be either 2-D or 3-D image. The <b>SE</b>, <b>PACKOPT</b>, and <b>SHAPE</b> input arguments must be a compile-time constant. The structuring element argument <b>SE</b> must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Name	Product	Remarks and Limitations
imerode	Image Processing Toolbox	<p>The input image <b>IM</b> must be either 2-D or 3-D image. The <b>SE</b>, <b>PACKOPT</b>, and <b>SHAPE</b> input arguments must be a compile-time constant. The structuring element argument <b>SE</b> must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imextendedmax	Image Processing Toolbox	<p>The optional third input argument, <b>conn</b>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imextendedmin	Image Processing Toolbox	<p>The optional third input argument, <b>conn</b>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>

Name	Product	Remarks and Limitations
imfill	Image Processing Toolbox	<p>The optional input connectivity, <code>conn</code> and the string <code>'holes'</code> must be compile time constants.</p> <p>Supports only up to 3-D inputs. (No N-D support.)</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code> is not supported in code generation.</p> <p><code>locations</code> can be a <math>P</math>-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a <math>P</math>-by-<code>ndims(I)</code> matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile-time, you cannot change it at run time. However, the number of points in <code>locations</code> can be varied at run time.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imfilter	Image Processing Toolbox	<p>The input image can be either 2-D or 3-D. The value of the input argument, <code>options</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhist	Image Processing Toolbox	<p>The optional second input argument, <code>n</code>, must be a compile-time constant. In addition, nonprogrammatic syntaxes are not supported. For example, the syntaxes where <code>imhist</code> displays the histogram are not supported.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>



Name	Product	Remarks and Limitations
imhmax	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imhmin	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imlincomb	Image Processing Toolbox	<p>The <code>output_class</code> argument must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imopen	Image Processing Toolbox	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imquantize	Image Processing Toolbox	—
imreconstruct	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
imref2d	Image Processing Toolbox	<p>The <code>XWorldLimits</code>, <code>YWorldLimits</code> and <code>ImageSize</code> properties can be set only during object construction. When generating code, you can only specify single objects—arrays of objects are not supported.</p>

Name	Product	Remarks and Limitations
imref3d	Image Processing Toolbox	The <code>XWorldLimits</code> , <code>YWorldLimits</code> , <code>ZWorldLimits</code> and <code>ImageSize</code> properties can be set only during object construction. When generating code, you can only specify single objects—arrays of objects are not supported.
imregionalmax	Image Processing Toolbox	The optional second input argument, <code>conn</code> , must be a compile-time constant.  Generated code for this function uses a precompiled platform-specific shared library.
imregionalmin	Image Processing Toolbox	The optional second input argument, <code>conn</code> , must be a compile-time constant.  Generated code for this function uses a precompiled platform-specific shared library.
imtophat	Image Processing Toolbox	The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.  Generated code for this function uses a precompiled platform-specific shared library.
imwarp	Image Processing Toolbox	Geometric transformation object input, <code>tform</code> , must be either <code>affine2d</code> or <code>projective2d</code> . Additionally, the interpolation method and optional parameter names must be string constants.  Generated code for this function uses a precompiled platform-specific shared library.
ind2sub	MATLAB	<ul style="list-style-type: none"> <li>• The first argument should be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
inf	MATLAB	<ul style="list-style-type: none"> <li>• Dimensions must be real, nonnegative, integers.</li> </ul>

Name	Product	Remarks and Limitations
insertMarker	Computer Vision System Toolbox	Compile-time constant input: marker Supports MATLAB Function block: Yes
insertShape	Computer Vision System Toolbox	Compile-time constant input: shape and SmoothEdges Supports MATLAB Function block: Yes
int8, int16, int32, int64	MATLAB	No integer overflow detection for <code>int64</code> in MEX or MATLAB Function block simulation on Windows 32-bit platforms.
int8, int16, int32, int64	Fixed-Point Designer	—
integralImage	Computer Vision System Toolbox	Supports MATLAB Function block: Yes
interp1	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
interp2	MATLAB	<ul style="list-style-type: none"> <li>• <code>Xq</code> and <code>Yq</code> must be the same size. Use <code>meshgrid</code> to evaluate on a grid.</li> <li>• For best results, provide <code>X</code> and <code>Y</code> as vectors.</li> <li>• For the <code>'cubic'</code> method, reports an error if the grid does not have uniform spacing. In this case, use the <code>'spline'</code> method.</li> <li>• For best results when you use the <code>'spline'</code> method: <ul style="list-style-type: none"> <li>• Use <code>meshgrid</code> to create the inputs <code>Xq</code> and <code>Yq</code>.</li> <li>• Use a small number of interpolation points relative to the dimensions of <code>V</code>. Interpolating over a large set of scattered points can be inefficient.</li> </ul> </li> </ul>

Name	Product	Remarks and Limitations
interp3	MATLAB	<ul style="list-style-type: none"><li>• Xq, Yq, and Zq must be the same size. Use <code>meshgrid</code> to evaluate on a grid.</li><li>• For best results, provide X, Y, and Z as vectors.</li><li>• For the 'cubic' method, reports an error if the grid does not have uniform spacing. In this case, use the 'spline' method.</li><li>• For best results when you use the 'spline' method:<ul style="list-style-type: none"><li>• Use <code>meshgrid</code> to create the inputs Xq, Yq, and Zq.</li><li>• Use a small number of interpolation points relative to the dimensions of V. Interpolating over a large set of scattered points can be inefficient.</li></ul></li></ul>

Name	Product	Remarks and Limitations
intersect	MATLAB	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option: <ul style="list-style-type: none"> <li>• Inputs <i>A</i> and <i>B</i> must be vectors. If you specify the 'legacy' option, inputs <i>A</i> and <i>B</i> must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• The input <code>[]</code> is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' option and the 'rows' option, the outputs <i>ia</i> and <i>ib</i> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <i>C</i> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <i>C</i>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <i>x</i>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>

Name	Product	Remarks and Limitations
intfilt	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.  <b>Specifying constants</b>  To specify a constant input for <code>codegen</code> , use <code>coder.Constant</code> . For more information, see “Specify Constant Inputs at the Command Line”.
intlut	Image Processing Toolbox	Generated code for this function uses a precompiled “platform-specific shared library”.
intmax	MATLAB	—
intmin	MATLAB	—
inv	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	MATLAB	—
ipermute	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
ipermute	Fixed-Point Designer	—
iptcheckconn	Image Processing Toolbox	All input arguments must be compile-time constants.
iptcheckmap	Image Processing Toolbox	—
iqcoef2imbal	Communications System Toolbox	—
iqimbal2coef	Communications System Toolbox	—
iqr	Statistics Toolbox	—
isa	MATLAB	—
iscell	MATLAB	—

Name	Product	Remarks and Limitations
ischar	MATLAB	—
iscolumn	MATLAB	—
iscolumn	Fixed-Point Designer	—
isdeployed	MATLAB Compiler	<ul style="list-style-type: none"> <li>• Returns true and false as appropriate for MEX and SIM targets</li> <li>• Returns false for other targets</li> </ul>
isempty	MATLAB	—
isempty	Fixed-Point Designer	—
isEpipoleInImage	Computer Vision System Toolbox	Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes
isequal	MATLAB	—
isequal	Fixed-Point Designer	—
isequaln	MATLAB	—
isfi	Fixed-Point Designer	—
isfield	MATLAB	<ul style="list-style-type: none"> <li>• Does not support cell input for second argument</li> </ul>
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	MATLAB	—
isfinite	Fixed-Point Designer	—
isfloat	MATLAB	—
ishermitian	MATLAB	—
isinf	MATLAB	—

Name	Product	Remarks and Limitations
isinf	Fixed-Point Designer	—
isinteger	MATLAB	—
isletter	MATLAB	<ul style="list-style-type: none"> <li>Input values from the <code>char</code> class must be in the range 0-127</li> </ul>
islogical	MATLAB	—
ismac	MATLAB	<ul style="list-style-type: none"> <li>Returns true or false based on the MATLAB version used for code generation.</li> <li>Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>
ismatrix	MATLAB	—
ismcc	MATLAB Compiler	<ul style="list-style-type: none"> <li>Returns true and false as appropriate for MEX and SIM targets.</li> <li>Returns false for other targets.</li> </ul>
ismember	MATLAB	<ul style="list-style-type: none"> <li>The second input, <code>B</code>, must be sorted in ascending order.</li> <li>Complex inputs must be <code>single</code> or <code>double</code>.</li> </ul>
isnan	MATLAB	—
isnan	Fixed-Point Designer	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Designer	—
isnumericitype	Fixed-Point Designer	—
isobject	MATLAB	—
ispc	MATLAB	<ul style="list-style-type: none"> <li>Returns true or false based on the MATLAB version you use for code generation.</li> <li>Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>



Name	Product	Remarks and Limitations
isprime	MATLAB	<ul style="list-style-type: none"> <li>• The maximum double precision input is <math>2^{33}</math>.</li> <li>• The maximum single precision input is <math>2^{25}</math>.</li> <li>• The input X cannot have type <code>int64</code> or <code>uint64</code>.</li> </ul>
isreal	MATLAB	—
isreal	Fixed-Point Designer	—
isrow	MATLAB	—
isrow	Fixed-Point Designer	—
isscalar	MATLAB	—
isscalar	Fixed-Point Designer	—
issigned	Fixed-Point Designer	—
issorted	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
isspace	MATLAB	<ul style="list-style-type: none"> <li>• Input values from the <code>char</code> class must be in the range 0–127.</li> </ul>
issparse	MATLAB	—
isstrprop	MATLAB	<ul style="list-style-type: none"> <li>• Supports only inputs from <code>char</code> and <code>integer</code> classes.</li> <li>• Input values must be in the range 0-127.</li> </ul>
isstruct	MATLAB	—
issymmetric	MATLAB	—
istrellis	Communications System Toolbox	—

Name	Product	Remarks and Limitations
isunix	MATLAB	<ul style="list-style-type: none"> <li>• Returns true or false based on the MATLAB version used for code generation.</li> <li>• Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>
isvector	MATLAB	—
isvector	Fixed-Point Designer	—
kaiser	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
kaiserord	Signal Processing Toolbox	—
kron	MATLAB	—
kurtosis	Statistics Toolbox	—

Name	Product	Remarks and Limitations
label2rgb	Image Processing Toolbox	Referring to the standard syntax: <code>RGB = label2rgb(L, map, zerocolor, order)</code> <ul style="list-style-type: none"> <li>• Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>.</li> <li>• <code>map</code> must be an <code>n</code>-by-3, double, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function.</li> <li>• If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning.</li> <li>• If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.</li> </ul>
lcm	MATLAB	—
lcmvweights	Phased Array System Toolbox	Does not support variable-size inputs.
ldivide	MATLAB	—
le	MATLAB	—
le	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Not supported for fixed-point signals with different biases.</li> </ul>
length	MATLAB	—
length	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
levinson	Signal Processing Toolbox	<ul style="list-style-type: none"> <li>• Code generation for this function requires the DSP System Toolbox software.</li> <li>• If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.</li> </ul> <p style="text-align: center;"><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
lineToBorderPoints	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions.            Supports MATLAB Function block: Yes</p>
linsolve	MATLAB	<ul style="list-style-type: none"> <li>• The option structure must be a constant.</li> <li>• Supports only a scalar option structure input. It does not support arrays of option structures.</li> <li>• Only optimizes these cases:               <ul style="list-style-type: none"> <li>• UT</li> <li>• LT</li> <li>• UHESS = true (the TRANSA can be either true or false)</li> <li>• SYM = true and POSDEF = true</li> </ul> </li> </ul> <p>Other options are equivalent to using <code>mldivide</code>.</p>
linspace	MATLAB	—

Name	Product	Remarks and Limitations
load	MATLAB	<ul style="list-style-type: none"> <li>• Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use <code>coder.load</code>.</li> <li>• Does not support use of the function without assignment to a structure or array. For example, use <code>S = load(filename)</code>, not <code>load(filename)</code>.</li> <li>• The output <code>S</code> must be the name of a structure or array without any subscripting. For example, <code>S[i] = load('myFile.mat')</code> is not allowed.</li> <li>• Arguments to <code>load</code> must be compile-time constant strings.</li> <li>• Does not support loading objects.</li> <li>• If the MAT-file contains unsupported constructs, use <code>load(filename, variables)</code> to load only the supported constructs.</li> <li>• You cannot use <code>save</code> in a function intended for code generation. The code generation software does not support the <code>save</code> function. Furthermore, you cannot use <code>coder.extrinsic</code> with <code>save</code>. Prior to generating code, you can use <code>save</code> to save the workspace data to a MAT-file.</li> </ul> <p>You must use <code>coder. varsized</code> to explicitly declare variable-size data loaded using the <code>load</code> function.</p>
local2globalcoord	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
log	MATLAB	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Designer	—
logncdf	Statistics Toolbox	—
logninv	Statistics Toolbox	—
lognpdf	Statistics Toolbox	—
lognrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>The output is nonscalar.</li> <li>An input parameter is invalid for the distribution.</li> </ul>
lognstat	Statistics Toolbox	—
logspace	MATLAB	—
lower	MATLAB	<ul style="list-style-type: none"> <li>Supports only <code>char</code> inputs.</li> <li>Input values must be in the range 0-127.</li> </ul>
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Supported for scalar fixed-point signals only.</li> <li>Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.</li> </ul>
lt	MATLAB	—

Name	Product	Remarks and Limitations
lteZadoffChuSeq	Communications System Toolbox	—
lt	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Not supported for fixed-point signals with different biases.</li> </ul>
lu	MATLAB	—
mad	Statistics Toolbox	Input <code>dim</code> cannot be empty.
magic	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
matchFeatures	Computer Vision System Toolbox	<p>Generates platform-dependent library: Yes for MATLAB host. The function generates portable C code for non-host target.</p> <p>Compile-time constant input: Method and Metric.</p> <p>Supports MATLAB Function block: Yes</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
max	MATLAB	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
max	Fixed-Point Designer	—
maxflat	Signal Processing Toolbox	<p>Inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
mdltest	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
mean	MATLAB	<ul style="list-style-type: none"> <li>Does not support the 'native' output class option for integer types.</li> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
mean	Fixed-Point Designer	N/A
mean2	Image Processing Toolbox	—
medfilt2	Image Processing Toolbox	<p>The <code>padopt</code> argument must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
median	MATLAB	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
median	Fixed-Point Designer	—
meshgrid	MATLAB	—
mfilename	MATLAB	—
min	MATLAB	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
min	Fixed-Point Designer	—
minus	MATLAB	—
minus	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Any non-<code>fi</code> input must be constant. Its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> </ul>



Name	Product	Remarks and Limitations
mkpp	MATLAB	<ul style="list-style-type: none"> <li>• The output structure <code>pp</code> differs from the <code>pp</code> structure in MATLAB. In MATLAB, <code>ppval</code> cannot use the <code>pp</code> structure from the code generation software. For code generation, <code>ppval</code> cannot use a <code>pp</code> structure created by MATLAB. <code>unmkpp</code> can use a MATLAB <code>pp</code> structure for code generation.</li> </ul> <p>To create a MATLAB <code>pp</code> structure from a <code>pp</code> structure created by the code generation software:</p> <ul style="list-style-type: none"> <li>• In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB.</li> <li>• In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.</li> </ul> <ul style="list-style-type: none"> <li>• If you do not provide <code>d</code>, then <code>coefs</code> must be two-dimensional and have a fixed number of columns. In this case, the number of columns is the order.</li> <li>• To define a piecewise constant polynomial, <code>coefs</code> must be a column vector or <code>d</code> must have at least two elements.</li> <li>• If you provide <code>d</code> and <code>d</code> is 1, <code>d</code> must be a constant. Otherwise, if the input to <code>ppval</code> is nonscalar, the shape of the output of <code>ppval</code> can differ from <code>ppval</code> in MATLAB.</li> <li>• If you provide <code>d</code>, it must have a fixed length. One of the following sets of statements must be true: <ul style="list-style-type: none"> <li>1 Suppose that <code>m = length(d)</code> and <code>npieces = length(breaks) - 1</code>.</li> </ul> <pre style="margin-left: 20px;"> size(coefs,j) = d(j) size(coefs,m+1) = npieces </pre> </li> </ul>

Name	Product	Remarks and Limitations
		<p> <code>size(coefs,m+2) = order</code>  <code>j = 1,2,...,m</code>. The dimension <code>m+2</code> must be fixed length.                 </p> <p> <b>2</b> Suppose that <code>m = length(d)</code> and <code>npieces = length(breaks) - 1</code>.                 </p> <p> <code>size(coefs,1) = prod(d)*npieces</code>  <code>size(coefs,2) = order</code>                      The second dimension must be fixed length.                 </p> <ul style="list-style-type: none"> <li>If you do not provide <code>d</code>, the following statements must be true:                         <p>                             Suppose that <code>m = length(d)</code> and <code>npieces = length(breaks) - 1</code>.                               <code>size(coefs,1) = prod(d)*npieces</code>  <code>size(coefs,2) = order</code>                              The second dimension must be fixed length.                         </p> </li> </ul>
mldivide	MATLAB	—
mnpdf	Statistics Toolbox	—
mod	MATLAB	<ul style="list-style-type: none"> <li>Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors.</li> </ul> <p>If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.</p>
mode	MATLAB	<ul style="list-style-type: none"> <li>Does not support third output argument <code>C</code> (cell array).</li> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
moment	Statistics Toolbox	If <code>order</code> is nonintegral and <code>X</code> is real, use <code>moment(complex(X), order)</code> .
mpower	MATLAB	—

Name	Product	Remarks and Limitations
mpower	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• The exponent input, <math>k</math>, must be constant; that is, its value must be known at compile time.</li> <li>• Variable-sized inputs are only supported when the <b>SumMode</b> property of the governing <b>fimath</b> is set to <b>Specify precision</b> or <b>Keep LSB</b>.</li> <li>• For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> <li>• In generated code, the output for variable-sized signals is computed using the <b>SumMode</b> property of the governing <b>fimath</b>.</li> <li>• In MATLAB, the output for variable-sized signals is computed using the <b>SumMode</b> property of the governing <b>fimath</b> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <b>ProductMode</b> of the governing <b>fimath</b>.</li> </ul> </li> </ul>
mpy	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Code generation in MATLAB does not support the syntax <code>F.mpy(a,b)</code>. You must use the syntax <code>mpy(F,a,b)</code>.</li> <li>• When you provide complex inputs to the <b>mpy</b> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the <b>Ports and data manager</b> and set the <b>Complexity</b> parameter for all known complex inputs to <b>On</b>.</li> </ul>
mrdivide	MATLAB	—
mrdivide	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
MSERRegions	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes</p> <p>For code generation, you must specify both the pixellist cell array and the length of each array, as the second input. The object outputs, regions.PixelList as an array. The region sizes are defined in regions.Lengths.</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
mtimes	MATLAB	<ul style="list-style-type: none"> <li>• Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generation software does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, <math>(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Inf}i</math>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

Name	Product	Remarks and Limitations
mtimes	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> <li>• Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.</li> <li>• For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> <li>• In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>.</li> <li>• In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.</li> </ul> </li> </ul>
multithresh	Image Processing Toolbox	—
mvdweights	Phased Array System Toolbox	Does not support variable-size inputs.
NaN or nan	MATLAB	<ul style="list-style-type: none"> <li>• Dimensions must be real, nonnegative, integers.</li> </ul>
nancov	Statistics Toolbox	If the input is variable-size and is <code>[]</code> at run time, returns <code>[]</code> not NaN.
nanmax	Statistics Toolbox	—
nanmean	Statistics Toolbox	—
nanmedian	Statistics Toolbox	—
nanmin	Statistics Toolbox	—

Name	Product	Remarks and Limitations
nanstd	Statistics Toolbox	—
nansum	Statistics Toolbox	—
nanvar	Statistics Toolbox	—
nargchk	MATLAB	<ul style="list-style-type: none"> <li>Output structure does not include stack information.</li> </ul> <hr/> <p><b>Note:</b> nargchk will be removed in a future release.</p>
nargin	MATLAB	—
narginchk	MATLAB	—
nargout	MATLAB	<ul style="list-style-type: none"> <li>For a function with no output arguments, returns 1 if called without a terminating semicolon.</li> </ul> <hr/> <p><b>Note:</b> This behavior also affects extrinsic calls with no terminating semicolon. nargout is 1 for the called function in MATLAB.</p>
nargoutchk	MATLAB	—
nbincdf	Statistics Toolbox	—
nbinv	Statistics Toolbox	—
nbipdf	Statistics Toolbox	—
nbirnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>The output is nonscalar.</li> <li>An input parameter is invalid for the distribution.</li> </ul>
nbinstat	Statistics Toolbox	—
ncfcdf	Statistics Toolbox	—
ncfinv	Statistics Toolbox	—

Name	Product	Remarks and Limitations
ncfpdf	Statistics Toolbox	—
ncfrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
ncfstat	Statistics Toolbox	—
nchoosek	MATLAB	<ul style="list-style-type: none"> <li>• When the first input, <math>x</math>, is a scalar, <code>nchoosek</code> returns a binomial coefficient. In this case, <math>x</math> must be a nonnegative integer. It cannot have type <code>int64</code> or <code>uint64</code>.</li> <li>• When the first input, <math>x</math>, is a vector, <code>nchoosek</code> treats it as a set. In this case, <math>x</math> can have type <code>int64</code> or <code>uint64</code>.</li> <li>• The second input, <math>k</math>, cannot have type <code>int64</code> or <code>uint64</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
nctcdf	Statistics Toolbox	—
nctinv	Statistics Toolbox	—
nctpdf	Statistics Toolbox	—
nctrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
nctstat	Statistics Toolbox	—
ncx2cdf	Statistics Toolbox	—

Name	Product	Remarks and Limitations
ncx2rnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
ncx2stat	Statistics Toolbox	—
ndgrid	MATLAB	—
ndims	MATLAB	—
ndims	Fixed-Point Designer	—
ne	MATLAB	—
ne	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Not supported for fixed-point signals with different biases.</li> </ul>
nearest	Fixed-Point Designer	—
nextpow2	MATLAB	—
nnz	MATLAB	—
noisepow	Phased Array System Toolbox	Does not support variable-size inputs.
nonzeros	MATLAB	—
norm	MATLAB	—
normcdf	Statistics Toolbox	—
normest	MATLAB	—
norminv	Statistics Toolbox	—
normpdf	Statistics Toolbox	—



Name	Product	Remarks and Limitations
<code>normrnd</code>	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
<code>normstat</code>	Statistics Toolbox	—
<code>not</code>	MATLAB	—
<code>npwgnthresh</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>nthroot</code>	MATLAB	—
<code>null</code>	MATLAB	<ul style="list-style-type: none"> <li>• Might return a different basis than MATLAB</li> <li>• Does not support rational basis option (second input)</li> </ul>
<code>num2hex</code>	MATLAB	—
<code>numberofelements</code>	Fixed-Point Designer	<code>numberofelements</code> will be removed in a future release. Use <code>numel</code> instead.
<code>numel</code>	MATLAB	—
<code>numel</code>	Fixed-Point Designer	—
<code>numerictype</code>	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Fixed-point signals coming into a MATLAB Function block from Simulink are assigned a <code>numerictype</code> object that is populated with the signal's data type and scaling information.</li> <li>• Returns the data type when the input is a nonfixed-point signal.</li> <li>• Use to create <code>numerictype</code> objects in the generated code.</li> <li>• All <code>numerictype</code> object properties related to the data type must be constant.</li> </ul>

Name	Product	Remarks and Limitations
nuttallwin	Signal Processing Toolbox	<p>Inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
ocr	Computer Vision System Toolbox	<p>Compile-time constant input: TextLayout, Language, and CharacterSet.</p> <p>Supports MATLAB Function block: No</p> <p>Generated code for this function uses a precompiled platform-specific shared library.</p>
ocrText	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions.</p> <p>Supports MATLAB Function block: No</p>
ode23	MATLAB	<ul style="list-style-type: none"> <li>• All <code>odeset</code> option arguments must be constant.</li> <li>• Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function .</li> <li>• You must provide at least the two output arguments T and Y.</li> <li>• Input types must be homogeneous—all double or all single.</li> <li>• Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.</li> </ul>

Name	Product	Remarks and Limitations
ode45	MATLAB	<ul style="list-style-type: none"> <li>• All <code>odeset</code> option arguments must be constant.</li> <li>• Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function .</li> <li>• You must provide at least the two output arguments <code>T</code> and <code>Y</code>.</li> <li>• Input types must be homogeneous—all double or all single.</li> <li>• Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.</li> </ul>
odeget	MATLAB	The <code>name</code> argument must be constant.
odeset	MATLAB	All inputs must be constant.
ones	MATLAB	<ul style="list-style-type: none"> <li>• Dimensions must be real, nonnegative integers.</li> <li>• The input <code>optimfun</code> must be a function supported for code generation.</li> </ul>
optimget	MATLAB	Input parameter names must be constant.

Name	Product	Remarks and Limitations
optimset	MATLAB	<ul style="list-style-type: none"> <li>• Does not support the syntax that has no input or output arguments: <code>optimset</code></li> <li>• Functions specified in the options must be supported for code generation.</li> <li>• The fields of the options structure <code>oldopts</code> must be fixed-size fields.</li> <li>• For code generation, optimization functions ignore the <code>Display</code> option.</li> <li>• Does not support the additional options in an options structure created by the Optimization Toolbox™ <code>optimset</code> function. If an input options structure includes the additional Optimization Toolbox options, the output structure does not include them.</li> </ul>
ordfilt2	Image Processing Toolbox	<p>The <code>padopt</code> argument must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
or	MATLAB	—
orth	MATLAB	<ul style="list-style-type: none"> <li>• Can return a different basis than MATLAB</li> </ul>
padarray	Image Processing Toolbox	<ul style="list-style-type: none"> <li>• Support only up to 3-D inputs.</li> <li>• Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.</li> </ul>
parfor	MATLAB	<ul style="list-style-type: none"> <li>• Treated as a <code>for</code>-loop in a MATLAB Function block or when used with <code>fiaccl</code>.</li> <li>• See the <code>parfor</code> reference page in the MATLAB Coder documentation.</li> <li>• “Generate Code with Parallel for-Loops (<code>parfor</code>)”.</li> </ul>

Name	Product	Remarks and Limitations
parzenwin	Signal Processing Toolbox	<p>Inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
pascal	MATLAB	—
pchip	MATLAB	<ul style="list-style-type: none"> <li>• Input <code>x</code> must be strictly increasing.</li> <li>• Does not remove <code>y</code> entries with NaN values.</li> <li>• If you generate code for the <code>pp = pchip(x,y)</code> syntax, you cannot input <code>pp</code> to the <code>ppval</code> function in MATLAB. To create a MATLAB <code>pp</code> structure from a <code>pp</code> structure created by the code generation software: <ul style="list-style-type: none"> <li>• In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB.</li> <li>• In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.</li> </ul> </li> </ul>
pdf	Statistics Toolbox	—
permute	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
permute	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
phased.ADPCACanceller	Phased Array System Toolbox	“Code Generation”
phased.AngleDoppler-Response	Phased Array System Toolbox	“Code Generation”

Name	Product	Remarks and Limitations
phased.ArrayGain	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>Does not support arrays containing polarized antenna elements, that is, the <code>phased.ShortDipoleAntennaElement</code> or <code>phased.CrossedDipoleAntennaElement</code> antennas.</li> <li>“Code Generation”.</li> </ul>
phased.ArrayResponse	Phased Array System Toolbox	“Code Generation”
phased.BarrageJammer	Phased Array System Toolbox	“Code Generation”
phased.Beamscan-Estimator	Phased Array System Toolbox	“Code Generation”
phased.Beamscan-Estimator2D	Phased Array System Toolbox	“Code Generation”
phased.Beamspace-ESPRITEstimator	Phased Array System Toolbox	“Code Generation”.
phased.CFARDetector	Phased Array System Toolbox	“Code Generation”
phased.Collector	Phased Array System Toolbox	“Code Generation”
phased.ConformalArray	Phased Array System Toolbox	<ul style="list-style-type: none"> <li><code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>“Code Generation”.</li> </ul>
phased.Constant-GammaClutter	Phased Array System Toolbox	“Code Generation”
phased.Cosine-AntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li><code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>“Code Generation”.</li> </ul>
phased.Crossed-DipoleAntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li><code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>“Code Generation”.</li> </ul>

Name	Product	Remarks and Limitations
phased.Custom-AntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.Custom-MicrophoneElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”</li> </ul>
phased.DPCACanceller	Phased Array System Toolbox	“Code Generation”
phased.ElementDelay	Phased Array System Toolbox	“Code Generation”
phased.ESPRITEstimator	Phased Array System Toolbox	“Code Generation”
phased.FMCWWaveform	Phased Array System Toolbox	“Code Generation”
phased.FreeSpace	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
phased.FrostBeamformer	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
phased.Isotropic-AntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.LCMVBeamformer	Phased Array System Toolbox	“Code Generation”
phased.LinearFMWaveform	Phased Array System Toolbox	“Code Generation”
phased.MatchedFilter	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• The <code>CustomSpectrumWindow</code> property is not supported.</li> <li>• “Code Generation”.</li> </ul>

Name	Product	Remarks and Limitations
phased.MVDRBeamformer	Phased Array System Toolbox	“Code Generation”
phased.MVDREstimator	Phased Array System Toolbox	“Code Generation”
phased.MVDREstimator2D	Phased Array System Toolbox	“Code Generation”
phased.Omnidirectional-MicrophoneElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.PartitionedArray	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.PhaseCoded-Waveform	Phased Array System Toolbox	“Code Generation”
phased.PhaseShift-Beamformer	Phased Array System Toolbox	“Code Generation”
phased.Platform	Phased Array System Toolbox	“Code Generation”
phased.RadarTarget	Phased Array System Toolbox	“Code Generation”
phased.Radiator	Phased Array System Toolbox	“Code Generation”
phased.Range-DopplerResponse	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• The <code>CustomRangeWindow</code> and the <code>CustomDopplerWindow</code> properties are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.Rectangular-Waveform	Phased Array System Toolbox	“Code Generation”
phased.ReceiverPreamp	Phased Array System Toolbox	“Code Generation”



Name	Product	Remarks and Limitations
phased.Replicated-Subarray	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.RootMUSIC-Estimator	Phased Array System Toolbox	“Code Generation”
phased.RootWSFEstimator	Phased Array System Toolbox	“Code Generation”
phased.ShortDipole-AntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.STAPSMI-Beamformer	Phased Array System Toolbox	“Code Generation”
phased.StretchProcessor	Phased Array System Toolbox	“Code Generation”
phased.SubbandPhase-ShiftBeamformer	Phased Array System Toolbox	“Code Generation”
phased.SteeringVector	Phased Array System Toolbox	“Code Generation”
phased.Stepped-FMWaveform	Phased Array System Toolbox	“Code Generation”
phased.SumDifference-MonopulseTracker	Phased Array System Toolbox	“Code Generation”.
phased.SumDifference-MonopulseTracker2D	Phased Array System Toolbox	“Code Generation”.
phased.TimeDelay-Beamformer	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
phased.TimeDelayLCMV-Beamformer	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>

Name	Product	Remarks and Limitations
<code>phased.TimeVaryingGain</code>	Phased Array System Toolbox	“Code Generation”.
<code>phased.Transmitter</code>	Phased Array System Toolbox	“Code Generation”.
<code>phased.ULA</code>	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
<code>phased.URA</code>	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
<code>phased.Wideband-Collector</code>	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
<code>phitheta2azel</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>phitheta2azelpat</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>phitheta2uv</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>phitheta2uvpat</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>physconst</code>	Phased Array System Toolbox	Does not support variable-size inputs.
<code>pi</code>	MATLAB	—
<code>pinv</code>	MATLAB	—
<code>planerot</code>	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
<code>plus</code>	MATLAB	—
<code>plus</code>	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> </ul>

Name	Product	Remarks and Limitations
poisscdf	Statistics Toolbox	—
poissinv	Statistics Toolbox	—
poisspdf	Statistics Toolbox	—
poissrnd	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
poisstat	Statistics Toolbox	—
pol2cart	MATLAB	—
pol2circpol	Phased Array System Toolbox	Does not support variable-size inputs.
polellip	Phased Array System Toolbox	Does not support variable-size inputs.
polloss	Phased Array System Toolbox	Does not support variable-size inputs.
polratio	Phased Array System Toolbox	Does not support variable-size inputs.
polsignature	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• Supported only when output arguments are specified.</li> </ul>
poly	MATLAB	<ul style="list-style-type: none"> <li>• Does not discard nonfinite input values</li> <li>• Complex input produces complex output</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
polyarea	MATLAB	—
poly2trellis	Communications System Toolbox	—
polyder	MATLAB	The output can contain fewer NaNs than the MATLAB output. However, if the input contains a NaN, the output contains at least one NaN.

Name	Product	Remarks and Limitations
polyfit	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
polyint	MATLAB	—
polyval	MATLAB	—
polyvalm	MATLAB	—
pow2	Fixed-Point Designer	—
pow2db	Signal Processing Toolbox	—
power	MATLAB	<ul style="list-style-type: none"> <li>Generates an error during simulation. When both <math>X</math> and <math>Y</math> are real, but <math>\text{power}(X, Y)</math> is complex, returns NaN in the generated code. To get the complex result, make the input value <math>X</math> complex by passing in <math>\text{complex}(X)</math>. For example, <math>\text{power}(\text{complex}(X), Y)</math>.</li> <li>Generates an error during simulation. When both <math>X</math> and <math>Y</math> are real, but <math>X.^Y</math> is complex, returns NaN in generated code. To get the complex result, make the input value <math>X</math> complex by using <math>\text{complex}(X)</math>. For example, <math>\text{complex}(X).^Y</math>.</li> </ul>
power	Fixed-Point Designer	<ul style="list-style-type: none"> <li>The exponent input, <math>k</math>, must be constant. Its value must be known at compile time.</li> </ul>

Name	Product	Remarks and Limitations
ppval	MATLAB	<p>The size of output <math>v</math> does not match MATLAB when both of the following statements are true:</p> <ul style="list-style-type: none"> <li>• The input <math>x</math> is a variable-size array that is not a variable-length vector.</li> <li>• <math>x</math> becomes a row vector at run time.</li> </ul> <p>The code generation software does not remove the singleton dimensions. However, MATLAB might remove singleton dimensions.</p> <p>For example, suppose that <math>x</math> is a :4-by-:5 array (the first dimension is variable size with an upper bound of 4 and the second dimension is variable size with an upper bound of 5). Suppose that <code>ppval(pp,0)</code> returns a 2-by-3 fixed-size array. <math>v</math> has size 2-by-3-by-:4-by-:5. At run time, suppose that, <code>size(x,1) = 1</code> and <code>size(x,2) = 5</code>. In the generated code, the <code>size(v)</code> is [2,3,1,5]. In MATLAB, the size is [2,3,5].</p>
prctile	Statistics Toolbox	<ul style="list-style-type: none"> <li>• “Automatic dimension restriction”</li> <li>• If the output <math>Y</math> is a vector, the orientation of <math>Y</math> differs from MATLAB when all of the following are true: <ul style="list-style-type: none"> <li>• You do not supply the <code>dim</code> input.</li> <li>• <math>X</math> is a variable-size array.</li> <li>• <math>X</math> is not a variable-length vector.</li> <li>• <math>X</math> is a vector at run time.</li> <li>• The orientation of the vector <math>X</math> does not match the orientation of the vector <math>p</math>.</li> </ul> </li> </ul> <p>In this case, the output <math>Y</math> matches the orientation of <math>X</math> not the orientation of <math>p</math>.</p>

Name	Product	Remarks and Limitations
primes	MATLAB	<ul style="list-style-type: none"> <li>The maximum double precision input is <math>2^{32}</math>.</li> <li>The maximum single precision input is <math>2^{24}</math>.</li> <li>The input <code>n</code> cannot have type <code>int64</code> or <code>uint64</code>.</li> </ul>
prod	MATLAB	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
projective2d	Image Processing Toolbox	When generating code, you can only specify single objects—arrays of objects are not supported.
psi	MATLAB	—
pulsint	Phased Array System Toolbox	Does not support variable-size inputs.
qr	MATLAB	—
quad2d	MATLAB	<ul style="list-style-type: none"> <li>Generates a warning if the size of the internal storage arrays is not large enough. If a warning occurs, a possible workaround is to divide the region of integration into pieces and sum the integrals over each piece.</li> </ul>
quadgk	MATLAB	—
quantile	Statistics Toolbox	—
quantize	Fixed-Point Designer	—
quatconj	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset™ software.
quatdivide	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset software.
quatinv	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset software.
quatmod	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset software.

Name	Product	Remarks and Limitations
quatmultiply	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset software.
quatnorm	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset software.
quatnormalize	Aerospace Toolbox	Code generation for this function requires the Aerospace Blockset software.
radareqpow	Phased Array System Toolbox	Does not support variable-size inputs.
radareqrng	Phased Array System Toolbox	Does not support variable-size inputs.
radareqsnr	Phased Array System Toolbox	Does not support variable-size inputs.
radarvcd	Phased Array System Toolbox	Does not support variable-size inputs.
radialspeed	Phased Array System Toolbox	Does not support variable-size inputs.
rand	MATLAB	<ul style="list-style-type: none"> <li>• <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>rand</code> method for other classes. For example, <code>rand(sz, 'myclass')</code> does not invoke <code>myclass.rand(sz)</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
randg	Statistics Toolbox	—
randi	MATLAB	<ul style="list-style-type: none"> <li>• <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>randi</code> method for other classes. For example, <code>randi(imax, sz, 'myclass')</code> does not invoke <code>myclass.randi(imax, sz)</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

Name	Product	Remarks and Limitations
randn	MATLAB	<ul style="list-style-type: none"> <li>• <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>randn</code> method for other classes. For example, <code>randn(sz, 'myclass')</code> does not invoke <code>myclass.randn(sz)</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
random	Statistics Toolbox	—
randperm	MATLAB	—
range	Fixed-Point Designer	—
range2beat	Phased Array System Toolbox	Does not support variable-size inputs.
range2bw	Phased Array System Toolbox	Does not support variable-size inputs.
range2time	Phased Array System Toolbox	Does not support variable-size inputs.
rangeangle	Phased Array System Toolbox	Does not support variable-size inputs.
rank	MATLAB	—
raylcdf	Statistics Toolbox	—
raylinv	Statistics Toolbox	—
raylpdf	Statistics Toolbox	—
raylrnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
raylstat	Statistics Toolbox	—
rcond	MATLAB	—



Name	Product	Remarks and Limitations
rcosdesign	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
rdcoupling	Phased Array System Toolbox	Does not support variable-size inputs.
rdivide	MATLAB	—
rdivide	Fixed-Point Designer	—
real	MATLAB	—
real	Fixed-Point Designer	—
reallog	MATLAB	—
realmax	MATLAB	—
realmax	Fixed-Point Designer	—
realmin	MATLAB	—
realmin	Fixed-Point Designer	—
realpow	MATLAB	—
realsqrt	MATLAB	—
rectint	MATLAB	—

Name	Product	Remarks and Limitations
rectwin	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
reinterpretpcast	Fixed-Point Designer	—
rem	MATLAB	<ul style="list-style-type: none"> <li>• Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors.</li> <li>• If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.</li> </ul>
removefimath	Fixed-Point Designer	—
repmat	MATLAB	—
repmat	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
resample	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>

Name	Product	Remarks and Limitations
rescale	Fixed-Point Designer	—
reshape	MATLAB	<ul style="list-style-type: none"> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
reshape	Fixed-Point Designer	—
return	MATLAB	—
rgb2ycbcr	Image Processing Toolbox	—
rng	MATLAB	<ul style="list-style-type: none"> <li>• For library code generation targets, executable code generation targets, and MEX targets with extrinsic calls disabled: <ul style="list-style-type: none"> <li>• Does not support the 'shuffle' input.</li> <li>• For the generator input, supports 'twister', 'v4', and 'v5normal'.</li> </ul> </li> </ul> <p>For these targets, the output of <code>s=rng</code> in the generated code differs from the MATLAB output. You cannot return the output of <code>s=rng</code> from the generated code and pass it to <code>rng</code> in MATLAB.</p> <ul style="list-style-type: none"> <li>• For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by <code>rng</code>.</li> </ul>
rocpfa	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• The <code>NonfluctuatingNoncoherent</code> signal type is not supported.</li> </ul>
rocsnr	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• Does not support the <code>NonfluctuatingNoncoherent</code> signal type.</li> </ul>
rootmusicdoa	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
roots	MATLAB	<ul style="list-style-type: none"> <li>• Output is variable size.</li> <li>• Output is complex.</li> <li>• Roots are not always in the same order as MATLAB.</li> <li>• Roots of poorly conditioned polynomials do not always match MATLAB.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
rosser	MATLAB	—
rot90	MATLAB	—
rot90	Fixed-Point Designer	In the syntax <code>rot90(A,k)</code> , the argument <code>k</code> must be a built-in type; it cannot be a <code>fi</code> object.
rotx	Phased Array System Toolbox	Does not support variable-size inputs.
roty	Phased Array System Toolbox	Does not support variable-size inputs.
rotz	Phased Array System Toolbox	Does not support variable-size inputs.
round	MATLAB	—
round	Fixed-Point Designer	—
rsf2csf	MATLAB	—
schur	MATLAB	Can return a different Schur decomposition in generated code than in MATLAB.
sec	MATLAB	—
secd	MATLAB	<ul style="list-style-type: none"> <li>• In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>• In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
sech	MATLAB	—

---

Name	Product	Remarks and Limitations
selectStrongestBbox	Computer Vision System Toolbox	Compile-time constant input: No restriction Supports MATLAB Function block: No
sensorcov	Phased Array System Toolbox	Does not support variable-size inputs.
sensorsig	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
setdiff	MATLAB	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option:               <ul style="list-style-type: none"> <li>• Inputs <i>A</i> and <i>B</i> must be vectors. If you specify the 'legacy' option, inputs <i>A</i> and <i>B</i> must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• Do not use [ ] to represent the empty set. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' and 'rows' options, the output <i>ia</i> is a column vector. If <i>ia</i> is empty, it is 0-by-1, never 0-by-0, even if the output <i>C</i> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <i>C</i>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following:               <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <i>x</i>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>

---

Name	Product	Remarks and Limitations
setfimath	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
setxor	MATLAB	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option:               <ul style="list-style-type: none"> <li>• Inputs A and B must be vectors with the same orientation. If you specify the 'legacy' option, inputs A and B must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• The input [ ] is not supported. Use a 1-by-0 or 0-by-1 input, for example , <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' option and the 'rows' option, the outputs <code>ia</code> and <code>ib</code> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <code>C</code> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' flag, the inputs must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following:               <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <code>x</code>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>



Name	Product	Remarks and Limitations
sfi	Fixed-Point Designer	<ul style="list-style-type: none"> <li>All properties related to data type must be constant for code generation.</li> </ul>
sgolay	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
shiftdim	MATLAB	<ul style="list-style-type: none"> <li>Second argument must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
shiftdim	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
shnidman	Phased Array System Toolbox	Does not support variable-size inputs.
sign	MATLAB	—
sign	Fixed-Point Designer	—
sin	MATLAB	—
sin	Fixed-Point Designer	—
sind	MATLAB	—
single	MATLAB	—
single	Fixed-Point Designer	—
sinh	MATLAB	—
size	MATLAB	—

Name	Product	Remarks and Limitations
size	Fixed-Point Designer	—
skewness	Statistics Toolbox	—
sort	MATLAB	If the input is a complex type, <code>sort</code> orders the output according to absolute value. When <code>x</code> is a complex type that has all zero imaginary parts, use <code>sort(real(x))</code> to compute the sort order for real types. See “Code Generation for Complex Data”.
sort	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
sortrows	MATLAB	If the input is a complex type, <code>sortrows</code> orders the output according to absolute value. When <code>x</code> is a complex type that has all zero imaginary parts, use <code>sortrows(real(x))</code> to compute the sort order for real types. See “Code Generation for Complex Data”.
sosfilt	Signal Processing Toolbox	—
speed2dop	Phased Array System Toolbox	Does not support variable-size inputs.
sph2cart	MATLAB	—
sph2cartvec	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
spline	MATLAB	<ul style="list-style-type: none"> <li>• Input <math>x</math> must be strictly increasing.</li> <li>• Does not remove <math>Y</math> entries with NaN values.</li> <li>• Does not report an error for infinite endslopes in <math>Y</math>.</li> <li>• If you generate code for the <code>pp = spline(x,Y)</code> syntax, you cannot input <code>pp</code> to the <code>ppval</code> function in MATLAB. To create a MATLAB <code>pp</code> structure from a <code>pp</code> structure created by the code generation software: <ul style="list-style-type: none"> <li>• In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB.</li> <li>• In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.</li> </ul> </li> </ul>
spsmooth	Phased Array System Toolbox	Does not support variable-size inputs.
squeeze	MATLAB	—
squeeze	Fixed-Point Designer	—
sqrt	MATLAB	<ul style="list-style-type: none"> <li>• Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
sqrt	Fixed-Point Designer	<ul style="list-style-type: none"> <li>• Complex and [Slope Bias] inputs error out.</li> <li>• Negative inputs yield a 0 result.</li> </ul>
sqrtm	MATLAB	—
std	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
steervec	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
stokes	Phased Array System Toolbox	<ul style="list-style-type: none"> <li>Does not support variable-size inputs.</li> <li>Supported only when output arguments are specified.</li> </ul>
storedInteger	Fixed-Point Designer	—
storedIntegerToDouble	Fixed-Point Designer	—
str2double	MATLAB	<ul style="list-style-type: none"> <li>Does not support cell arrays.</li> <li>Always returns a complex result.</li> </ul>
str2func	MATLAB	String must be constant/known at compile time.
strcmp	MATLAB	—
strcmpi	MATLAB	<ul style="list-style-type: none"> <li>Input values from the char class must be in the range 0-127.</li> </ul>
strel	Image Processing Toolbox	Input arguments must be compile-time constants. The following methods are not supported for code generation: <code>getsequence</code> , <code>reflect</code> , <code>translate</code> , <code>disp</code> , <code>display</code> , <code>loadobj</code> . When generating code, you can only specify single objects—arrays of objects are not supported.
stretchfreq2rng	Phased Array System Toolbox	Does not support variable-size inputs.
stretchlim	Image Processing Toolbox	Generated code for this function uses a precompiled “platform-specific shared library”.
strfind	MATLAB	<ul style="list-style-type: none"> <li>Does not support cell arrays.</li> <li>If <code>pattern</code> does not exist in <code>str</code>, returns <code>zeros(1,0)</code> not <code>[]</code>. To check for an empty return, use <code>isempty</code>.</li> <li>Inputs must be character row vectors.</li> </ul>
strjust	MATLAB	—
strncmp	MATLAB	—

Name	Product	Remarks and Limitations
<code>strncmpi</code>	MATLAB	<ul style="list-style-type: none"> <li>Input values from the <code>char</code> class must be in the range 0-127.</li> </ul>
<code>strrep</code>	MATLAB	<ul style="list-style-type: none"> <li>Does not support cell arrays.</li> <li>If <code>oldSubstr</code> does not exist in <code>origStr</code>, returns <code>blanks(0)</code>. To check for an empty return, use <code>isempty</code>.</li> <li>Inputs must be character row vectors.</li> </ul>
<code>strtok</code>	MATLAB	—
<code>strtrim</code>	MATLAB	<ul style="list-style-type: none"> <li>Supports only inputs from the <code>char</code> class.</li> <li>Input values must be in the range 0-127.</li> </ul>
<code>struct</code>	MATLAB	—
<code>structfun</code>	MATLAB	<ul style="list-style-type: none"> <li>Does not support the <code>ErrorHandler</code> option.</li> <li>The number of outputs must be less than or equal to three.</li> </ul>
<code>sub</code>	Fixed-Point Designer	Code generation in MATLAB does not support the syntax <code>F.sub(a,b)</code> . You must use the syntax <code>sub(F,a,b)</code> .
<code>sub2ind</code>	MATLAB	<ul style="list-style-type: none"> <li>The first argument should be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
<code>subsasgn</code>	Fixed-Point Designer	—
<code>subspace</code>	MATLAB	—
<code>subsref</code>	Fixed-Point Designer	—
<code>sum</code>	MATLAB	<ul style="list-style-type: none"> <li>Specify <code>dim</code> as a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

Name	Product	Remarks and Limitations
sum	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.</li> </ul>
surfacegamma	Phased Array System Toolbox	Does not support variable-size inputs.
surfclutterrcs	Phased Array System Toolbox	Does not support variable-size inputs.
SURFPoints	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions. Supports MATLAB Function block: No To index locations with this object, use the syntax: <code>points.Location(idx, :)</code>, for <code>points</code> object. See <code>visionRecovertransformCodeGeneration_kernel.m</code>, which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.</p>
svd	MATLAB	Uses a different SVD implementation than MATLAB. Because the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
swapbytes	MATLAB	Inheritance of the class of the input to <code>swapbytes</code> in a MATLAB Function block is supported only when the class of the input is <code>double</code> . For non- <code>double</code> inputs, the input port data types must be specified, not inherited.
switch, case, otherwise	MATLAB	<ul style="list-style-type: none"> <li>If all case expressions are scalar integer values, generates a C <code>switch</code> statement. At run time, if the switch value is not an integer, generates an error.</li> <li>When the case expressions contain noninteger or nonscalar values, the code generation software generates C <code>if</code> statements in place of a C <code>switch</code> statement.</li> </ul>

Name	Product	Remarks and Limitations
systemp	Phased Array System Toolbox	Does not support variable-size inputs.
tan	MATLAB	—
tand	MATLAB	<ul style="list-style-type: none"> <li>• In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>• In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
tanh	MATLAB	—
taylorwin	Signal Processing Toolbox	<p>Inputs must be constant</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
tcdf	Statistics Toolbox	—
tf2ca	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tf2cl	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
time2range	Phased Array System Toolbox	Does not support variable-size inputs.
times	MATLAB	Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generation software does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, $(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Inf}i$ .

Name	Product	Remarks and Limitations
times	Fixed-Point Designer	<ul style="list-style-type: none"> <li>Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> <li>When you provide complex inputs to the <code>times</code> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the <b>Ports and data manager</b> and set the <b>Complexity</b> parameter for all known complex inputs to <code>On</code>.</li> </ul>
tinv	Statistics Toolbox	—
toeplitz	MATLAB	—
tpdf	Statistics Toolbox	—
trace	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Designer	—
trapz	MATLAB	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
triang	Signal Processing Toolbox	<p>Inputs must be constant</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
tril	MATLAB	<ul style="list-style-type: none"> <li>If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.</li> </ul>
tril	Fixed-Point Designer	<ul style="list-style-type: none"> <li>If supplied, the index, <code>k</code>, must be a real and scalar integer value that is not a <code>fi</code> object.</li> </ul>



Name	Product	Remarks and Limitations
triu	MATLAB	<ul style="list-style-type: none"> <li>If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.</li> </ul>
triu	Fixed-Point Designer	<ul style="list-style-type: none"> <li>If supplied, the index, <math>k</math>, must be a real and scalar integer value that is not a <code>fi</code> object.</li> </ul>
trnd	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>The output is nonscalar.</li> <li>An input parameter is invalid for the distribution.</li> </ul>
true	MATLAB	<ul style="list-style-type: none"> <li>Dimensions must be real, nonnegative, integers.</li> </ul>
tstat	Statistics Toolbox	—
tukeywin	Signal Processing Toolbox	<p>Inputs must be constant.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
typecast	MATLAB	<ul style="list-style-type: none"> <li>Value of string input argument <code>type</code> must be lowercase.</li> <li>When you use <code>typecast</code> with inheritance of input port data types in MATLAB Function blocks, you can receive a size error. To avoid this error, specify the block's input port data types explicitly.</li> <li>Integer input or result classes must map directly to a C type on the target hardware.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

Name	Product	Remarks and Limitations
<code>ufi</code>	Fixed-Point Designer	<ul style="list-style-type: none"> <li>All properties related to data type must be constant for code generation.</li> </ul>
<code>uint8, uint16, uint32, uint64</code>	MATLAB	No integer overflow detection for <code>int64</code> in MEX or MATLAB Function block simulation on Windows 32-bit platforms.
<code>uint8, uint16, uint32, uint64</code>	Fixed-Point Designer	—
<code>uminus</code>	MATLAB	—
<code>uminus</code>	Fixed-Point Designer	—
<code>unidcdf</code>	Statistics Toolbox	—
<code>unidinv</code>	Statistics Toolbox	—
<code>unidpdf</code>	Statistics Toolbox	—
<code>unidrnd</code>	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>The output is nonscalar.</li> <li>An input parameter is invalid for the distribution.</li> </ul>
<code>unidstat</code>	Statistics Toolbox	—
<code>unifcdf</code>	Statistics Toolbox	—
<code>unifinv</code>	Statistics Toolbox	—
<code>unifpdf</code>	Statistics Toolbox	—
<code>unifrnd</code>	Statistics Toolbox	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>The output is nonscalar.</li> <li>An input parameter is invalid for the distribution.</li> </ul>
<code>unifstat</code>	Statistics Toolbox	—
<code>unigrid</code>	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
union	MATLAB	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option: <ul style="list-style-type: none"> <li>• Inputs <i>A</i> and <i>B</i> must be vectors with the same orientation. If you specify the 'legacy' option, inputs <i>A</i> and <i>B</i> must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• The input <code>[]</code> is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' option and the 'rows' option, the outputs <i>ia</i> and <i>ib</i> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <i>C</i> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <i>C</i>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <i>x</i>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>

Name	Product	Remarks and Limitations
unique	MATLAB	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option:               <ul style="list-style-type: none"> <li>• The input <b>A</b> must be a vector. If you specify the 'legacy' option, the input <b>A</b> must be a row vector.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• The input <code>[ ]</code> is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'rows' option and the 'legacy' option, outputs <b>ia</b> and <b>ic</b> are column vectors. If these outputs are empty, they are 0-by-1, even if the output <b>C</b> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the input <b>A</b> must already be sorted in ascending order. The first output, <b>C</b>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> </ul>
unmkpp	MATLAB	<ul style="list-style-type: none"> <li>• <b>pp</b> must be a valid piecewise polynomial structure created by <code>mkpp</code>, <code>spline</code>, or <code>pchip</code> in MATLAB or by the code generation software.</li> <li>• Does not support <b>pp</b> structures created by <code>interp1</code> in MATLAB.</li> </ul>

Name	Product	Remarks and Limitations
unwrap	MATLAB	<ul style="list-style-type: none"> <li>• Row vector input is only supported when the first two inputs are vectors and nonscalar</li> <li>• Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors</li> </ul>
upfirdn	Signal Processing Toolbox	<ul style="list-style-type: none"> <li>• Code generation for this function requires the DSP System Toolbox software.</li> <li>• Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change.</li> </ul> <p style="text-align: center;"><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
uplus	MATLAB	—
uplus	Fixed-Point Designer	—
upper	MATLAB	<ul style="list-style-type: none"> <li>• Supports only <code>char</code> inputs.</li> <li>• Input values must be in the range 0-127.</li> </ul>
upperbound	Fixed-Point Designer	—
upsample	Signal Processing Toolbox	<p>Either declare input <code>n</code> as constant, or use the <code>assert</code> function in the calling function to set upper bounds for <code>n</code>. For example,</p> <pre>assert(n&lt;10)</pre>
uv2azel	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
uv2azelpat	Phased Array System Toolbox	Does not support variable-size inputs.
uv2phitheta	Phased Array System Toolbox	Does not support variable-size inputs.
uv2phithetapat	Phased Array System Toolbox	Does not support variable-size inputs.
val2ind	Phased Array System Toolbox	Does not support variable-size inputs.
vander	MATLAB	—
var	MATLAB	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
vertcat	Fixed-Point Designer	—
vision.AlphaBlender	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Autocorrelator	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Autothresher	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.BlobAnalysis	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.BoundaryTracer	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.CascadeObject-Detector	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.ChromaResampler	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Color-SpaceConverter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
vision.Connected-ComponentLabeler	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Convolver	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ContrastAdjuster	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Crosscorrelator	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Demosaic-Interpolator	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.DCT	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Deinterlacer	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Deployable	Computer Vision System Toolbox	Generates code on Windows host only. Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.DeployableVideo-Player	Computer Vision System Toolbox	Supports MATLAB Function block: Yes Generates code on Linux <sup>®</sup> and Windows platforms Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.EdgeDetector	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.FFT	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Foreground-Detector	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”

<b>Name</b>	<b>Product</b>	<b>Remarks and Limitations</b>
<code>vision.GammaCorrector</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.GeometricRotator</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.GeometricScaler</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.GeometricShearer</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.Geometric-Transformer</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.Geometric-Translator</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.Histogram</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.Histogram-BasedTracker</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.Histogram-Equalizer</code>		Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.HoughLines</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.HoughTransform</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.IDCT</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.IFFT</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.Image-Complementer</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.ImageFilter</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"
<code>vision.ImageDataType-Converter</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes "System Objects in MATLAB Code Generation"



Name	Product	Remarks and Limitations
vision.ImagePadder	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.KalmanFilter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.LocalMaxima-Finder	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MarkerInserter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Maximum	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Median	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MedianFilter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Mean	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Minimum	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Morphological-Close	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Morphological-Dilate	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Morphological-Erode	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Morphological-Open	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.PeopleDetector	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.PointTracker	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
<code>vision.PSNR</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.Pyramid</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.ShapeInserter</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.Standard-Deviation</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.TemplateMatcher</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.TextInserter</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.Variance</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<code>vision.VideoFileReader</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
<code>vision.VideoFileWriter</code>	Computer Vision System Toolbox	Supports MATLAB Function block: Yes Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
<code>wblcdf</code>	Statistics Toolbox	—
<code>wblinv</code>	Statistics Toolbox	—
<code>wblpdf</code>	Statistics Toolbox	—
<code>wblrnd</code>	Statistics Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
<code>wblstat</code>	Statistics Toolbox	—
<code>while</code>	MATLAB	—

Name	Product	Remarks and Limitations
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	—
xor	MATLAB	—
ycbcr2rgb	Image Processing Toolbox	—
yulewalk	Signal Processing Toolbox	<p>If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.</p> <p><b>Specifying constants</b></p> <p>To specify a constant input for <code>codegen</code>, use <code>coder.Constant</code>. For more information, see “Specify Constant Inputs at the Command Line”.</p>
zeros	MATLAB	<ul style="list-style-type: none"> <li>• Dimensions must be real, nonnegative, integers.</li> </ul>
zp2tf	MATLAB	—
zscore	Statistics Toolbox	—

# Functions and Objects Supported for C and C++ Code Generation — Category List

You can generate efficient C and C++ code for a subset of MATLAB built-in functions and toolbox functions, classes, and System objects that you call from MATLAB code. These functions, classes, and System objects are listed by MATLAB category or toolbox category in the following tables.

For an alphabetical list of supported functions, classes, and System objects, see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List”.

---

**Note:** For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB”.

---

### In this section...

- “Aerospace Toolbox” on page 4-136
- “Arithmetic Operations in MATLAB” on page 4-136
- “Bit-Wise Operations MATLAB” on page 4-137
- “Casting in MATLAB” on page 4-138
- “Communications System Toolbox” on page 4-138
- “Complex Numbers in MATLAB” on page 4-144
- “Computer Vision System Toolbox” on page 4-144
- “Control Flow in MATLAB” on page 4-153
- “Data and File Management in MATLAB” on page 4-154
- “Data Types in MATLAB” on page 4-157
- “Desktop Environment in MATLAB” on page 4-158
- “Discrete Math in MATLAB” on page 4-158
- “DSP System Toolbox” on page 4-159
- “Error Handling in MATLAB” on page 4-166
- “Exponents in MATLAB” on page 4-167
- “Filtering and Convolution in MATLAB” on page 4-167
- “Fixed-Point Designer” on page 4-168

**In this section...**

- “HDL Coder” on page 4-178
- “Histograms in MATLAB” on page 4-178
- “Image Acquisition Toolbox” on page 4-178
- “Image Processing in MATLAB” on page 4-178
- “Image Processing Toolbox” on page 4-179
- “Input and Output Arguments in MATLAB” on page 4-186
- “Interpolation and Computational Geometry in MATLAB” on page 4-187
- “Linear Algebra in MATLAB” on page 4-190
- “Logical and Bit-Wise Operations in MATLAB” on page 4-191
- “MATLAB Compiler” on page 4-191
- “Matrices and Arrays in MATLAB” on page 4-192
- “Neural Network Toolbox” on page 4-199
- “Nonlinear Numerical Methods in MATLAB” on page 4-199
- “Numerical Integration and Differentiation in MATLAB” on page 4-199
- “Optimization Functions in MATLAB” on page 4-200
- “Phased Array System Toolbox” on page 4-201
- “Polynomials in MATLAB” on page 4-209
- “Programming Utilities in MATLAB” on page 4-209
- “Relational Operators in MATLAB” on page 4-209
- “Rounding and Remainder Functions in MATLAB” on page 4-210
- “Set Operations in MATLAB” on page 4-210
- “Signal Processing in MATLAB” on page 4-215
- “Signal Processing Toolbox” on page 4-216
- “Special Values in MATLAB” on page 4-221
- “Specialized Math in MATLAB” on page 4-221
- “Statistics in MATLAB” on page 4-222
- “Statistics Toolbox” on page 4-222
- “String Functions in MATLAB” on page 4-231
- “Structures in MATLAB” on page 4-233

**In this section...**

“Trigonometry in MATLAB” on page 4-233

### Aerospace Toolbox

C and C++ code generation for the following Aerospace Toolbox quaternion functions requires the Aerospace Blockset software.

Function	Remarks and Limitations
quatconj	—
quatdivide	—
quatinv	—
quatmod	—
quatmultiply	—
quatnorm	—
quatnormalize	—

### Arithmetic Operations in MATLAB

See “Array vs. Matrix Operations” for detailed descriptions of the following operator equivalent functions.

Function	Remarks and Limitations
ctranspose	—
idivide	• For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.
isa	—
ldivide	—
minus	—
mldivide	—
mpower	—

Function	Remarks and Limitations
mrdivide	—
mtimes	<ul style="list-style-type: none"> <li>• Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generation software does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, <math>(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Inf}i</math>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
plus	—
power	<ul style="list-style-type: none"> <li>• Generates an error during simulation. When both X and Y are real, but <code>power(X,Y)</code> is complex, returns NaN in the generated code. To get the complex result, make the input value X complex by passing in <code>complex(X)</code>. For example, <code>power(complex(X),Y)</code>.</li> <li>• Generates an error during simulation. When both X and Y are real, but <code>X.^Y</code> is complex, returns NaN in generated code. To get the complex result, make the input value X complex by using <code>complex(X)</code>. For example, <code>complex(X).^Y</code>.</li> </ul>
rdivide	—
times	Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generation software does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, $(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Inf}i$ .
transpose	—
uminus	—
uplus	—

## Bit-Wise Operations MATLAB

Function	Remarks and Limitations
flintmax	—
swapbytes	Inheritance of the class of the input to <code>swapbytes</code> in a MATLAB Function block is supported only when the class of the input is <code>double</code> . For non-double inputs, the input port data types must be specified, not inherited.

## Casting in MATLAB

Function	Remarks and Limitations
cast	—
char	—
class	—
double	—
int8, int16, int32, int64	No integer overflow detection for <code>int64</code> in MEX or MATLAB Function block simulation on Windows 32-bit platforms.
logical	—
single	—
typecast	<ul style="list-style-type: none"> <li>• Value of string input argument <code>type</code> must be lowercase.</li> <li>• When you use <code>typecast</code> with inheritance of input port data types in MATLAB Function blocks, you can receive a size error. To avoid this error, specify the block's input port data types explicitly.</li> <li>• Integer input or result classes must map directly to a C type on the target hardware.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
uint8, uint16, uint32, uint64	No integer overflow detection for <code>int64</code> in MEX or MATLAB Function block simulation on Windows 32-bit platforms.

## Communications System Toolbox

C and C++ code generation for the following functions and System objects requires the Communications System Toolbox software.

Name	Remarks and Limitations
<b>Input and Output</b>	
<code>comm.BarkerCode</code>	“System Objects in MATLAB Code Generation”
<code>comm.GoldSequence</code>	“System Objects in MATLAB Code Generation”
<code>comm.HadamardCode</code>	“System Objects in MATLAB Code Generation”
<code>comm.KasamiSequence</code>	“System Objects in MATLAB Code Generation”
<code>comm.WalshCode</code>	“System Objects in MATLAB Code Generation”



Name	Remarks and Limitations
comm.PNSequence	“System Objects in MATLAB Code Generation”
lteZadoffChuSeq	—
<b>Signal and Delay Management</b>	
bi2de	—
de2bi	—
<b>Display and Visual Analysis</b>	
comm.ConstellationDiagram	“System Objects in MATLAB Code Generation”
dsp.ArrayPlot	“System Objects in MATLAB Code Generation”
dsp.SpectrumAnalyzer	“System Objects in MATLAB Code Generation”
dsp.TimeScope	“System Objects in MATLAB Code Generation”
<b>Source Coding</b>	
comm.DifferentialDecoder	“System Objects in MATLAB Code Generation”
comm.DifferentialEncoder	“System Objects in MATLAB Code Generation”
<b>Cyclic Redundancy Check Coding</b>	
comm.CRCDetector	“System Objects in MATLAB Code Generation”
comm.CRCGenerator	“System Objects in MATLAB Code Generation”
comm.HDLCRCDetector	“System Objects in MATLAB Code Generation”
comm.HDLCRCGenerator	“System Objects in MATLAB Code Generation”
<b>BCH Codes</b>	
comm.BCHDecoder	“System Objects in MATLAB Code Generation”
comm.BCHEncoder	“System Objects in MATLAB Code Generation”
<b>Reed-Solomon Codes</b>	
comm.RSDecoder	“System Objects in MATLAB Code Generation”
comm.RSEncoder	“System Objects in MATLAB Code Generation”
comm.HDLRSDecoder	“System Objects in MATLAB Code Generation”
comm.HDLRSEncoder	“System Objects in MATLAB Code Generation”
<b>LDPC Codes</b>	
comm.LDPCDecoder	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
comm.LDPCDecoder	“System Objects in MATLAB Code Generation”
<b>Convolutional Coding</b>	
comm.APPDecoder	“System Objects in MATLAB Code Generation”
comm.ConvolutionalEncoder	“System Objects in MATLAB Code Generation”
comm.TurboDecoder	“System Objects in MATLAB Code Generation”
comm.TurboEncoder	“System Objects in MATLAB Code Generation”
comm.ViterbiDecoder	“System Objects in MATLAB Code Generation”
istrellis	—
poly2trellis	—
<b>Signal Operations</b>	
comm.Descrambler	“System Objects in MATLAB Code Generation”
comm.Scrambler	“System Objects in MATLAB Code Generation”
<b>Interleaving</b>	
comm.AlgebraicDeinterleaver	“System Objects in MATLAB Code Generation”
comm.AlgebraicInterleaver	“System Objects in MATLAB Code Generation”
comm.BlockDeinterleaver	“System Objects in MATLAB Code Generation”
comm.BlockInterleaver	“System Objects in MATLAB Code Generation”
comm.ConvolutionalDeinterleaver	“System Objects in MATLAB Code Generation”
comm.ConvolutionalInterleaver	“System Objects in MATLAB Code Generation”
comm.HelicalDeinterleaver	“System Objects in MATLAB Code Generation”
comm.HelicalInterleaver	“System Objects in MATLAB Code Generation”
comm.MatrixDeinterleaver	“System Objects in MATLAB Code Generation”
comm.MatrixInterleaver	“System Objects in MATLAB Code Generation”
comm.MatrixHelicalScanDeinterleaver	“System Objects in MATLAB Code Generation”
comm.MatrixHelicalScanInterleaver	“System Objects in MATLAB Code Generation”
comm.MultiplexedDeinterleaver	“System Objects in MATLAB Code Generation”
comm.MultiplexedInterleaver	“System Objects in MATLAB Code Generation”
<b>Frequency Modulation</b>	

Name	Remarks and Limitations
comm.FSKDemodulator	“System Objects in MATLAB Code Generation”
comm.FSKModulator	“System Objects in MATLAB Code Generation”
<b>Phase Modulation</b>	
comm.BPSKDemodulator	“System Objects in MATLAB Code Generation”
comm.BPSKModulator	“System Objects in MATLAB Code Generation”
comm.DBPSKDemodulator	“System Objects in MATLAB Code Generation”
comm.DBPSKModulator	“System Objects in MATLAB Code Generation”
comm.DPSKDemodulator	“System Objects in MATLAB Code Generation”
comm.DPSKModulator	“System Objects in MATLAB Code Generation”
comm.DQPSKDemodulator	“System Objects in MATLAB Code Generation”
comm.DQPSKModulator	“System Objects in MATLAB Code Generation”
comm.OQPSKDemodulator	“System Objects in MATLAB Code Generation”
comm.OQPSKModulator	“System Objects in MATLAB Code Generation”
comm.PSKDemodulator	“System Objects in MATLAB Code Generation”
comm.PSKModulator	“System Objects in MATLAB Code Generation”
comm.QPSKDemodulator	“System Objects in MATLAB Code Generation”
comm.QPSKModulator	“System Objects in MATLAB Code Generation”
<b>Amplitude Modulation</b>	
comm.GeneralQAMDemodulator	“System Objects in MATLAB Code Generation”
comm.GeneralQAMModulator	“System Objects in MATLAB Code Generation”
comm.PAMDemodulator	“System Objects in MATLAB Code Generation”
comm.PAMModulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMDemodulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMModulator	“System Objects in MATLAB Code Generation”
<b>Continuous Phase Modulation</b>	
comm.CPFSKDemodulator	“System Objects in MATLAB Code Generation”
comm.CPFSKModulator	“System Objects in MATLAB Code Generation”
comm.CPMDemodulator	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
comm.CPMModulator	“System Objects in MATLAB Code Generation”
comm.GMSKDemodulator	“System Objects in MATLAB Code Generation”
comm.GMSKModulator	“System Objects in MATLAB Code Generation”
comm.MSKDemodulator	“System Objects in MATLAB Code Generation”
comm.MSKModulator	“System Objects in MATLAB Code Generation”
<b>Trellis Coded Modulation</b>	
comm.GeneralQAMTCMDemodulator	“System Objects in MATLAB Code Generation”
comm.GeneralQAMTCMModulator	“System Objects in MATLAB Code Generation”
comm.PSKTCMDemodulator	“System Objects in MATLAB Code Generation”
comm.PSKTCMModulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMTCMDemodulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMTCMModulator	“System Objects in MATLAB Code Generation”
<b>Orthogonal Frequency-Division Modulation</b>	
comm.OFDMDemodulator	“System Objects in MATLAB Code Generation”
comm.OFDMModulator	“System Objects in MATLAB Code Generation”
<b>Filtering</b>	
comm.IntegrateAndDumpFilter	“System Objects in MATLAB Code Generation”
comm.RaisedCosineReceiveFilter	“System Objects in MATLAB Code Generation”
comm.RaisedCosineTransmitFilter	“System Objects in MATLAB Code Generation”
<b>Carrier Phase Synchronization</b>	
comm.CPMCarrierPhaseSynchronizer	“System Objects in MATLAB Code Generation”
comm.PSKCarrierPhaseSynchronizer	“System Objects in MATLAB Code Generation”
<b>Timing Phase Synchronization</b>	
comm.EarlyLateGateTimingSynchronizer	“System Objects in MATLAB Code Generation”
comm.GardnerTimingSynchronizer	“System Objects in MATLAB Code Generation”
comm.GMSKTimingSynchronizer	“System Objects in MATLAB Code Generation”
comm.MSKTimingSynchronizer	“System Objects in MATLAB Code Generation”
comm.MuellerMullerTimingSynchronizer	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
<b>Synchronization Utilities</b>	
comm.DiscreteTimeVCO	“System Objects in MATLAB Code Generation”
<b>Equalization</b>	
comm.MLSEEqualizer	“System Objects in MATLAB Code Generation”
<b>MIMO</b>	
comm.LTEMIMOChannel	“System Objects in MATLAB Code Generation”
comm.MIMOChannel	“System Objects in MATLAB Code Generation”
comm.OSTBCCombiner	“System Objects in MATLAB Code Generation”
comm.OSTBCEncoder	“System Objects in MATLAB Code Generation”
comm.SphereDecoder	“System Objects in MATLAB Code Generation”
<b>Channel Modeling and RF Impairments</b>	
comm.AGC	“System Objects in MATLAB Code Generation”
comm.AWGNChannel	“System Objects in MATLAB Code Generation”
comm.BinarySymmetricChannel	“System Objects in MATLAB Code Generation”
comm.IQImbalanceCompensator	“System Objects in MATLAB Code Generation”
comm.LTEMIMOChannel	“System Objects in MATLAB Code Generation”
comm.MemorylessNonlinearity	“System Objects in MATLAB Code Generation”
comm.MIMOChannel	“System Objects in MATLAB Code Generation”
comm.PhaseFrequencyOffset	“System Objects in MATLAB Code Generation”
comm.PhaseNoise	“System Objects in MATLAB Code Generation”
comm.RayleighChannel	“System Objects in MATLAB Code Generation”
comm.RicianChannel	“System Objects in MATLAB Code Generation”
comm.ThermalNoise	“System Objects in MATLAB Code Generation”
comm.PSKCoarseFrequencyEstimator	“System Objects in MATLAB Code Generation”
comm.QAMCoarseFrequencyEstimator	“System Objects in MATLAB Code Generation”
iqcoef2imbal	—
iqimbal2coef	—
<b>Measurements and Analysis</b>	

Name	Remarks and Limitations
comm.ACPR	“System Objects in MATLAB Code Generation”
comm.CCDF	“System Objects in MATLAB Code Generation”
comm.ErrorRate	“System Objects in MATLAB Code Generation”
comm.EVM	“System Objects in MATLAB Code Generation”
comm.MER	“System Objects in MATLAB Code Generation”

### Complex Numbers in MATLAB

Function	Remarks and Limitations
complex	—
conj	—
imag	—
isnumeric	—
isreal	—
isscalar	—
real	—
unwrap	<ul style="list-style-type: none"> <li>• Row vector input is only supported when the first two inputs are vectors and nonscalar</li> <li>• Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors</li> </ul>

### Computer Vision System Toolbox

C and C++ code generation for the following functions and System objects requires the Computer Vision System Toolbox software.

Name	Remarks and Limitations
<b>Feature Detection, Extraction, and Matching</b>	
BRISKPoints	Compile-time constant inputs: No restriction Supports MATLAB Function block: No To index locations with this object, use the syntax: <code>points.Location(idx, :)</code> ,

Name	Remarks and Limitations
	for <code>points</code> object. See <code>visionRecoverFromCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
<code>cornerPoints</code>	Compile-time constant input: No restriction Supports MATLAB Function block: No To index locations with this object, use the syntax: <code>points.Location(idx,:)</code> , for <code>points</code> object. See <code>visionRecoverFromCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
<code>detectBRISKFeatures</code>	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
<code>detectFASTFeatures</code>	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
<code>detectHarrisFeatures</code>	Compile-time constant input: <code>FilterSize</code> Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
<code>detectMinEigenFeatures</code>	Compile-time constant input: <code>FilterSize</code> Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
<code>detectMSERFeatures</code>	Compile-time constant input: No restriction Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library. For code generation, the function outputs <code>regions.PixelList</code> as an array. The region sizes are defined in <code>regions.Lengths</code> .

Name	Remarks and Limitations
detectSURFFeatures	<p>Compile-time constant input: No restrictions            Supports MATLAB Function block: No            Generated code for this function uses a precompiled platform-specific shared library.</p>
extractFeatures	<p>Generates platform-dependent library: Yes for BRISK, FREAK, and SURF methods only.            Compile-time constant input: Method            Supports MATLAB Function block: Yes for Block method only.            Generated code for this function uses a precompiled platform-specific shared library.</p>
extractHOGFeatures	<p>Compile-time constant input: No            Supports MATLAB Function block: No</p>
matchFeatures	<p>Generates platform-dependent library: Yes for MATLAB host.            Generates portable C code for non-host target.            Compile-time constant input: Method and Metric.            Supports MATLAB Function block: Yes</p>
MSERRegions	<p>Compile-time constant input: No restrictions.            Supports MATLAB Function block: Yes            For code generation, you must specify both the pixellist cell array and the length of each array, as the second input. The object outputs, regions.PixelList as an array. The region sizes are defined in regions.Lengths.            Generated code for this function uses a precompiled platform-specific shared library.</p>



Name	Remarks and Limitations
SURFPoints	Compile-time constant input: No restrictions. Supports MATLAB Function block: No To index locations with this object, use the syntax: <code>points.Location(idx, :)</code> , for <code>points</code> object. See <code>visionRecovertransformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
vision.BoundaryTracer	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.EdgeDetector	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Image Registration and Geometric Transformations</b>	
estimateGeometricTransform	Compile-time constant input: <code>transformType</code> Supports MATLAB Function block: No
vision.GeometricRotator	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricScaler	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricShearer	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricTransformer	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricTranslator	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Object Detection and Recognition</b>	
ocr	Compile-time constant input: <code>TextLayout</code> , <code>Language</code> , and <code>CharacterSet</code> . Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.

Name	Remarks and Limitations
ocrText	Compile-time constant input: No restrictions. Supports MATLAB Function block: No
vision.PeopleDetector	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.CascadeObjectDetector	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
<b>Tracking and Motion Estimation</b>	
assignDetectionsToTracks	Compile-time constant input: No restriction. Supports MATLAB Function block: Yes
vision.BlockMatcher	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ForegroundDetector	Supports MATLAB Function block: No Generates platform-dependent library: Yes for MATLAB host. Generates portable C code for non-host target. Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.HistogramBasedTracker	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.KalmanFilter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.OpticalFlow	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.PointTracker	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.TemplateMatcher	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Camera Calibration and Stereo Vision</b>	

Name	Remarks and Limitations
bboxOverlapRatio	Compile-time constant input: No restriction Supports MATLAB Function block: No
disparity	Compile-time constant input: Method. Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
epipolarline	Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes
estimateFundamentalMatrix	Compile-time constant input: Method, OutputClass, DistanceType, and ReportRuntimeError. Supports MATLAB Function block: Yes
estimateUncalibratedRectification	Compile-time constant input: transformType Supports MATLAB Function block: No
isEpipoleInImage	Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes
lineToBorderPoints	Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes
selectStrongestBbox	Compile-time constant input: No restriction Supports MATLAB Function block: No
<b>Statistics</b>	
vision.Autocorrelator	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.BlobAnalysis	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Crosscorrelator	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Histogram	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.LocalMaximaFinder	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Maximum	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
vision.Mean	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Median	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Minimum	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.PSNR	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.StandardDeviation	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Variance	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Morphological Operations</b>	
vision.ConnectedComponentLabeler	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MorphologicalBottomHat	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MorphologicalClose	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MorphologicalDilate	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MorphologicalErode	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MorphologicalOpen	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MorphologicalTopHat	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Filters, Transforms, and Enhancements</b>	
integralImage	Supports MATLAB Function block: Yes
vision.Convolver	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
vision.ContrastAdjuster	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.DCT	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Deinterlacer	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.EdgeDetector	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.FFT	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.HistogramEqualizer	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.HoughLines	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.HoughTransform	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.IDCT	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.IFFT	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ImageFilter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.MedianFilter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Pyramid	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Video Loading, Saving, and Streaming</b>	
vision.DeployableVideoPlayer	Supports MATLAB Function block: Yes Generates code on Linux and Windows platforms Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
vision.VideoFileReader	Supports MATLAB Function block: Yes Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
vision.VideoFileWriter	Supports MATLAB Function block: Yes Generated code for this function uses a precompiled platform-specific shared library. “System Objects in MATLAB Code Generation”
<b>Color Space Formatting and Conversions</b>	
vision.Autothresher	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ChromaResampler	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ColorSpaceConverter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.DemosaicInterpolator	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GammaCorrector	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ImageComplementer	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ImageDataTypeConverter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ImagePadder	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
<b>Graphics</b>	
insertMarker	Compile-time constant input: marker Supports MATLAB Function block: Yes
insertShape	Compile-time constant input: shape and SmoothEdges Supports MATLAB Function block: Yes
vision.AlphaBlender	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
vision.MarkerInserter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ShapeInserter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.TextInserter	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

## Control Flow in MATLAB

Function	Remarks and Limitations
break	—
continue	—
end	—
for	—
if, elseif, else	—
parfor	<ul style="list-style-type: none"> <li>• Treated as a for-loop in a MATLAB Function block or when used with <code>fiaccel</code>.</li> <li>• See the <code>parfor</code> reference page in the MATLAB Coder documentation.</li> <li>• “Generate Code with Parallel for-Loops (<code>parfor</code>)”.</li> </ul>
return	—
switch, case, otherwise	<ul style="list-style-type: none"> <li>• If all case expressions are scalar integer values, generates a C <code>switch</code> statement. At run time, if the switch value is not an integer, generates an error.</li> <li>• When the case expressions contain noninteger or nonscalar values, the code generation software generates C <code>if</code> statements in place of a C <code>switch</code> statement.</li> </ul>
while	—

## Data and File Management in MATLAB

Function	Remarks and Limitations
computer	<ul style="list-style-type: none"> <li>Information about the computer on which the code generation software is running.</li> <li>Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>
fclose	—
feof	—
fopen	<ul style="list-style-type: none"> <li>Does not support: <ul style="list-style-type: none"> <li>machineformat, encoding, or fileID inputs</li> <li>message output</li> <li>fopen('all')</li> </ul> </li> <li>If you disable extrinsic calls, you cannot return fileIDs created with fopen to MATLAB or extrinsic functions. You can use such fileIDs only internally.</li> <li>When generating C/C++ executables, static libraries, or dynamic libraries, you can open up to 20 files.</li> <li>The generated code does not report errors from invalid file identifiers. Write your own file open error handling in your MATLAB code. Test whether fopen returns -1, which indicates that the file open failed. For example: <pre> ... fid = fopen(filename, 'r'); if fid == -1     % fopen failed  else     % fopen successful, okay to call fread A = fread(fid); ... </pre> </li> <li>The behavior of the generated code for fread is compiler-dependent if you: <ol style="list-style-type: none"> <li>Open a file using fopen with a permission of a+.</li> </ol> </li> </ul>



Function	Remarks and Limitations
	<p><b>2</b> Read the file using <code>fread</code> before calling an I/O function, such as <code>fseek</code> or <code>frewind</code>, that sets the file position indicator.</p>
<p><code>fprintf</code></p>	<ul style="list-style-type: none"> <li>• Does not support:                             <ul style="list-style-type: none"> <li>• <code>b</code> and <code>t</code> subtypes on <code>%u</code>, <code>%o</code> <code>%x</code>, and <code>%X</code> formats.</li> <li>• <code>\$</code> flag for reusing input arguments.</li> <li>• printing arrays.</li> </ul> </li> <li>• There is no automatic casting. Input arguments must match their format types for predictable results.</li> <li>• Escaped characters are limited to the decimal range of 0–127.</li> <li>• A call to <code>fprintf</code> with <code>fileID</code> equal to 1 or 2 becomes <code>printf</code> in the generated C/C++ code in the following cases:                             <ul style="list-style-type: none"> <li>• The <code>fprintf</code> call is inside a <code>parfor</code> loop.</li> <li>• Extrinsic calls are disabled.</li> </ul> </li> <li>• When the MATLAB behavior differs from the C compiler behavior, <code>fprintf</code> matches the C compiler behavior in the following cases:                             <ul style="list-style-type: none"> <li>• The format specifier has a corresponding C format specifier, for example, <code>%e</code> or <code>%E</code>.</li> <li>• The <code>fprintf</code> call is inside a <code>parfor</code> loop.</li> <li>• Extrinsic calls are disabled.</li> </ul> </li> <li>• When you call <code>fprintf</code> with the format specifier <code>%s</code>, do not put a null character in the middle of the input string. Use <code>fprintf(fid, '%c', char(0))</code> to write a null character.</li> <li>• When you call <code>fprintf</code> with an integer format specifier, the type of the integer argument must be a type that the target hardware can represent as a native C type. For example, if you call <code>fprintf('%d', int64(n))</code>, the target hardware must have a native C type that supports a 64-bit integer.</li> </ul>

Function	Remarks and Limitations
fread	<ul style="list-style-type: none"> <li>• precision must be a constant.</li> <li>• The source and output that precision specifies cannot have values long, ulong, unsigned long, bitN, or ubitN.</li> <li>• You cannot use the machineformat input.</li> <li>• If the source or output that precision specifies is a C type, for example, int, the target and production sizes for that type must:               <ul style="list-style-type: none"> <li>• Match.</li> <li>• Map directly to a MATLAB type.</li> </ul> </li> <li>• The source type that precision specifies must map directly to a C type on the target hardware.</li> <li>• If the fread call reads the entire file, all of the data must fit in the largest array available for code generation.</li> <li>• If sizeA is not constant or contains a nonfinite element, then dynamic memory allocation is required.</li> <li>• Treats a char value for source or output as a signed 8-bit integer. Use values between 0 and 127 only.</li> <li>• The generated code does not report file read errors. Write your own file read error handling in your MATLAB code. Test that the number of bytes read matches the number of bytes that you requested. For example:               <pre style="margin-left: 20px;"> ... N = 100; [vals, numRead] = fread(fid, N, '*double'); if numRead ~= N     % fewer elements read than expected end ... </pre> </li> </ul>
frewind	—

Function	Remarks and Limitations
load	<ul style="list-style-type: none"> <li>• Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use <code>coder.load</code>.</li> <li>• Does not support use of the function without assignment to a structure or array. For example, use <code>S = load(filename)</code>, not <code>load(filename)</code>.</li> <li>• The output <code>S</code> must be the name of a structure or array without any subscripting. For example, <code>S[i] = load('myFile.mat')</code> is not allowed.</li> <li>• Arguments to <code>load</code> must be compile-time constant strings.</li> <li>• Does not support loading objects.</li> <li>• If the MAT-file contains unsupported constructs, use <code>load(filename, variables)</code> to load only the supported constructs.</li> <li>• You cannot use <code>save</code> in a function intended for code generation. The code generation software does not support the <code>save</code> function. Furthermore, you cannot use <code>coder.extrinsic</code> with <code>save</code>. Prior to generating code, you can use <code>save</code> to save the workspace data to a MAT-file.</li> </ul> <p>You must use <code>coder.varsizes</code> to explicitly declare variable-size data loaded using the <code>load</code> function.</p>

## Data Types in MATLAB

Function	Remarks and Limitations
deal	—
iscell	—
isobject	—
nargchk	<ul style="list-style-type: none"> <li>• Output structure does not include stack information.</li> </ul> <hr/> <p><b>Note:</b> <code>nargchk</code> will be removed in a future release.</p>
narginchk	—
nargoutchk	—

Function	Remarks and Limitations
str2func	<ul style="list-style-type: none"> <li>String must be constant/known at compile time</li> </ul>
structfun	<ul style="list-style-type: none"> <li>Does not support the <code>ErrorHandler</code> option.</li> <li>The number of outputs must be less than or equal to three.</li> </ul>

### Desktop Environment in MATLAB

Function	Remarks and Limitations
ismac	<ul style="list-style-type: none"> <li>Returns true or false based on the MATLAB version used for code generation.</li> <li>Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>
ispc	<ul style="list-style-type: none"> <li>Returns true or false based on the MATLAB version you use for code generation.</li> <li>Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>
isunix	<ul style="list-style-type: none"> <li>Returns true or false based on the MATLAB version used for code generation.</li> <li>Use only when the code generation target is S-function (Simulation) or MEX-function.</li> </ul>

### Discrete Math in MATLAB

Function	Remarks and Limitations
factor	<ul style="list-style-type: none"> <li>The maximum double precision input is <math>2^{33}</math>.</li> <li>The maximum single precision input is <math>2^{25}</math>.</li> <li>The input <math>n</math> cannot have type <code>int64</code> or <code>uint64</code>.</li> </ul>
gcd	—
isprime	<ul style="list-style-type: none"> <li>The maximum double precision input is <math>2^{33}</math>.</li> <li>The maximum single precision input is <math>2^{25}</math>.</li> <li>The input <math>X</math> cannot have type <code>int64</code> or <code>uint64</code>.</li> </ul>
lcm	—

Function	Remarks and Limitations
nchoosek	<ul style="list-style-type: none"> <li>When the first input, <math>x</math>, is a scalar, nchoosek returns a binomial coefficient. In this case, <math>x</math> must be a nonnegative integer. It cannot have type <code>int64</code> or <code>uint64</code>.</li> <li>When the first input, <math>x</math>, is a vector, nchoosek treats it as a set. In this case, <math>x</math> can have type <code>int64</code> or <code>uint64</code>.</li> <li>The second input, <math>k</math>, cannot have type <code>int64</code> or <code>uint64</code>.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
primes	<ul style="list-style-type: none"> <li>The maximum double precision input is <math>2^{32}</math>.</li> <li>The maximum single precision input is <math>2^{24}</math>.</li> <li>The input <math>n</math> cannot have type <code>int64</code> or <code>uint64</code>.</li> </ul>

## DSP System Toolbox

C code generation for the following functions and System objects requires the DSP System Toolbox license. Many DSP System Toolbox functions require constant inputs for code generation. See “Define Constant Input Parameters in a Project” and “Specify Constant Inputs at the Command Line”.

Name	Remarks and Limitations
<b>Estimation</b>	
<code>dsp.BurgAREstimator</code>	“System Objects in MATLAB Code Generation”
<code>dsp.BurgSpectrumEstimator</code>	“System Objects in MATLAB Code Generation”
<code>dsp.CepstralToLPC</code>	“System Objects in MATLAB Code Generation”
<code>dsp.CrossSpectrumEstimator</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LevinsonSolver</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LPCToAutocorrelation</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LPCToCepstral</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LPCToLSF</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LPCToLSP</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LPCToRC</code>	“System Objects in MATLAB Code Generation”
<code>dsp.LSFToLPC</code>	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
dsp.LSPToLPC	“System Objects in MATLAB Code Generation”
dsp.RCToAutocorrelation	“System Objects in MATLAB Code Generation”
dsp.RCToLPC	“System Objects in MATLAB Code Generation”
dsp.SpectrumEstimator	“System Objects in MATLAB Code Generation”
dsp.TransferFunctionEstimator	“System Objects in MATLAB Code Generation”
<b>Filters</b>	
ca2tf	All inputs must be constant. Expressions or variables are allowed if their values do not change.
cl2tf	All inputs must be constant. Expressions or variables are allowed if their values do not change.
dsp.AdaptiveLatticeFilter	“System Objects in MATLAB Code Generation”
dsp.AffineProjectionFilter	“System Objects in MATLAB Code Generation”
dsp.AllpoleFilter	<ul style="list-style-type: none"> <li>• “System Objects in MATLAB Code Generation”</li> <li>• Only the <b>Denominator</b> property is tunable for code generation.</li> </ul>
dsp.BiquadFilter	“System Objects in MATLAB Code Generation”
dsp.CICCompensationDecimator	“System Objects in MATLAB Code Generation”
dsp.CICCompensationInterpolator	“System Objects in MATLAB Code Generation”
dsp.CICDecimator	“System Objects in MATLAB Code Generation”
dsp.CICInterpolator	“System Objects in MATLAB Code Generation”
dsp.FarrowRateConverter	“System Objects in MATLAB Code Generation”
dsp.FastTransversalFilter	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
dsp.FilterCascade	<ul style="list-style-type: none"> <li>• You cannot generate code directly from dsp.FilterCascade. You can use the generateFilteringCode method to generate a MATLAB function. You can generate C/C++ code from this MATLAB function.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.FilteredXLMSFilter	“System Objects in MATLAB Code Generation”
dsp.FIRDecimator	“System Objects in MATLAB Code Generation”
dsp.FIRFilter	<ul style="list-style-type: none"> <li>• “System Objects in MATLAB Code Generation”</li> <li>• Only the Numerator property is tunable for code generation.</li> </ul>
dsp.FIRHalfbandDecimator	“System Objects in MATLAB Code Generation”
dsp.FIRHalfbandInterpolator	“System Objects in MATLAB Code Generation”
dsp.FIRInterpolator	“System Objects in MATLAB Code Generation”
dsp.FIRRateConverter	“System Objects in MATLAB Code Generation”
dsp.FrequencyDomainAdaptiveFilter	“System Objects in MATLAB Code Generation”
dsp.IIRFilter	<ul style="list-style-type: none"> <li>• Only the Numerator and Denominator properties are tunable for code generation.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.KalmanFilter	“System Objects in MATLAB Code Generation”
dsp.LMSFilter	“System Objects in MATLAB Code Generation”
dsp.RLSFilter	“System Objects in MATLAB Code Generation”
dsp.SampleRateConverter	“System Objects in MATLAB Code Generation”
firceqip	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Remarks and Limitations
fireqint	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firgr	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
firhalfband	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firlpnorm	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
firminphase	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firnyquist	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firpr2chfb	All inputs must be constant. Expressions or variables are allowed if their values do not change.
ifir	All inputs must be constant. Expressions or variables are allowed if their values do not change.
iircomb	All inputs must be constant. Expressions or variables are allowed if their values do not change.



Name	Remarks and Limitations
iirgrpdelay	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
iirlpnorm	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
iirlpnormc	<ul style="list-style-type: none"> <li>• All inputs must be constant. Expressions or variables are allowed if their values do not change.</li> <li>• Does not support syntaxes that have cell array input.</li> </ul>
iirnotch	All inputs must be constant. Expressions or variables are allowed if their values do not change.
iirpeak	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tf2ca	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tf2cl	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<b>Math Operations</b>	
dsp.ArrayVectorAdder	“System Objects in MATLAB Code Generation”
dsp.ArrayVectorDivider	“System Objects in MATLAB Code Generation”
dsp.ArrayVectorMultiplier	“System Objects in MATLAB Code Generation”
dsp.ArrayVectorSubtractor	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
dsp.CumulativeProduct	“System Objects in MATLAB Code Generation”
dsp.CumulativeSum	“System Objects in MATLAB Code Generation”
dsp.LDLFactor	“System Objects in MATLAB Code Generation”
dsp.LevinsonSolver	“System Objects in MATLAB Code Generation”
dsp.LowerTriangularSolver	“System Objects in MATLAB Code Generation”
dsp.LUFactor	“System Objects in MATLAB Code Generation”
dsp.Normalizer	“System Objects in MATLAB Code Generation”
dsp.UpperTriangularSolver	“System Objects in MATLAB Code Generation”
<b>Quantizers</b>	
dsp.ScalarQuantizerDecoder	“System Objects in MATLAB Code Generation”
dsp.ScalarQuantizerEncoder	“System Objects in MATLAB Code Generation”
dsp.VectorQuantizerDecoder	“System Objects in MATLAB Code Generation”
dsp.VectorQuantizerEncoder	“System Objects in MATLAB Code Generation”
<b>Scopes</b>	
dsp.SpectrumAnalyzer	This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
dsp.TimeScope	This System object does not generate code. It is automatically declared as an <i>extrinsic</i> variable using the <code>coder.extrinsic</code> function.
<b>Signal Management</b>	
dsp.Counter	“System Objects in MATLAB Code Generation”
dsp.DelayLine	“System Objects in MATLAB Code Generation”
<b>Signal Operations</b>	
dsp.Convolver	“System Objects in MATLAB Code Generation”
dsp.DCBlocker	“System Objects in MATLAB Code Generation”
dsp.Delay	“System Objects in MATLAB Code Generation”
dsp.DigitalDownConverter	“System Objects in MATLAB Code Generation”
dsp.DigitalUpConverter	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
dsp.Interpolator	“System Objects in MATLAB Code Generation”
dsp.NCO	“System Objects in MATLAB Code Generation”
dsp.PeakFinder	“System Objects in MATLAB Code Generation”
dsp.PhaseExtractor	“System Objects in MATLAB Code Generation”
dsp.PhaseUnwrapper	“System Objects in MATLAB Code Generation”
dsp.VariableFractionalDelay	“System Objects in MATLAB Code Generation”
dsp.VariableIntegerDelay	“System Objects in MATLAB Code Generation”
dsp.Window	<ul style="list-style-type: none"> <li>• This object has no tunable properties for code generation.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.ZeroCrossingDetector	“System Objects in MATLAB Code Generation”
<b>Sinks</b>	
dsp.AudioPlayer	“System Objects in MATLAB Code Generation”
dsp.AudioFileWriter	“System Objects in MATLAB Code Generation”
dsp.UDPSender	“System Objects in MATLAB Code Generation”
<b>Sources</b>	
dsp.AudioFileReader	“System Objects in MATLAB Code Generation”
dsp.AudioRecorder	“System Objects in MATLAB Code Generation”
dsp.SignalSource	“System Objects in MATLAB Code Generation”
dsp.SineWave	<ul style="list-style-type: none"> <li>• This object has no tunable properties for code generation.</li> <li>• “System Objects in MATLAB Code Generation”</li> </ul>
dsp.UDPReceiver	“System Objects in MATLAB Code Generation”
<b>Statistics</b>	
dsp.Autocorrelator	“System Objects in MATLAB Code Generation”
dsp.Crosscorrelator	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
dsp.Histogram	<ul style="list-style-type: none"> <li>This object has no tunable properties for code generation.</li> <li>“System Objects in MATLAB Code Generation”</li> </ul>
dsp.Maximum	“System Objects in MATLAB Code Generation”
dsp.Mean	“System Objects in MATLAB Code Generation”
dsp.Median	“System Objects in MATLAB Code Generation”
dsp.Minimum	“System Objects in MATLAB Code Generation”
dsp.PeakToPeak	“System Objects in MATLAB Code Generation”
dsp.PeakToRMS	“System Objects in MATLAB Code Generation”
dsp.RMS	“System Objects in MATLAB Code Generation”
dsp.StandardDeviation	“System Objects in MATLAB Code Generation”
dsp.StateLevels	“System Objects in MATLAB Code Generation”
dsp.Variance	“System Objects in MATLAB Code Generation”
<b>Transforms</b>	
dsp.AnalyticSignal	“System Objects in MATLAB Code Generation”
dsp.DCT	“System Objects in MATLAB Code Generation”
dsp.FFT	“System Objects in MATLAB Code Generation”
dsp.IDCT	“System Objects in MATLAB Code Generation”
dsp.IFFT	“System Objects in MATLAB Code Generation”

### Error Handling in MATLAB

Function	Remarks and Limitations
assert	<ul style="list-style-type: none"> <li>Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time.</li> <li>For standalone code generation, excluded from the generated code.</li> <li>See “Rules for Using assert Function”.</li> </ul>
error	For standalone code generation, excluded from the generated code.

## Exponents in MATLAB

Function	Remarks and Limitations
exp	—
expm	—
expm1	—
factorial	—
log	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
log2	—
log10	—
log1p	—
nextpow2	—
nthroot	—
reallog	—
realpow	—
realsqrt	—
sqrt	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>

## Filtering and Convolution in MATLAB

Function	Remarks and Limitations
conv	—
conv2	—
convn	—
deconv	—

Function	Remarks and Limitations
detrend	<ul style="list-style-type: none"> <li>• If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> <li>• Be real.</li> <li>• Be sorted in ascending order.</li> <li>• Restrict elements to integers in the interval <math>[1, n-2]</math>. <math>n</math> is the number of elements in a column of input argument <code>X</code>, or the number of elements in <code>X</code> when <code>X</code> is a row vector.</li> <li>• Contain all unique values.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul> </li> </ul>
filter	—
filter2	—

### Fixed-Point Designer

In addition to function-specific limitations listed in the table, the following general limitations apply to the use of Fixed-Point Designer functions in generated code, with `fiaccel`:

- `fipref` and `quantizer` objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numericType` of a given `fi` variable after that variable has been created.
- The boolean value of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- You can use parallel for (`parfor`) loops in code compiled with `fiaccel`, but those loops are treated like regular `for` loops.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.

- The general limitations of C/C++ code generated from MATLAB apply. For more information, see “MATLAB Language Features Supported for C/C++ Code Generation”.

Function	Remarks/Limitations
abs	N/A
accumneg	N/A
accumpos	N/A
add	• Code generation in MATLAB does not support the syntax <code>F.add(a,b)</code> . You must use the syntax <code>add(F,a,b)</code> .
all	N/A
any	N/A
atan2	N/A
bitand	Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	N/A
bitcmp	N/A
bitconcat	N/A
bitget	N/A
bitor	Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	N/A
bitreplicate	N/A
bitrol	N/A
bitror	N/A
bitset	N/A
bitshift	N/A
bitsliceget	N/A
bitsll	Generated code may not handle out of range shifting.
bitsra	Generated code may not handle out of range shifting.
bitsrl	Generated code may not handle out of range shifting.
bitxor	Not supported for slope-bias scaled <code>fi</code> objects.

Function	Remarks/Limitations
bitxorreduce	N/A
ceil	N/A
complex	N/A
conj	N/A
conv	<ul style="list-style-type: none"> <li>• Variable-sized inputs are only supported when the <b>SumMode</b> property of the governing <b>fimath</b> is set to <b>Specify precision</b> or <b>Keep LSB</b>.</li> <li>• For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> <li>• In the generated code, the output for variable-sized signals is computed using the <b>SumMode</b> property of the governing <b>fimath</b>.</li> <li>• In MATLAB, the output for variable-sized signals is computed using the <b>SumMode</b> property of the governing <b>fimath</b> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <b>ProductMode</b> of the governing <b>fimath</b>.</li> </ul> </li> </ul>
convergent	N/A
cordicabs	Variable-size signals are not supported.
cordicangle	Variable-size signals are not supported.
cordicatan2	Variable-size signals are not supported.
cordiccart2pol	Variable-size signals are not supported.
cordicccexp	Variable-size signals are not supported.
cordicccos	Variable-size signals are not supported.
cordicpol2cart	Variable-size signals are not supported.
cordicrotate	Variable-size signals are not supported.
cordicsin	Variable-size signals are not supported.
cordicsincos	Variable-size signals are not supported.
cos	N/A
ctranspose	N/A



Function	Remarks/Limitations
diag	If supplied, the index, $k$ , must be a real and scalar integer value that is not a <code>fi</code> object.
divide	<ul style="list-style-type: none"> <li>Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> <li>Complex and imaginary divisors are not supported.</li> <li>Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.</li> </ul>
double	For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts. Instead of using casts, supply a replacement function. For more information, see “Function Replacements”.
end	N/A
eps	<ul style="list-style-type: none"> <li>Supported for scalar fixed-point signals only.</li> <li>Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.</li> </ul>
eq	Not supported for fixed-point signals with different biases.
fi	<ul style="list-style-type: none"> <li>The default constructor syntax without any input arguments is not supported.</li> <li>If the <code>numericType</code> is not fully specified, the input to <code>fi</code> must be a constant, a <code>fi</code>, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows <code>fi</code> to autoscale its fraction length based on the known data type of the input.</li> <li>All properties related to data type must be constant for code generation.</li> <li><code>numericType</code> object information must be available for nonfixed-point Simulink inputs.</li> </ul>
filter	<ul style="list-style-type: none"> <li>Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.</li> </ul>

Function	Remarks/Limitations
<code>fimath</code>	<ul style="list-style-type: none"> <li>Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>fimath</code> object. You define this object in the MATLAB Function block dialog in the Model Explorer.</li> <li>Use to create <code>fimath</code> objects in the generated code.</li> <li>If the <code>ProductMode</code> property of the <code>fimath</code> object is set to anything other than <code>FullPrecision</code>, the <code>ProductWordLength</code> and <code>ProductFractionLength</code> properties must be constant.</li> <li>If the <code>SumMode</code> property of the <code>fimath</code> object is set to anything other than <code>FullPrecision</code>, the <code>SumWordLength</code> and <code>SumFractionLength</code> properties must be constant.</li> </ul>
<code>fix</code>	N/A
<code>fixed.Quantizer</code>	N/A
<code>flip</code>	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
<code>fliplr</code>	N/A
<code>flipud</code>	N/A
<code>floor</code>	N/A
<code>for</code>	N/A
<code>ge</code>	Not supported for fixed-point signals with different biases.
<code>get</code>	The syntax <code>structure = get(0)</code> is not supported.
<code>getlsb</code>	N/A
<code>getmsb</code>	N/A
<code>gt</code>	Not supported for fixed-point signals with different biases.
<code>horzcat</code>	N/A
<code>imag</code>	N/A
<code>int8, int16, int32, int64</code>	N/A
<code>ipermute</code>	N/A
<code>iscolumn</code>	N/A
<code>isempty</code>	N/A

Function	Remarks/Limitations
<code>isequal</code>	N/A
<code>isfi</code>	Avoid using the <code>isfi</code> function in code that you intend to convert using the automated workflow. The value returned by <code>isfi</code> in the fixed-point code might differ from the value returned in the original MATLAB algorithm. The behavior of the fixed-point code might differ from the behavior of the original algorithm.
<code>isfimath</code>	N/A
<code>isfimathlocal</code>	N/A
<code>isfinite</code>	N/A
<code>isinf</code>	N/A
<code>isnan</code>	N/A
<code>isnumeric</code>	N/A
<code>isnumericitype</code>	N/A
<code>isreal</code>	N/A
<code>isrow</code>	N/A
<code>isscalar</code>	N/A
<code>assigned</code>	N/A
<code>isvector</code>	N/A
<code>le</code>	Not supported for fixed-point signals with different biases.
<code>length</code>	N/A
<code>logical</code>	N/A
<code>lowerbound</code>	N/A
<code>lsb</code>	<ul style="list-style-type: none"> <li>Supported for scalar fixed-point signals only.</li> <li>Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.</li> </ul>
<code>lt</code>	Not supported for fixed-point signals with different biases.
<code>max</code>	N/A
<code>mean</code>	N/A
<code>median</code>	N/A

Function	Remarks/Limitations
min	N/A
minus	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
mpower	<ul style="list-style-type: none"> <li>• When the exponent <code>k</code> is a variable and the input is a scalar, the <b>ProductMode</b> property of the governing <code>fimath</code> must be <b>SpecifyPrecision</b>.</li> <li>• When the exponent <code>k</code> is a variable and the input is not scalar, the <b>SumMode</b> property of the governing <code>fimath</code> must be <b>SpecifyPrecision</b>.</li> <li>• Variable-sized inputs are only supported when the <b>SumMode</b> property of the governing <code>fimath</code> is set to <b>SpecifyPrecision</b> or <b>Keep LSB</b>.</li> <li>• For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> <li>• In the generated code, the output for variable-sized signals is computed using the <b>SumMode</b> property of the governing <code>fimath</code>.</li> <li>• In MATLAB, the output for variable-sized signals is computed using the <b>SumMode</b> property of the governing <code>fimath</code> when the first input, <code>a</code>, is nonscalar. However, when <code>a</code> is a scalar, MATLAB computes the output using the <b>ProductMode</b> of the governing <code>fimath</code>.</li> </ul> </li> </ul>
mpy	<ul style="list-style-type: none"> <li>• Code generation in MATLAB does not support the syntax <code>F.mpy(a,b)</code>. You must use the syntax <code>mpy(F,a,b)</code>.</li> <li>• When you provide complex inputs to the <code>mpy</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the <b>Ports and data manager</b> and set the <b>Complexity</b> parameter for all known complex inputs to <b>On</b>.</li> </ul>
mrdivide	N/A

Function	Remarks/Limitations
mtimes	<ul style="list-style-type: none"> <li>• Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> <li>• Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>SpecifyPrecision</code> or <code>KeepLSB</code>.</li> <li>• For variable-sized signals, you may see different results between the generated code and MATLAB.                             <ul style="list-style-type: none"> <li>• In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>.</li> <li>• In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.</li> </ul> </li> </ul>
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	<code>numberofelements</code> will be removed in a future release. Use <code>numel</code> instead.
numel	N/A
numerictype	<ul style="list-style-type: none"> <li>• Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>numerictype</code> object that is populated with the signal's data type and scaling information.</li> <li>• Returns the data type when the input is a nonfixed-point signal.</li> <li>• Use to create <code>numerictype</code> objects in generated code.</li> <li>• All <code>numerictype</code> object properties related to the data type must be constant.</li> </ul>
permute	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
plus	Any non- <code>fi</code> inputs must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
pow2	N/A

Function	Remarks/Limitations
power	When the exponent <code>k</code> is a variable, the <code>ProductMode</code> property of the governing <code>fimath</code> must be <code>SpecifyPrecision</code> .
qr	N/A
quantize	N/A
range	N/A
rdivide	N/A
real	N/A
realmax	N/A
realmin	N/A
reinterpretcast	N/A
removefimath	N/A
repmat	The <code>dimensions</code> argument must be a built-in type; it cannot be a <code>fi</code> object.
rescale	N/A
reshape	N/A
rot90	In the syntax <code>rot90(A,k)</code> , the argument <code>k</code> must be a built-in type; it cannot be a <code>fi</code> object.
round	N/A
setfimath	N/A
sfi	<ul style="list-style-type: none"> <li>All properties related to data type must be constant for code generation.</li> </ul>
shiftdim	The <code>dimensions</code> argument must be a built-in type; it cannot be a <code>fi</code> object.
sign	N/A
sin	N/A
single	For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts. Instead of using casts, supply a replacement function. For more information, see “Function Replacements”.

Function	Remarks/Limitations
size	N/A
sort	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
squeeze	N/A
sqrt	<ul style="list-style-type: none"> <li>Complex and [Slope Bias] inputs error out.</li> <li>Negative inputs yield a 0 result.</li> </ul>
storedInteger	N/A
storedIntegerToDouble	N/A
sub	<ul style="list-style-type: none"> <li>Code generation in MATLAB does not support the syntax <code>F.sub(a,b)</code>. You must use the syntax <code>sub(F,a,b)</code>.</li> </ul>
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code> .
times	<ul style="list-style-type: none"> <li>Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.</li> <li>When you provide complex inputs to the <code>times</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the <b>Ports and data manager</b> and set the <b>Complexity</b> parameter for all known complex inputs to <code>On</code>.</li> </ul>
transpose	N/A
tril	If supplied, the index, $k$ , must be a real and scalar integer value that is not a <code>fi</code> object.
triu	If supplied, the index, $k$ , must be a real and scalar integer value that is not a <code>fi</code> object.
ufi	<ul style="list-style-type: none"> <li>All properties related to data type must be constant for code generation.</li> </ul>
uint8, uint16, uint32, uint64	N/A
uminus	N/A

Function	Remarks/Limitations
uplus	N/A
upperbound	N/A
vertcat	N/A

### HDL Coder

Function	Remarks and Limitations
hdl.RAM	This System object is available with MATLAB.

### Histograms in MATLAB

Function	Remarks and Limitations
hist	<ul style="list-style-type: none"><li>• Histogram bar plotting not supported; call with at least one output argument.</li><li>• If supplied, the second argument <i>x</i> must be a scalar constant.</li><li>• Inputs must be real.</li></ul>
histc	<ul style="list-style-type: none"><li>• The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.</li><li>• If supplied, <i>dim</i> must be a constant.</li><li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li></ul>

### Image Acquisition Toolbox

If you install Image Acquisition Toolbox software, you can generate C and C++ code for the VideoDevice System object. See `imaq.VideoDevice` and “Code Generation with VideoDevice System Object”.

### Image Processing in MATLAB

Function	Remarks and Limitations
im2double	—



## Image Processing Toolbox

The following table lists the Image Processing Toolbox functions that have been enabled for code generation. You must have the MATLAB Coder and Image Processing Toolbox software installed to generate C code from MATLAB for these functions.

Image Processing Toolbox provides three types of code generation support:

- Functions that generate C code.
- Functions that generate C code that depends on a platform-specific shared library (.dll, .so, or .dylib). Use of a shared library preserves performance optimizations in these functions, but this limits the target platforms for which you can generate code. For more information, see “Code Generation for Image Processing”.
- Functions that generate C code or C code that depends on a shared library, depending on which target platform you specify in MATLAB Coder. If you specify the generic **MATLAB Host Computer** target platform, these functions generate C code that depends on a shared library. If you specify any other target platform, these functions generate C code.

In generated code, each supported toolbox function has the same name, arguments, and functionality as its Image Processing Toolbox counterpart. However, some functions have limitations. The following table includes information about code generation limitations that might exist for each function. In the following table, all the functions generate C code. The table identifies those functions that generate C code that depends on a shared library, and those functions that can do both, depending on which target platform you choose.

Function	Remarks/Limitations
<code>affine2d</code>	When generating code, you can only specify single objects—arrays of objects are not supported.
<code>bwdist</code>	The <code>method</code> argument must be a compile-time constant. Input images must have fewer than $2^{32}$ pixels.  Generated code for this function uses a precompiled, “platform-specific shared library”.
<code>bwlookup</code>	For best results, specify an input image of class <code>logical</code> .  If you choose the generic <b>MATLAB Host Computer</b> target platform, generated code uses a precompiled, “platform-specific shared library”.

Function	Remarks/Limitations
<code>bwmorph</code>	<p>The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code>.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>bwpack</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>bwselect</code>	<p>Supports only the 3 and 4 input argument syntaxes: <code>BW2 = bwselect(BW,c,r)</code> and <code>BW2 = bwselect(BW,c,r,n)</code>. The optional fourth input argument, <code>n</code>, must be a compile-time constant. In addition, with code generation, <code>bwselect</code> only supports only the 1 and 2 output argument syntaxes: <code>BW2 = bwselect(___)</code> or <code>[BW2, idx] = bwselect(___)</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>bwtraceboundary</code>	The <code>dir</code> , <code>fstep</code> , and <code>conn</code> arguments must be compile-time constants.
<code>bwunpack</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>conndef</code>	Input arguments must be compile-time constants.
<code>edge</code>	<p>The <code>method</code>, <code>direction</code>, and <code>sigma</code> arguments must be a compile-time constants. In addition, nonprogrammatic syntaxes are not supported. For example, the syntax <code>edge(im)</code>, where <code>edge</code> does not return a value but displays an image instead, is not supported.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>fitgeotrans</code>	<p>The <code>transformtype</code> argument must be a compile-time constant. The function supports the following transformation types: <code>'nonreflectivesimilarity'</code>, <code>'similarity'</code>, <code>'affine'</code>, or <code>'projective'</code>.</p>
<code>fspecial</code>	Inputs must be compile-time constants. Expressions or variables are allowed if their values do not change.
<code>getrangefromclass</code>	—

Function	Remarks/Limitations
<code>histeq</code>	<p>All the syntaxes that include indexed images are not supported. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>im2uint8</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>im2uint16</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>im2int16</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>im2single</code>	—
<code>im2double</code>	—
<code>imadjust</code>	<p>Does not support syntaxes that include indexed images. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code>.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imbothat</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imclearborder</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant. Supports only up to 3-D inputs.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imclose</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imcomplement</code>	Does not support <code>int64</code> and <code>uint64</code> data types.

Function	Remarks/Limitations
<p><code>imdilate</code></p>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The <code>SE</code>, <code>PACKOPT</code>, and <code>SHAPE</code> input arguments must be a compile-time constant. The structuring element argument <code>SE</code> must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<p><code>imerode</code></p>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The <code>SE</code>, <code>PACKOPT</code>, and <code>SHAPE</code> input arguments must be a compile-time constant. The structuring element argument <code>SE</code> must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<p><code>imextendedmax</code></p>	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<p><code>imextendedmin</code></p>	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Function	Remarks/Limitations
imfill	<p>The optional input connectivity, <code>conn</code> and the string 'holes' must be compile-time constants.</p> <p>Supports only up to 3-D inputs.</p> <p>The interactive mode to select points, <code>imfill(BW,0,CONN)</code> is not supported in code generation.</p> <p><code>locations</code> can be a <math>P</math>-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a <math>P</math>-by-<code>ndims(I)</code> matrix, in which case each row contains the array indices of one of the starting locations. Once you select a format at compile-time, you cannot change it at run time. However, the number of points in <code>locations</code> can be varied at run time.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imfilter	<p>The input image can be either 2-D or 3-D. The value of the input argument, <code>options</code>, must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
imhist	<p>The optional second input argument, <code>n</code>, must be a compile-time constant. In addition, nonprogrammatic syntaxes are not supported. For example, the syntaxes where <code>imhist</code> displays the histogram are not supported.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
imhmax	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Function	Remarks/Limitations
<code>imhmin</code>	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imlincomb</code>	<p>The <code>output_class</code> argument must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imopen</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imquantize</code>	—
<code>imreconstruct</code>	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imref2d</code>	The <code>XWorldLimits</code> , <code>YWorldLimits</code> and <code>ImageSize</code> properties can be set only during object construction. When generating code, you can only specify single objects—arrays of objects are not supported.
<code>imref3d</code>	The <code>XWorldLimits</code> , <code>YWorldLimits</code> , <code>ZWorldLimits</code> and <code>ImageSize</code> properties can be set only during object construction. When generating code, you can only specify single objects—arrays of objects are not supported.
<code>imregionalmax</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>

Function	Remarks/Limitations
<code>imregionalmin</code>	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>imtophat</code>	<p>The input image <code>IM</code> must be either 2-D or 3-D image. The structuring element input argument <code>SE</code> must be a compile-time constant.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, “platform-specific shared library”.</p>
<code>imwarp</code>	<p>The geometric transformation object input, <code>tform</code>, must be either <code>affine2d</code> or <code>projective2d</code>. Additionally, the interpolation method and optional parameter names must be string constants.</p> <p>Generated code for this function uses a precompiled “platform-specific shared library”.</p>
<code>intlut</code>	Generated code for this function uses a precompiled “platform-specific shared library”.
<code>iptcheckconn</code>	Input arguments must be compile-time constants.
<code>iptcheckmap</code>	—
<code>label2rgb</code>	<p>Referring to the standard syntax:</p> <pre>RGB = label2rgb(L, map, zerocolor, order)</pre> <ul style="list-style-type: none"> <li>• Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>.</li> <li>• <code>map</code> must be an <code>n-by-3, double</code>, colormap matrix. You cannot use a string containing the name of a MATLAB colormap function or a function handle of a colormap function.</li> <li>• If you set the boundary color <code>zerocolor</code> to the same color as one of the regions, <code>label2rgb</code> will not issue a warning.</li> <li>• If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.</li> </ul>
<code>mean2</code>	—

Function	Remarks/Limitations
medfilt2	The <code>padopt</code> argument must be a compile-time constant.  Generated code for this function uses a precompiled “platform-specific shared library”.
multithresh	—
ordfilt2	The <code>padopt</code> argument must be a compile-time constant.  Generated code for this function uses a precompiled “platform-specific shared library”.
padarray	Support only up to 3-D inputs.  Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.
projective2d	When generating code, you can only specify single objects—arrays of objects are not supported.
rgb2ycbcr	—
strel	Input arguments must be compile-time constants. The following methods are not supported for code generation: <code>getsequence</code> , <code>reflect</code> , <code>translate</code> , <code>disp</code> , <code>display</code> , <code>loadobj</code> . When generating code, you can only specify single objects—arrays of objects are not supported.
stretchlim	Generated code for this function uses a precompiled “platform-specific shared library”.
ycbcr2rgb	—

## Input and Output Arguments in MATLAB

Function	Remarks and Limitations
nargin	—
nargout	<ul style="list-style-type: none"> <li>For a function with no output arguments, returns 1 if called without a terminating semicolon.</li> </ul>



Function	Remarks and Limitations
	<b>Note:</b> This behavior also affects extrinsic calls with no terminating semicolon. <code>nargout</code> is 1 for the called function in MATLAB.

## Interpolation and Computational Geometry in MATLAB

Function	Remarks and Limitations
<code>cart2pol</code>	—
<code>cart2sph</code>	—
<code>interp1</code>	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
<code>interp2</code>	<ul style="list-style-type: none"> <li>• <code>Xq</code> and <code>Yq</code> must be the same size. Use <code>meshgrid</code> to evaluate on a grid.</li> <li>• For best results, provide <code>X</code> and <code>Y</code> as vectors.</li> <li>• For the 'cubic' method, reports an error if the grid does not have uniform spacing. In this case, use the 'spline' method.</li> <li>• For best results when you use the 'spline' method:                             <ul style="list-style-type: none"> <li>• Use <code>meshgrid</code> to create the inputs <code>Xq</code> and <code>Yq</code>.</li> <li>• Use a small number of interpolation points relative to the dimensions of <code>V</code>. Interpolating over a large set of scattered points can be inefficient.</li> </ul> </li> </ul>
<code>interp3</code>	<ul style="list-style-type: none"> <li>• <code>Xq</code>, <code>Yq</code>, and <code>Zq</code> must be the same size. Use <code>meshgrid</code> to evaluate on a grid.</li> <li>• For best results, provide <code>X</code>, <code>Y</code>, and <code>Z</code> as vectors.</li> <li>• For the 'cubic' method, reports an error if the grid does not have uniform spacing. In this case, use the 'spline' method.</li> <li>• For best results when you use the 'spline' method:                             <ul style="list-style-type: none"> <li>• Use <code>meshgrid</code> to create the inputs <code>Xq</code>, <code>Yq</code>, and <code>Zq</code>.</li> <li>• Use a small number of interpolation points relative to the dimensions of <code>V</code>. Interpolating over a large set of scattered points can be inefficient.</li> </ul> </li> </ul>
<code>meshgrid</code>	—

Function	Remarks and Limitations
mkpp	<ul style="list-style-type: none"> <li>• The output structure <code>pp</code> differs from the <code>pp</code> structure in MATLAB. In MATLAB, <code>ppval</code> cannot use the <code>pp</code> structure from the code generation software. For code generation, <code>ppval</code> cannot use a <code>pp</code> structure created by MATLAB. <code>unmkpp</code> can use a MATLAB <code>pp</code> structure for code generation.</li> </ul> <p>To create a MATLAB <code>pp</code> structure from a <code>pp</code> structure created by the code generation software:</p> <ul style="list-style-type: none"> <li>• In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB.</li> <li>• In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.</li> </ul> <ul style="list-style-type: none"> <li>• If you do not provide <code>d</code>, then <code>coefs</code> must be two-dimensional and have a fixed number of columns. In this case, the number of columns is the order.</li> <li>• To define a piecewise constant polynomial, <code>coefs</code> must be a column vector or <code>d</code> must have at least two elements.</li> <li>• If you provide <code>d</code> and <code>d</code> is 1, <code>d</code> must be a constant. Otherwise, if the input to <code>ppval</code> is nonscalar, the shape of the output of <code>ppval</code> can differ from <code>ppval</code> in MATLAB.</li> <li>• If you provide <code>d</code>, it must have a fixed length. One of the following sets of statements must be true:             <ol style="list-style-type: none"> <li>1 Suppose that <code>m = length(d)</code> and <code>npieces = length(breaks) - 1</code>.                     <pre style="margin-left: 40px;">size(coefs,j) = d(j) size(coefs,m+1) = npieces size(coefs,m+2) = order j = 1,2,...,m. The dimension m+2 must be fixed length.</pre> </li> <li>2 Suppose that <code>m = length(d)</code> and <code>npieces = length(breaks) - 1</code>.                     <pre style="margin-left: 40px;">size(coefs,1) = prod(d)*npieces size(coefs,2) = order The second dimension must be fixed length.</pre> </li> </ol> </li> </ul> <ul style="list-style-type: none"> <li>• If you do not provide <code>d</code>, the following statements must be true:</li> </ul>

Function	Remarks and Limitations
	Suppose that $m = \text{length}(d)$ and $\text{npieces} = \text{length}(\text{breaks}) - 1$ . $\text{size}(\text{coefs},1) = \text{prod}(d) * \text{npieces}$ $\text{size}(\text{coefs},2) = \text{order}$ The second dimension must be fixed length.
pchip	<ul style="list-style-type: none"> <li>• Input <math>x</math> must be strictly increasing.</li> <li>• Does not remove <math>y</math> entries with NaN values.</li> <li>• If you generate code for the <code>pp = pchip(x,y)</code> syntax, you cannot input <code>pp</code> to the <code>ppval</code> function in MATLAB. To create a MATLAB <code>pp</code> structure from a <code>pp</code> structure created by the code generation software:                             <ul style="list-style-type: none"> <li>• In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB.</li> <li>• In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.</li> </ul> </li> </ul>
pol2cart	—
polyarea	—
ppval	The size of output $v$ does not match MATLAB when both of the following statements are true: <ul style="list-style-type: none"> <li>• The input <math>x</math> is a variable-size array that is not a variable-length vector.</li> <li>• <math>x</math> becomes a row vector at run time.</li> </ul> The code generation software does not remove the singleton dimensions. However, MATLAB might remove singleton dimensions.  For example, suppose that $x$ is a <code>:4-by-:5</code> array (the first dimension is variable size with an upper bound of 4 and the second dimension is variable size with an upper bound of 5). Suppose that <code>ppval(pp,0)</code> returns a <code>2-by-3</code> fixed-size array. $v$ has size <code>2-by-3-by-:4-by-:5</code> . At run time, suppose that, <code>size(x,1) = 1</code> and <code>size(x,2) = 5</code> . In the generated code, the size( $v$ ) is <code>[2,3,1,5]</code> . In MATLAB, the size is <code>[2,3,5]</code> .
rectint	—
sph2cart	—

Function	Remarks and Limitations
spline	<ul style="list-style-type: none"> <li>• Input <math>x</math> must be strictly increasing.</li> <li>• Does not remove <math>Y</math> entries with NaN values.</li> <li>• Does not report an error for infinite end slopes in <math>Y</math>.</li> <li>• If you generate code for the <code>pp = spline(x, Y)</code> syntax, you cannot input <code>pp</code> to the <code>ppval</code> function in MATLAB. To create a MATLAB <code>pp</code> structure from a <code>pp</code> structure created by the code generation software: <ul style="list-style-type: none"> <li>• In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB.</li> <li>• In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.</li> </ul> </li> </ul>
unmkpp	<ul style="list-style-type: none"> <li>• <code>pp</code> must be a valid piecewise polynomial structure created by <code>mkpp</code>, <code>spline</code>, or <code>pchip</code> in MATLAB or by the code generation software.</li> <li>• Does not support <code>pp</code> structures created by <code>interp1</code> in MATLAB.</li> </ul>

## Linear Algebra in MATLAB

Function	Remarks and Limitations
ishermitian	—
issymmetric	—
linsolve	<ul style="list-style-type: none"> <li>• The option structure must be a constant.</li> <li>• Supports only a scalar option structure input. It does not support arrays of option structures.</li> <li>• Only optimizes these cases: <ul style="list-style-type: none"> <li>• <code>UT</code></li> <li>• <code>LT</code></li> <li>• <code>UHES = true</code> (the <code>TRANSA</code> can be either <code>true</code> or <code>false</code>)</li> <li>• <code>SYM = true</code> and <code>POSDEF = true</code></li> </ul> </li> </ul> <p>Other options are equivalent to using <code>mldivide</code>.</p>
null	<ul style="list-style-type: none"> <li>• Might return a different basis than MATLAB</li> <li>• Does not support rational basis option (second input)</li> </ul>

Function	Remarks and Limitations
orth	• Can return a different basis than MATLAB
rsf2csf	—
schur	Can return a different Schur decomposition in generated code than in MATLAB.
sqrtm	—

## Logical and Bit-Wise Operations in MATLAB

Function	Remarks and Limitations
and	—
bitand	—
bitcmp	—
bitget	—
bitor	—
bitset	—
bitshift	—
bitxor	—
not	—
or	—
xor	—

## MATLAB Compiler

C and C++ code generation for the following functions requires the MATLAB Compiler software.

Function	Remarks and Limitations
isdeployed	<ul style="list-style-type: none"> <li>• Returns true and false as appropriate for MEX and SIM targets</li> <li>• Returns false for other targets</li> </ul>

Function	Remarks and Limitations
ismcc	<ul style="list-style-type: none"> <li>Returns true and false as appropriate for MEX and SIM targets.</li> <li>Returns false for other targets.</li> </ul>

## Matrices and Arrays in MATLAB

Function	Remarks and Limitations
abs	—
all	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
angle	—
any	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
blkdiag	—
bsxfun	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
cat	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
circshift	—
colon	<ul style="list-style-type: none"> <li>Does not accept complex inputs.</li> <li>The input <code>i</code> cannot have a logical value.</li> <li>Does not accept vector inputs.</li> <li>Inputs must be constants.</li> <li>Uses single-precision arithmetic to produce single-precision results.</li> </ul>
compan	—
cond	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
cov	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
cross	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
cumprod	<ul style="list-style-type: none"> <li>Does not support logical inputs. Cast input to <code>double</code> first.</li> <li>Does not support the <code>direction</code> argument.</li> </ul>

Function	Remarks and Limitations
cumsum	<ul style="list-style-type: none"> <li>• Does not support logical inputs. Cast input to <b>double</b> first.</li> <li>• Does not support the <b>direction</b> argument.</li> </ul>
det	—
diag	<ul style="list-style-type: none"> <li>• If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.</li> <li>• For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), <b>diag</b>:                         <ul style="list-style-type: none"> <li>• Treats the input as a vector input.</li> <li>• Returns a matrix with the given vector along the specified diagonal.</li> </ul> </li> <li>• For variable-size inputs that are not variable-length vectors, <b>diag</b>:                         <ul style="list-style-type: none"> <li>• Treats the input as a matrix.</li> <li>• Does not support inputs that are vectors at run time.</li> <li>• Returns a variable-length vector.</li> </ul> <p>If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, the output is 0-by-1 not 0-by-0. However, if the input is a constant size 0-by-0, the output is [ ].</p> </li> <li>• For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), <b>diag</b> treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force <b>diag</b> to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use:                         <ul style="list-style-type: none"> <li>• <code>diag(x(:))</code> instead of <code>diag(x)</code></li> <li>• <code>diag(x(:),k)</code> instead of <code>diag(x,k)</code></li> </ul> </li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

Function	Remarks and Limitations
<code>diff</code>	<ul style="list-style-type: none"> <li>If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
<code>dot</code>	—
<code>eig</code>	<ul style="list-style-type: none"> <li>For code generation, QZ algorithm is used in all cases. MATLAB can use different algorithms for different inputs. Consequently, <code>V</code> might represent a different basis of eigenvectors. The eigenvalues in <code>D</code> might not be in the same order as in MATLAB.</li> <li>With one input, <code>[V,D] = eig(A)</code>, the results are similar to those obtained using <code>[V,D] = eig(A,eye(size(A)), 'qz')</code> in MATLAB, except that for code generation, the columns of <code>V</code> are normalized.</li> <li>Options <code>'balance'</code>, and <code>'nobalance'</code> are not supported for the standard eigenvalue problem. <code>'chol'</code> is not supported for the symmetric generalized eigenvalue problem.</li> <li>Outputs are of complex type.</li> <li>Does not support the option to calculate left eigenvectors.</li> </ul>
<code>eye</code>	<code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>eye</code> method for other classes. For example, <code>eye(m, n, 'myclass')</code> does not invoke <code>myclass.eye(m,n)</code> .
<code>false</code>	<ul style="list-style-type: none"> <li>Dimensions must be real, nonnegative, integers.</li> </ul>
<code>find</code>	<ul style="list-style-type: none"> <li>Issues an error if a variable-sized input becomes a row vector at run time.</li> </ul> <hr/> <p><b>Note:</b> This limitation does not apply when the input is scalar or a variable-length row vector.</p> <hr/> <ul style="list-style-type: none"> <li>For variable-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or <code>[]</code> at run time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.</li> </ul>



Function	Remarks and Limitations
flip	—
flipdim	<b>Note:</b> flipdim will be removed in a future release. Use flip instead.
fliplr	—
flipud	—
full	—
hadamard	—
hankel	—
hilb	—
ind2sub	<ul style="list-style-type: none"> <li>• The first argument should be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
inv	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	—
ipermute	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
iscolumn	—
isempty	—
isequal	—
isequaln	—
isfinite	—
isfloat	—
isinf	—
isinteger	—
islogical	—
ismatrix	—
isnan	—
isrow	—

Function	Remarks and Limitations
issparse	—
isvector	—
kron	—
length	—
linspace	—
logspace	—
lu	—
magic	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
max	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
min	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
ndgrid	—
ndims	—
nnz	—
nonzeros	—
norm	—
normest	—
numel	—
ones	<ul style="list-style-type: none"> <li>• Dimensions must be real, nonnegative integers.</li> <li>• The input <code>optimfun</code> must be a function supported for code generation.</li> </ul>
pascal	—
permute	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
pinv	—
planerot	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”

Function	Remarks and Limitations
prod	<ul style="list-style-type: none"> <li>• If supplied, <code>dim</code> must be a constant.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
qr	—
rand	<ul style="list-style-type: none"> <li>• <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>rand</code> method for other classes. For example, <code>rand(sz, 'myclass')</code> does not invoke <code>myclass.rand(sz)</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
randi	<ul style="list-style-type: none"> <li>• <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>randi</code> method for other classes. For example, <code>randi(imax, sz, 'myclass')</code> does not invoke <code>myclass.randi(imax, sz)</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
randn	<ul style="list-style-type: none"> <li>• <code>classname</code> must be a built-in MATLAB numeric type. Does not invoke the static <code>randn</code> method for other classes. For example, <code>randn(sz, 'myclass')</code> does not invoke <code>myclass.randn(sz)</code>.</li> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
randperm	—
rank	—
rcond	—
repmat	—
reshape	<ul style="list-style-type: none"> <li>• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

Function	Remarks and Limitations
<code>rng</code>	<ul style="list-style-type: none"> <li>For library code generation targets, executable code generation targets, and MEX targets with extrinsic calls disabled: <ul style="list-style-type: none"> <li>Does not support the <code>'shuffle'</code> input.</li> <li>For the generator input, supports <code>'twister'</code>, <code>'v4'</code>, and <code>'v5normal'</code>.</li> </ul> </li> </ul> <p>For these targets, the output of <code>s=rng</code> in the generated code differs from the MATLAB output. You cannot return the output of <code>s=rng</code> from the generated code and pass it to <code>rng</code> in MATLAB.</p> <ul style="list-style-type: none"> <li>For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by <code>rng</code>.</li> </ul>
<code>rosser</code>	—
<code>rot90</code>	—
<code>shiftdim</code>	<ul style="list-style-type: none"> <li>Second argument must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
<code>sign</code>	—
<code>size</code>	—
<code>sort</code>	If the input is a complex type, <code>sort</code> orders the output according to absolute value. When <code>x</code> is a complex type that has all zero imaginary parts, use <code>sort(real(x))</code> to compute the sort order for real types. See “Code Generation for Complex Data”.
<code>sortrows</code>	If the input is a complex type, <code>sortrows</code> orders the output according to absolute value. When <code>x</code> is a complex type that has all zero imaginary parts, use <code>sortrows(real(x))</code> to compute the sort order for real types. See “Code Generation for Complex Data”.
<code>squeeze</code>	—
<code>sub2ind</code>	<ul style="list-style-type: none"> <li>The first argument should be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
<code>subspace</code>	—

Function	Remarks and Limitations
sum	<ul style="list-style-type: none"> <li>Specify <code>dim</code> as a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
toeplitz	—
trace	—
tril	<ul style="list-style-type: none"> <li>If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.</li> </ul>
triu	<ul style="list-style-type: none"> <li>If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.</li> </ul>
true	<ul style="list-style-type: none"> <li>Dimensions must be real, nonnegative, integers.</li> </ul>
vander	—
wilkinson	—
zeros	<ul style="list-style-type: none"> <li>Dimensions must be real, nonnegative, integers.</li> </ul>

## Neural Network Toolbox

You can use `genFunction` in the Neural Network Toolbox™ to generate a standalone MATLAB function for a trained neural network. You can generate C/C++ code from this standalone MATLAB function. To generate Simulink blocks, use `theGenSim` function. See “Deploy Neural Network Functions”.

## Nonlinear Numerical Methods in MATLAB

Function	Remarks and Limitations
quad2d	<ul style="list-style-type: none"> <li>Generates a warning if the size of the internal storage arrays is not large enough. If a warning occurs, a possible workaround is to divide the region of integration into pieces and sum the integrals over each piece.</li> </ul>
quadgk	—

## Numerical Integration and Differentiation in MATLAB

Function	Remarks and Limitations
cumtrapz	—

Function	Remarks and Limitations
<code>del2</code>	—
<code>diff</code>	<ul style="list-style-type: none"> <li>If supplied, the arguments representing the number of times to apply <code>diff</code> and the dimension along which to calculate the difference must be constants.</li> </ul>
<code>gradient</code>	—
<code>ode23</code>	<ul style="list-style-type: none"> <li>All <code>odeset</code> option arguments must be constant.</li> <li>Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function .</li> <li>You must provide at least the two output arguments <code>T</code> and <code>Y</code>.</li> <li>Input types must be homogeneous—all double or all single.</li> <li>Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.</li> </ul>
<code>ode45</code>	<ul style="list-style-type: none"> <li>All <code>odeset</code> option arguments must be constant.</li> <li>Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function .</li> <li>You must provide at least the two output arguments <code>T</code> and <code>Y</code>.</li> <li>Input types must be homogeneous—all double or all single.</li> <li>Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.</li> </ul>
<code>odeget</code>	The <code>name</code> argument must be constant.
<code>odeset</code>	All inputs must be constant.
<code>trapz</code>	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

## Optimization Functions in MATLAB

Function	Remarks and Limitations
<code>fminsearch</code>	<ul style="list-style-type: none"> <li> Ignores the <code>Display</code> option. Does not print status information during execution. Test the <code>exitflag</code> output for the exit condition.</li> <li>The output structure does not include the <code>algorithm</code> or <code>message</code> fields.</li> </ul>

Function	Remarks and Limitations
	<ul style="list-style-type: none"> <li>• Ignores the <code>OutputFcn</code> and <code>PlotFcns</code> options.</li> </ul>
<code>fzero</code>	<ul style="list-style-type: none"> <li>• The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument.</li> <li>• Supports up to three output arguments. Does not support the fourth output argument (the <code>output</code> structure).</li> </ul>
<code>optimget</code>	Input parameter names must be constant.
<code>optimset</code>	<ul style="list-style-type: none"> <li>• Does not support the syntax that has no input or output arguments: <code>optimset</code></li> <li>• Functions specified in the options must be supported for code generation.</li> <li>• The fields of the options structure <code>oldopts</code> must be fixed-size fields.</li> <li>• For code generation, optimization functions ignore the <code>Display</code> option.</li> <li>• Does not support the additional options in an options structure created by the Optimization Toolbox <code>optimset</code> function. If an input options structure includes the additional Optimization Toolbox options, the output structure does not include them.</li> </ul>

## Phased Array System Toolbox

C and C++ code generation for the following functions requires the Phased Array System Toolbox software.

Name	Remarks and Limitations
<b>Antenna and Microphone Elements</b>	
<code>aperture2gain</code>	Does not support variable-size inputs.
<code>azel2phithetapat</code>	Does not support variable-size inputs.
<code>azel2uvpat</code>	Does not support variable-size inputs.
<code>circpol2pol</code>	Does not support variable-size inputs.
<code>gain2aperture</code>	Does not support variable-size inputs.
<code>phased.CosineAntennaElement</code>	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>

Name	Remarks and Limitations
phased.CrossedDipoleAntennaElement	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.CustomAntennaElement	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.CustomMicrophoneElement	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.IsotropicAntennaElement	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.OmnidirectionalMicrophoneElement	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.ShortDipoleAntennaElement	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phitheta2azelpat	Does not support variable-size inputs.
phitheta2uvpat	Does not support variable-size inputs.
pol2circpol	Does not support variable-size inputs.
polellip	Does not support variable-size inputs.
polloss	Does not support variable-size inputs.
polratio	Does not support variable-size inputs.
polsignature	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• Supported only when output arguments are specified.</li> </ul>
stokes	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• Supported only when output arguments are specified.</li> </ul>



Name	Remarks and Limitations
uv2azelpat	Does not support variable-size inputs.
uv2phithetapat	Does not support variable-size inputs.
<b>Array Geometries and Analysis</b>	
az2broadside	Does not support variable-size inputs.
broadside2az	Does not support variable-size inputs.
phased.ArrayGain	<ul style="list-style-type: none"> <li>• Does not support arrays containing polarized antenna elements, that is, the <code>phased.ShortDipoleAntennaElement</code> or <code>phased.CrossedDipoleAntennaElement</code> antennas.</li> <li>• “Code Generation”.</li> </ul>
phased.ArrayResponse	“Code Generation”.
phased.ConformalArray	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.ElementDelay	“Code Generation”.
phased.PartitionedArray	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.ReplicatedSubarray	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.SteeringVector	See “Code Generation”.
phased.ULA	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.URA	<ul style="list-style-type: none"> <li>• <code>plotResponse</code> and <code>viewArray</code> methods are not supported.</li> <li>• “Code Generation”.</li> </ul>
<b>Signal Radiation and Collection</b>	

Name	Remarks and Limitations
phased.Collector	“Code Generation”.
phased.Radiator	“Code Generation”.
phased.WidebandCollector	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
sensorsig	Does not support variable-size inputs.
<b>Waveforms</b>	
ambgfun	Does not support variable-size inputs.
phased.FMCWaveform	“Code Generation”.
phased.LinearFMWaveform	“Code Generation”.
phased.PhaseCodedWaveform	“Code Generation”.
phased.RectangularWaveform	“Code Generation”.
phased.SteppedFMWaveform	“Code Generation”.
range2bw	Does not support variable-size inputs.
range2time	Does not support variable-size inputs.
time2range	Does not support variable-size inputs.
unigrid	Does not support variable-size inputs.
<b>Transmitters and Receivers</b>	
delayseq	Does not support variable-size inputs.
noisepow	Does not support variable-size inputs.
phased.ReceiverPreamp	“Code Generation”.
phased.Transmitter	“Code Generation”.
systemp	Does not support variable-size inputs.
<b>Beamforming</b>	
cbfweights	Does not support variable-size inputs.
lcmvweights	Does not support variable-size inputs.
mvdweights	Does not support variable-size inputs.

Name	Remarks and Limitations
phased.FrostBeamformer	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
phased.LCMVBeamformer	“Code Generation”.
phased.MVDRBeamformer	“Code Generation”.
phased.PhaseShiftBeamformer	“Code Generation”.
phased.SteeringVector	“Code Generation”.
phased.SubbandPhaseShiftBeamformer	“Code Generation”.
phased.TimeDelayBeamformer	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
phased.TimeDelayLCMVBeamformer	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
sensorcov	Does not support variable-size inputs.
steervec	Does not support variable-size inputs.
<b>Direction of Arrival (DOA) Estimation</b>	
aicstest	Does not support variable-size inputs.
espritdoa	Does not support variable-size inputs.
mdltest	Does not support variable-size inputs.
phased.BeamspaceEstimator	“Code Generation”.
phased.BeamspaceEstimator2D	“Code Generation”.
phased.BeamspaceESPRITEstimator	“Code Generation”.
phased.ESPRITEstimator	“Code Generation”.
phased.MVDREstimator	“Code Generation”.
phased.MVDREstimator2D	“Code Generation”.

Name	Remarks and Limitations
phased.RootMUSICEstimator	“Code Generation”.
phased.RootWSFEstimator	“Code Generation”.
phased.SumDifferenceMonopulseTracker	“Code Generation”.
phased.SumDifferenceMonopulseTracker2D	“Code Generation”.
rootmusicdoa	Does not support variable-size inputs.
spsmooth	Does not support variable-size inputs.
<b>Space-Time Adaptive Processing (STAP)</b>	
dopsteeringvec	Does not support variable-size inputs.
phased.ADPCACanceller	“Code Generation”.
phased.AngleDopplerResponse	“Code Generation”.
phased.DPCACanceller	“Code Generation”.
phased.STAPSMIBeamformer	“Code Generation”.
val2ind	Does not support variable-size inputs.
<b>Signal Propagation and Environment</b>	
billingsleyicm	Does not support variable-size inputs.
depressionang	Does not support variable-size inputs.
effearthradius	Does not support variable-size inputs.
fspl	Does not support variable-size inputs.
grazingang	Does not support variable-size inputs.
horizonrange	Does not support variable-size inputs.
phased.BarrageJammer	“Code Generation”.
phased.ConstantGammaClutter	“Code Generation”.
phased.FreeSpace	<ul style="list-style-type: none"> <li>• Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”.</li> <li>• “Code Generation”.</li> </ul>
phased.RadarTarget	“Code Generation”.
physconst	Does not support variable-size inputs.

Name	Remarks and Limitations
surfacegamma	Does not support variable-size inputs.
surfcluttercs	Does not support variable-size inputs.
<b>Detection and System Analysis</b>	
albersheim	Does not support variable-size inputs.
beat2range	Does not support variable-size inputs.
dechirp	Does not support variable-size inputs.
npwgnthresh	Does not support variable-size inputs.
phased.CFARDetector	“Code Generation”.
phased.MatchedFilter	<ul style="list-style-type: none"> <li>• The CustomSpectrumWindow property is not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.RangeDopplerResponse	<ul style="list-style-type: none"> <li>• The CustomRangeWindow and the CustomDopplerWindow properties are not supported.</li> <li>• “Code Generation”.</li> </ul>
phased.StretchProcessor	“Code Generation”.
phased.TimeVaryingGain	“Code Generation”.
pulsint	Does not support variable-size inputs.
radareqpow	Does not support variable-size inputs.
radareqrng	Does not support variable-size inputs.
radareqsnr	Does not support variable-size inputs.
radarvcd	Does not support variable-size inputs.
range2beat	Does not support variable-size inputs.
rdcoupling	Does not support variable-size inputs.
rocpfa	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• The NonfluctuatingNoncoherent signal type is not supported.</li> </ul>

Name	Remarks and Limitations
rocsnr	<ul style="list-style-type: none"> <li>• Does not support variable-size inputs.</li> <li>• The <code>NonfluctuatingNoncoherent</code> signal type is not supported.</li> </ul>
shnidman	Does not support variable-size inputs.
stretchfreq2rng	Does not support variable-size inputs.
<b>Motion Modeling and Coordinate Systems</b>	
azel2phitheta	Does not support variable-size inputs.
azel2uv	Does not support variable-size inputs.
azelaxes	Does not support variable-size inputs.
cart2sphvec	Does not support variable-size inputs.
dop2speed	Does not support variable-size inputs.
global2localcoord	Does not support variable-size inputs.
local2globalcoord	Does not support variable-size inputs.
phased.Platform	“Code Generation”.
phitheta2azel	Does not support variable-size inputs.
phitheta2uv	Does not support variable-size inputs.
radialspeed	Does not support variable-size inputs.
rangeangle	Does not support variable-size inputs.
rotx	Does not support variable-size inputs.
roty	Does not support variable-size inputs.
rotz	Does not support variable-size inputs.
speed2dop	Does not support variable-size inputs.
sph2cartvec	Does not support variable-size inputs.
uv2azel	Does not support variable-size inputs.
uv2phitheta	Does not support variable-size inputs.

## Polynomials in MATLAB

Function	Remarks and Limitations
poly	<ul style="list-style-type: none"> <li>Does not discard nonfinite input values</li> <li>Complex input produces complex output</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
polyder	The output can contain fewer NaNs than the MATLAB output. However, if the input contains a NaN, the output contains at least one NaN.
polyfit	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
polyint	—
polyval	—
polyvalm	—
roots	<ul style="list-style-type: none"> <li>Output is variable size.</li> <li>Output is complex.</li> <li>Roots are not always in the same order as MATLAB.</li> <li>Roots of poorly conditioned polynomials do not always match MATLAB.</li> </ul>

## Programming Utilities in MATLAB

Function	Remarks and Limitations
mfilename	—

## Relational Operators in MATLAB

Function	Remarks and Limitations
eq	—
ge	—
gt	—
le	—
lt	—
ne	—

## Rounding and Remainder Functions in MATLAB

Function	Remarks and Limitations
ceil	—
fix	—
floor	—
mod	<ul style="list-style-type: none"> <li>Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors.</li> </ul> <p>If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.</p>
rem	<ul style="list-style-type: none"> <li>Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors.</li> <li>If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.</li> </ul>
round	—

## Set Operations in MATLAB

Function	Remarks and Limitations
intersect	<ul style="list-style-type: none"> <li>When you do not specify the <code>'rows'</code> option: <ul style="list-style-type: none"> <li>Inputs <code>A</code> and <code>B</code> must be vectors. If you specify the <code>'legacy'</code> option, inputs <code>A</code> and <code>B</code> must be row vectors.</li> <li>The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>The input <code>[]</code> is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>If you specify the <code>'legacy'</code> option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>When you specify both the <code>'legacy'</code> option and the <code>'rows'</code> option, the outputs <code>ia</code> and <code>ib</code> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <code>C</code> is 0-by-0.</li> </ul>



Function	Remarks and Limitations
	<ul style="list-style-type: none"> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following:                             <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <code>x</code>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>
<code>ismember</code>	<ul style="list-style-type: none"> <li>• The second input, <code>B</code>, must be sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> </ul>
<code>issorted</code>	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”

Function	Remarks and Limitations
setdiff	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option:               <ul style="list-style-type: none"> <li>• Inputs <b>A</b> and <b>B</b> must be vectors. If you specify the 'legacy' option, inputs <b>A</b> and <b>B</b> must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• Do not use <code>[]</code> to represent the empty set. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' and 'rows' options, the output <b>ia</b> is a column vector. If <b>ia</b> is empty, it is 0-by-1, never 0-by-0, even if the output <b>C</b> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <b>C</b>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following:               <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <b>x</b>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>

Function	Remarks and Limitations
setxor	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option:                             <ul style="list-style-type: none"> <li>• Inputs <b>A</b> and <b>B</b> must be vectors with the same orientation. If you specify the 'legacy' option, inputs <b>A</b> and <b>B</b> must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• The input [ ] is not supported. Use a 1-by-0 or 0-by-1 input, for example , <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' option and the 'rows' option, the outputs <b>ia</b> and <b>ib</b> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <b>C</b> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' flag, the inputs must already be sorted in ascending order. The first output, <b>C</b>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following:                             <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <b>x</b>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>

Function	Remarks and Limitations
union	<ul style="list-style-type: none"> <li>• When you do not specify the 'rows' option:               <ul style="list-style-type: none"> <li>• Inputs <b>A</b> and <b>B</b> must be vectors with the same orientation. If you specify the 'legacy' option, inputs <b>A</b> and <b>B</b> must be row vectors.</li> <li>• The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>• The input <code>[]</code> is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>• If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>• When you specify both the 'legacy' option and the 'rows' option, the outputs <b>ia</b> and <b>ib</b> are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output <b>C</b> is 0-by-0.</li> <li>• When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the inputs must already be sorted in ascending order. The first output, <b>C</b>, is sorted in ascending order.</li> <li>• Complex inputs must be <code>single</code> or <code>double</code>.</li> <li>• When one input is complex and the other input is real, do one of the following:               <ul style="list-style-type: none"> <li>• Set <code>setOrder</code> to 'stable'.</li> <li>• Sort the real input in complex ascending order (by absolute value). Suppose the real input is <b>x</b>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</li> </ul> </li> </ul>

Function	Remarks and Limitations
unique	<ul style="list-style-type: none"> <li>When you do not specify the 'rows' option:                             <ul style="list-style-type: none"> <li>The input <b>A</b> must be a vector. If you specify the 'legacy' option, the input <b>A</b> must be a row vector.</li> <li>The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.</li> <li>The input [ ] is not supported. Use a 1-by-0 or 0-by-1 input, for example, <code>zeros(1,0)</code>, to represent the empty set.</li> <li>If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0.</li> </ul> </li> <li>When you specify both the 'rows' option and the 'legacy' option, outputs <b>ia</b> and <b>ic</b> are column vectors. If these outputs are empty, they are 0-by-1, even if the output <b>C</b> is 0-by-0.</li> <li>When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the input <b>A</b> must already be sorted in ascending order. The first output, <b>C</b>, is sorted in ascending order.</li> <li>Complex inputs must be <code>single</code> or <code>double</code>.</li> </ul>

## Signal Processing in MATLAB

Function	Remarks and Limitations
chol	—
conv	—
fft	<ul style="list-style-type: none"> <li>Length of input vector must be a power of 2.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
fft2	<ul style="list-style-type: none"> <li>Length of input matrix dimensions must each be a power of 2.</li> </ul>
fftn	<ul style="list-style-type: none"> <li>Length of input matrix dimensions must each be a power of 2.</li> </ul>
fftshift	—
filter	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li><code>v</code></li> </ul>
freqspace	—

Function	Remarks and Limitations
ifft	<ul style="list-style-type: none"> <li>Length of input vector must be a power of 2.</li> <li>Output of ifft block is complex.</li> <li>Does not support the 'symmetric' option.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
ifft2	<ul style="list-style-type: none"> <li>Length of input matrix dimensions must each be a power of 2.</li> <li>Does not support the 'symmetric' option.</li> </ul>
ifftn	<ul style="list-style-type: none"> <li>Length of input matrix dimensions must each be a power of 2.</li> <li>Does not support the 'symmetric' option.</li> </ul>
ifftshift	—
svd	Uses a different SVD implementation than MATLAB. Because the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
zp2tf	—

## Signal Processing Toolbox

C and C++ code generation for the following functions requires the Signal Processing Toolbox software. These functions do not support variable-size inputs, you must define the size and type of the function inputs. For more information, see “Specifying Inputs in Code Generation from MATLAB”.

---

**Note:** Many Signal Processing Toolbox functions require constant inputs in generated code. To specify a constant input for `codegen`, use `coder.Constant`.

---

Function	Remarks/Limitations
barthannwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bartlett	Window length must be a constant. Expressions or variables are allowed if their values do not change.
besselap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
bitrevorder	—
blackman	Window length must be a constant. Expressions or variables are allowed if their values do not change.
blackmanharris	Window length must be a constant. Expressions or variables are allowed if their values do not change.
bohmanwin	Window length must be a constant. Expressions or variables are allowed if their values do not change.
buttap	Filter order must be a constant. Expressions or variables are allowed if their values do not change.
butter	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
buttord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cfirpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
db2pow	—

Function	Remarks/Limitations
dct	C and C++ code generation for <code>dct</code> requires DSP System Toolbox software.  Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
downsample	—
dpss	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellip	Inputs must be constant. Expressions or variables are allowed if their values do not change.
ellipap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
ellipord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
filtfilt	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.
findpeaks	—
fir1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fir2	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
fircls1	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firls	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.



Function	Remarks/Limitations
<code>flattopwin</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>freqz</code>	<p>When called with no output arguments, and without a semicolon at the end, <code>freqz</code> returns the complex frequency response of the input filter, evaluated at 512 points.</p> <p>If the semicolon is added, the function produces a plot of the magnitude and phase response of the filter.</p> <p>See “<code>freqz</code> With No Output Arguments”.</p>
<code>gausswin</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>hamming</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>hann</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>idct</code>	<p>C and C++ code generation for <code>idct</code> requires DSP System Toolbox software.</p> <p>Length of transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.</p>
<code>intfilt</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>kaiser</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>kaiserord</code>	—
<code>levinson</code>	<p>C and C++ code generation for <code>levinson</code> requires DSP System Toolbox software.</p> <p>If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.</p>
<code>maxflat</code>	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
nuttallwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
parzenwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
pow2db	—
rcosdesign	All inputs must be constant. Expressions or variables are allowed if their values do not change.
rectwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
resample	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
sgolay	All inputs must be constant. Expressions or variables are allowed if their values do not change.
sosfilt	—
taylorwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
triang	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tukeywin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
upfirdn	C and C++ code generation for <code>upfirdn</code> requires DSP System Toolbox software.  Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change.  Variable-size inputs are not supported.
upsample	Either declare input <code>n</code> as constant, or use the <code>assert</code> function in the calling function to set upper bounds for <code>n</code> . For example,  <code>assert(n&lt;10)</code>
xcorr	—

Function	Remarks/Limitations
yulewalk	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

## Special Values in MATLAB

Function	Remarks and Limitations
eps	<ul style="list-style-type: none"> <li>Supported for scalar fixed-point signals only.</li> <li>Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.</li> </ul>
inf	<ul style="list-style-type: none"> <li>Dimensions must be real, nonnegative, integers.</li> </ul>
intmax	—
intmin	—
NaN or nan	<ul style="list-style-type: none"> <li>Dimensions must be real, nonnegative, integers.</li> </ul>
pi	—
realmax	—
realmin	—

## Specialized Math in MATLAB

Function	Remarks and Limitations
beta	—
betainc	Always returns a complex result.
betaincinv	Always returns a complex result.
betaln	—
ellipke	—
erf	—
erfc	—
erfcinv	—
erfcx	—

Function	Remarks and Limitations
erfinv	—
expint	—
gamma	—
gammainc	Output is always complex.
gammaincinv	Output is always complex.
gammaln	—
psi	—

### Statistics in MATLAB

Function	Remarks and Limitations
corrcoef	<ul style="list-style-type: none"> <li>Row-vector input is only supported when the first two inputs are vectors and nonscalar.</li> </ul>
mean	<ul style="list-style-type: none"> <li>Does not support the 'native' output class option for integer types.</li> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
median	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
mode	<ul style="list-style-type: none"> <li>Does not support third output argument <code>C</code> (cell array).</li> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>
std	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”
var	<ul style="list-style-type: none"> <li>If supplied, <code>dim</code> must be a constant.</li> <li>“Variable-Sizing Restrictions for Code Generation of Toolbox Functions”</li> </ul>

### Statistics Toolbox

C and C++ code generation for the following functions requires the Statistics Toolbox software.

Function	Remarks and Limitations
betacdf	—
betainv	—
betapdf	—
betarnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
betastat	—
binocdf	—
binoinv	—
binopdf	—
binornd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
binostat	—
cdf	—
chi2cdf	—
chi2inv	—
chi2pdf	—
chi2rnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
chi2stat	—
evcdf	—
evinv	—
evpdf	—

Function	Remarks and Limitations
evrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
evstat	—
expcdf	—
expinv	—
exppdf	—
exprnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
expstat	—
fcdf	—
finv	—
fpdf	—
frnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
fstat	—
gamcdf	—
gaminv	—
gampdf	—
gamrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>

Function	Remarks and Limitations
gamstat	—
geocdf	—
geoinv	—
geomean	—
geopdf	—
geornd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
geostat	—
gevcdf	—
gevinv	—
gevpdf	—
gevrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
gevstat	—
gpcdf	—
gpinv	—
gppdf	—
gprnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
gpstat	—
harmmean	—
hygecdf	—

Function	Remarks and Limitations
hygeinv	—
hygepdf	—
hygernd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
hygestat	—
icdf	—
iqr	—
kurtosis	—
logncdf	—
logninv	—
lognpdf	—
lognrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
lognstat	—
mad	Input <code>dim</code> cannot be empty.
mnpdf	Input <code>dim</code> cannot be empty.
moment	If <code>order</code> is nonintegral and <code>X</code> is real, use <code>moment(complex(X), order)</code> .
nancov	If the input is variable-size and is [ ] at run time, returns [ ] not NaN.
nanmax	—
nanmean	—
nanmedian	—
nanmin	—
nanstd	—



Function	Remarks and Limitations
nansum	—
nanvar	—
nbincdf	—
nbininv	—
nbinpdf	—
nbinrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
nbinstat	—
ncfcdf	—
ncfinv	—
ncfpdf	—
ncfrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
ncfstat	—
nctcdf	—
nctinv	—
nctpdf	—
nctrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
nctstat	—
ncx2cdf	—

Function	Remarks and Limitations
ncx2rnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
ncx2stat	—
normcdf	—
norminv	—
normpdf	—
normrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
normstat	—
pdf	—
poisscdf	—
poissinv	—
poisspdf	—
poissrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
poisstat	—

Function	Remarks and Limitations
prctile	<ul style="list-style-type: none"> <li>• “Automatic dimension restriction”</li> <li>• If the output Y is a vector, the orientation of Y differs from MATLAB when all of the following are true:                             <ul style="list-style-type: none"> <li>• You do not supply the <code>dim</code> input.</li> <li>• X is a variable-size array.</li> <li>• X is not a variable-length vector.</li> <li>• X is a vector at run time.</li> <li>• The orientation of the vector X does not match the orientation of the vector <code>p</code>.</li> </ul> </li> </ul> <p>In this case, the output Y matches the orientation of X not the orientation of <code>p</code>.</p>
quantile	—
randg	—
random	—
raylcdf	—
raylinv	—
raylpdf	—
raylrnd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
raylstat	—
skewness	—
tcdf	—
tinv	—
tpdf	—

Function	Remarks and Limitations
trnd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
tstat	—
unidcdf	—
unidinv	—
unidpdf	—
unidrnd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
unidstat	—
unifcdf	—
unifinv	—
unifpdf	—
unifrnd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>
unifstat	—
wblcdf	—
wblinv	—
wblpdf	—
wblrnd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> <li>• The output is nonscalar.</li> <li>• An input parameter is invalid for the distribution.</li> </ul>

Function	Remarks and Limitations
wblstat	—
zscore	—

## String Functions in MATLAB

Function	Remarks and Limitations
bin2dec	<ul style="list-style-type: none"> <li>Does not match MATLAB when the input is empty.</li> </ul>
bitmax	—
blanks	—
char	—
deblank	<ul style="list-style-type: none"> <li>Supports only inputs from the <code>char</code> class.</li> <li>Input values must be in the range 0-127.</li> </ul>
dec2bin	<ul style="list-style-type: none"> <li>If input <code>d</code> is <code>double</code>, <code>d</code> must be less than <math>2^{52}</math>.</li> <li>If input <code>d</code> is <code>single</code>, <code>d</code> must be less than <math>2^{23}</math>.</li> <li>Unless you specify input <code>n</code> to be constant and <code>n</code> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <code>n</code> must be at least 52 for <code>double</code>, 23 for <code>single</code>, 16 for <code>char</code>, 32 for <code>int32</code>, 16 for <code>int16</code>, and so on.</li> </ul>
dec2hex	<ul style="list-style-type: none"> <li>If input <code>d</code> is <code>double</code>, <code>d</code> must be less than <math>2^{52}</math>.</li> <li>If input <code>d</code> is <code>single</code>, <code>d</code> must be less than <math>2^{23}</math>.</li> <li>Unless you specify input <code>n</code> to be constant and <code>n</code> is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, <code>n</code> must be at least 13 for <code>double</code>, 6 for <code>single</code>, 4 for <code>char</code>, 8 for <code>int32</code>, 4 for <code>int16</code>, and so on.</li> </ul>
hex2dec	—
hex2num	<ul style="list-style-type: none"> <li>For <code>n = hex2num(S)</code>, <code>size(S,2) &lt;= length(num2hex(0))</code></li> </ul>
ischar	—
isletter	<ul style="list-style-type: none"> <li>Input values from the <code>char</code> class must be in the range 0-127</li> </ul>
isspace	<ul style="list-style-type: none"> <li>Input values from the <code>char</code> class must be in the range 0–127.</li> </ul>

Function	Remarks and Limitations
<code>isstrprop</code>	<ul style="list-style-type: none"> <li>• Supports only inputs from <code>char</code> and <code>integer</code> classes.</li> <li>• Input values must be in the range 0-127.</li> </ul>
<code>lower</code>	<ul style="list-style-type: none"> <li>• Supports only <code>char</code> inputs.</li> <li>• Input values must be in the range 0-127.</li> </ul>
<code>num2hex</code>	—
<code>str2double</code>	<ul style="list-style-type: none"> <li>• Does not support cell arrays.</li> <li>• Always returns a complex result.</li> </ul>
<code>strcmp</code>	—
<code>strcmpi</code>	Input values from the <code>char</code> class must be in the range 0-127.
<code>strfind</code>	<ul style="list-style-type: none"> <li>• Does not support cell arrays.</li> <li>• If <code>pattern</code> does not exist in <code>str</code>, returns <code>zeros(1,0)</code> not <code>[]</code>. To check for an empty return, use <code>isempty</code>.</li> <li>• Inputs must be character row vectors.</li> </ul>
<code>strjust</code>	—
<code>strncmp</code>	—
<code>strncmpi</code>	• Input values from the <code>char</code> class must be in the range 0-127.
<code>strrep</code>	<ul style="list-style-type: none"> <li>• Does not support cell arrays.</li> <li>• If <code>oldSubstr</code> does not exist in <code>origStr</code>, returns <code>blanks(0)</code>. To check for an empty return, use <code>isempty</code>.</li> <li>• Inputs must be character row vectors.</li> </ul>
<code>strtok</code>	—
<code>strtrim</code>	<ul style="list-style-type: none"> <li>• Supports only inputs from the <code>char</code> class.</li> <li>• Input values must be in the range 0-127.</li> </ul>
<code>upper</code>	<ul style="list-style-type: none"> <li>• Supports only <code>char</code> inputs.</li> <li>• Input values must be in the range 0-127.</li> </ul>

## Structures in MATLAB

Function	Remarks and Limitations
isfield	<ul style="list-style-type: none"> <li>Does not support cell input for second argument</li> </ul>
isstruct	—
struct	—

## Trigonometry in MATLAB

Function	Remarks and Limitations
acos	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
acosd	—
acosh	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
acot	—
acotd	—
acoth	—
acsc	—
acscd	—
acsch	—
asec	—
asecd	—
asech	—
asin	<ul style="list-style-type: none"> <li>Generates an error during simulation and returns NaN in generated code when the input value <math>x</math> is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code>.</li> </ul>
asind	—

Function	Remarks and Limitations
asinh	—
atan	—
atan2	—
atan2d	—
atand	—
atanh	Generates an error during simulation and returns NaN in generated code when the input value $x$ is real, but the output should be complex. To get the complex result, make the input value complex by passing in <code>complex(x)</code> .
cos	—
cosd	—
cosh	—
cot	—
cotd	<ul style="list-style-type: none"> <li>• In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>• In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
coth	—
csc	—
cscd	<ul style="list-style-type: none"> <li>• In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>• In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
csch	—
hypot	—
sec	—
secd	<ul style="list-style-type: none"> <li>• In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>.</li> <li>• In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.</li> </ul>
sech	—
sin	—
sind	—
sinh	—
tan	—



Function	Remarks and Limitations
tand	<ul style="list-style-type: none"> <li>• In some cases, returns -Inf when MATLAB returns Inf.</li> <li>• In some cases, returns Inf when MATLAB returns -Inf.</li> </ul>
tanh	—



# Defining MATLAB Variables for C/C++ Code Generation

---

- “Variables Definition for Code Generation” on page 5-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 5-3
- “Eliminate Redundant Copies of Variables in Generated Code” on page 5-7
- “Reassignment of Variable Properties” on page 5-9
- “Define and Initialize Persistent Variables” on page 5-10
- “Reuse the Same Variable with Different Properties” on page 5-11
- “Avoid Overflows in for-Loops” on page 5-15
- “Supported Variable Types” on page 5-17

### Variables Definition for Code Generation

In the MATLAB language, variables can change their properties dynamically at run time so you can use the same variable to hold a value of any class, size, or complexity. For example, the following code works in MATLAB:

```
function x = foo(c) %#codegen
if(c>0)
    x = 0;
else
    x = [1 2 3];
end
disp(x);
end
```

However, statically-typed languages like C must be able to determine variable properties at compile time. Therefore, for C/C++ code generation, you must explicitly define the class, size, and complexity of variables in MATLAB source code before using them. For example, rewrite the above source code with a definition for *x*:

```
function x = foo(c) %#codegen
x = zeros(1,3);
if(c>0)
    x = 0;
else
    x = [1 2 3];
end
disp(x);
end
```

For more information, see “Best Practices for Defining Variables for C/C++ Code Generation” on page 5-3.

## Best Practices for Defining Variables for C/C++ Code Generation

### In this section...

“Define Variables By Assignment Before Using Them” on page 5-3

“Use Caution When Reassigning Variables” on page 5-5

“Use Type Cast Operators in Variable Definitions” on page 5-5

“Define Matrices Before Assigning Indexed Variables” on page 5-6

### Define Variables By Assignment Before Using Them

For C/C++ code generation, you should explicitly and unambiguously define the class, size, and complexity of variables before using them in operations or returning them as outputs. Define variables by assignment, but note that the assignment copies not only the value, but also the size, class, and complexity represented by that value to the new variable. For example:

Assignment:	Defines:
<code>a = 14.7;</code>	a as a real double scalar.
<code>b = a;</code>	b with properties of a (real double scalar).
<code>c = zeros(5,2);</code>	c as a real 5-by-2 array of doubles.
<code>d = [1 2 3 4 5; 6 7 8 9 0];</code>	d as a real 5-by-2 array of doubles.
<code>y = int16(3);</code>	y as a real 16-bit integer scalar.

Define properties this way so that the variable is defined on the required execution paths during C/C++ code generation (see Defining a Variable for Multiple Execution Paths).

The data that you assign to a variable can be a scalar, matrix, or structure. If your variable is a structure, define the properties of each field explicitly (see Defining Fields in a Structure).

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in “Eliminate Redundant Copies of Variables in Generated Code” on page 5-7.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see “Define and Initialize Persistent Variables” on page 5-10.

### Defining a Variable for Multiple Execution Paths

Consider the following MATLAB code:

```
...
if c > 0
    x = 11;
end
% Later in your code ...
if c > 0
    use(x);
end
...
```

Here,  $x$  is assigned only if  $c > 0$  and used only when  $c > 0$ . This code works in MATLAB, but generates a compilation error during code generation because it detects that  $x$  is undefined on some execution paths (when  $c \leq 0$ ),

To make this code suitable for code generation, define  $x$  before using it:

```
x = 0;
...
if c > 0
    x = 11;
end
% Later in your code ...
if c > 0
    use(x);
end
...
```

### Defining Fields in a Structure

Consider the following MATLAB code:

```
...
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
end
```

```

    s.b = 12;
end
% Try to use s
use(s);
...

```

Here, the first part of the `if` statement uses only the field `a`, and the `else` clause uses fields `a` and `b`. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see “Structure Definition for Code Generation”.

To make this code suitable for C/C++ code generation, define all fields of `s` before using it.

```

...
% Define all fields in structure s
s = struct('a',0, 'b', 0);
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    s.b = 12;
end
% Use s
use(s);
...

```

## Use Caution When Reassigning Variables

In general, you should adhere to the "one variable/one type" rule for C/C++ code generation; that is, each variable must have a specific class, size and complexity. Generally, if you reassign variable properties after the initial assignment, you get a compilation error during code generation, but there are exceptions, as described in “Reassignment of Variable Properties” on page 5-9.

## Use Type Cast Operators in Variable Definitions

By default, constants are of type `double`. To define variables of other types, you can use type cast operators in variable definitions. For example, the following code defines variable `y` as an integer:

```

...

```

```
x = 15; % x is of type double by default.  
y = uint8(x); % y has the value of x, but cast to uint8.  
...
```

### Define Matrices Before Assigning Indexed Variables

When generating C/C++ code from MATLAB, you cannot grow a variable by writing into an element beyond its current size. Such indexing operations produce run-time errors. You must define the matrix first before assigning values to its elements.

For example, the following initial assignment is not allowed for code generation:

```
g(3,2) = 14.6; % Not allowed for creating g  
           % OK for assigning value once created
```

For more information about indexing matrices, see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-30.



## Eliminate Redundant Copies of Variables in Generated Code

### In this section...

“When Redundant Copies Occur” on page 5-7

“How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 5-7

“Defining Uninitialized Variables” on page 5-8

### When Redundant Copies Occur

During C/C++ code generation, MATLAB checks for statements that attempt to access uninitialized memory. If it detects execution paths where a variable is used but is potentially not defined, it generates a compile-time error. To prevent these errors, define variables by assignment before using them in operations or returning them as function outputs.

Note, however, that variable assignments not only copy the properties of the assigned data to the new variable, but also initialize the new variable to the assigned value. This forced initialization sometimes results in redundant copies in C/C++ code. To eliminate redundant copies, define uninitialized variables by using the `coder.nullcopy` function, as described in “How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 5-7.

### How to Eliminate Redundant Copies by Defining Uninitialized Variables

- 1 Define the variable with `coder.nullcopy`.
- 2 Initialize the variable before reading it.

When the uninitialized variable is an array, you must initialize all of its elements before passing the array as an input to a function or operator — even if the function or operator does not read from the uninitialized portion of the array.

### What happens if you access uninitialized data?

Uninitialized memory contains arbitrary values. Therefore, accessing uninitialized data may lead to segmentation violations or nondeterministic program behavior (different runs of the same program may yield inconsistent results).

### Defining Uninitialized Variables

In the following code, the assignment statement `X = zeros(1,N)` not only defines `X` to be a 1-by-5 vector of real doubles, but also initializes each element of `X` to zero.

```
function X = fcn %#codegen

N = 5;
X = zeros(1,N);
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    else
        X(i) = 0;
    end
end
```

This forced initialization creates an extra copy in the generated code. To eliminate this overhead, use `coder.nullcopy` in the definition of `X`:

```
function X = fcn2 %#codegen

N = 5;
X = coder.nullcopy(zeros(1,N));
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    else
        X(i) = 0;
    end
end
```

## Reassignment of Variable Properties

For C/C++ code generation, there are certain variables that you can reassign after the initial assignment with a value of different class, size, or complexity:

### Dynamically sized variables

A variable can hold values that have the same class and complexity but different sizes. If the size of the initial assignment is not constant, the variable is dynamically sized in generated code. For more information, see “Variable-Size Data”.

### Variables reused in the code for different purposes

You can reassign the type (class, size, and complexity) of a variable after the initial assignment if each occurrence of the variable can have only one type. In this case, the variable is renamed in the generated code to create multiple independent variables. For more information, see “Reuse the Same Variable with Different Properties” on page 5-11.

### Define and Initialize Persistent Variables

Persistent variables are local to the function in which they are defined, but they retain their values in memory between calls to the function. To define persistent variables for C/C++ code generation, use the `persistent` statement, as in this example:

```
persistent PROD_X;
```

The definition should appear at the top of the function body, after the header and comments, but before the first use of the variable. During code generation, the value of the persistent variable is initialized to an empty matrix by default. You can assign your own value after the definition by using the `isempty` statement, as in this example:

```
function findProduct(inputvalue) %#codegen
persistent PROD_X

if isempty(PROD_X)
    PROD_X = 1;
end
PROD_X = PROD_X * inputvalue;
end
```

## Reuse the Same Variable with Different Properties

### In this section...

“When You Can Reuse the Same Variable with Different Properties” on page 5-11

“When You Cannot Reuse Variables” on page 5-11

“Limitations of Variable Reuse” on page 5-14

### When You Can Reuse the Same Variable with Different Properties

You can reuse (reassign) an input, output, or local variable with different class, size, or complexity if MATLAB can unambiguously determine the properties of each occurrence of this variable during C/C++ code generation. If so, MATLAB creates separate uniquely named local variables in the generated code. You can view these renamed variables in the code generation report (see “Viewing Variables in Your MATLAB Code”).

A common example of variable reuse is in `if-elseif-else` or `switch-case` statements. For example, the following function `example1` first uses the variable `t` in an `if` statement, where it holds a scalar double, then reuses `t` outside the `if` statement to hold a vector of doubles.

```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
```

To compile this example and see how MATLAB renames the reused variable `t`, see Variable Reuse in an `if` Statement.

### When You Cannot Reuse Variables

You cannot reuse (reassign) variables if it is not possible to determine the class, size, and complexity of an occurrence of a variable unambiguously during code generation. In this case, variables cannot be renamed and a compilation error occurs.

For example, the following `example2` function assigns a fixed-point value to `x` in the `if` statement and reuses `x` to store a matrix of doubles in the `else` clause. It then uses `x`

after the `if-else` statement. This function generates a compilation error because after the `if-else` statement, variable `x` can have different properties depending on which `if-else` clause executes.

```
function y = example2(use_fixpoint, data) %#codegen
    if use_fixpoint
        % x is fixed-point
        x = fi(data, 1, 12, 3);
    else
        % x is a matrix of doubles
        x = data;
    end
    % When x is reused here, it is not possible to determine its
    % class, size, and complexity
    t = sum(sum(x));
    y = t > 0;
end
```

### Variable Reuse in an if Statement

To see how MATLAB renames a reused variable `t`:

- 1 Create a MATLAB file `example1.m` containing the following code.

```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
end
```

- 2 Compile `example1`.

For example, to generate a MEX function, enter:

```
codegen -o example1x -report example1.m -args {ones(5,5)}
```

---

**Note:** `codegen` requires a MATLAB Coder license.

---

When the compilation is complete, `codegen` generates a MEX function, `example1x` in the current folder, and provides a link to the code generation report.

- 3 Open the code generation report.
- 4 In the MATLAB code pane of the code generation report, place your pointer over the variable `t` inside the `if` statement.

The code generation report highlights both instances of `t` in the `if` statement because they share the same class, size, and complexity. It displays the data type information for `t` at this point in the code. Here, `t` is a scalar double.

```
% First time t is used to hold a scalar double value.
t = mean(mean(u)) / numel(u);
u = u - t;
```

Information for the selected variable:	
Size	1 x 1
Complex	No
Class	double

- 5 In the MATLAB code pane of the report, place your pointer over the variable `t` outside the for-loop.

This time, the report highlights both instances of `t` outside the `if` statement. The report indicates that `t` might hold up to 25 doubles. The size of `t` is `:25`, that is, a column vector containing a maximum of 25 doubles.

```
t = find(u);
y = sum(u(t(2:end-1)));
```

Information for the selected variable:	
Size	:25
Complex	No
Class	double

- 6 Click the **Variables** tab to view the list of variables used in `example1`.

The report displays a list of the variables in `example1`. There are two uniquely named local variables `t>1` and `t>2`.

- 7 In the list of variables, place your pointer over `t>1`.

The code generation report highlights both instances of `t` in the `if` statement.

- 8 In the list of variables, place your pointer over `t>2`

The code generation report highlights both instances of `t` outside the `if` statement.

### Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

- Persistent variables.
- Global variables.
- Variables passed to C code using `coder.ref`, `coder.rref`, `coder.wref`.
- Variables whose size is set using `coder.ysize`.
- Variables whose names are controlled using `coder.cstructname`.
- The index variable of a `for`-loop when it is used inside the loop body.
- The block outputs of a MATLAB Function block in a Simulink model.
- Chart-owned variables of a MATLAB function in a Stateflow<sup>®</sup> chart.



## Avoid Overflows in for-Loops

When memory integrity checks are enabled, if the code generation software detects that a loop variable might overflow on the last iteration of the `for`-loop, it reports an error.

To avoid this error, use the workarounds provided in the following table.

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"> <li>The loop counter increments by 1</li> <li>The end value equals the maximum value of the integer type</li> <li>The loop is not covering the full range of the integer type</li> </ul>	Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace: <pre>N=intmax('int16') for k=N-10:N</pre> with: <pre>for k=1:10</pre>
<ul style="list-style-type: none"> <li>The loop counter decrements by 1</li> <li>The end value equals the minimum value of the integer type</li> <li>The loop is not covering the full range of the integer type</li> </ul>	Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace: <pre>N=intmin('int32') for k=N+10:-1:N</pre> with: <pre>for k=10:-1:1</pre>
<ul style="list-style-type: none"> <li>The loop counter increments or decrements by 1</li> <li>The start value equals the minimum or maximum value of the integer type</li> <li>The end value equals the maximum or minimum value of the integer type</li> </ul> The loop covers the full range of the integer type.	Rewrite the loop casting the type of the loop counter start, step, and end values to a bigger integer or to double. For example, rewrite: <pre>M= intmin('int16'); N= intmax('int16'); for k=M:N % Loop body end to  M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N) % Loop body</pre>

Loop conditions causing the error	Workaround
<ul style="list-style-type: none"><li>• The loop counter increments or decrements by a value not equal to 1</li><li>• On last loop iteration, the loop variable value is not equal to the end value</li></ul>	end
<b>Note:</b> The software error checking is conservative. It may incorrectly report a loop as being potentially infinite.	Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.

## Supported Variable Types

You can use the following data types for C/C++ code generation from MATLAB:

Type	Description
char	Character array (string)
complex	Complex data. Cast function takes real and imaginary components
double	Double-precision floating point
int8, int16, int32, int64	Signed integer
logical	Boolean true or false
single	Single-precision floating point
struct	Structure
uint8, uint16, uint32, uint64	Unsigned integer
Fixed-point	See “Fixed-Point Data Types”.



# Defining Data for Code Generation

---

- “Data Definition for Code Generation” on page 6-2
- “Code Generation for Complex Data” on page 6-4
- “Code Generation for Characters” on page 6-6
- “Array Size Restrictions for Code Generation” on page 6-7

## Data Definition for Code Generation

To generate efficient standalone code, you must define the following types and classes of data differently than you normally would when running your code in the MATLAB environment:

Data	What's Different	More Information
Arrays	Maximum number of elements is restricted	“Array Size Restrictions for Code Generation” on page 6-7
Complex numbers	<ul style="list-style-type: none"> <li>Complexity of variables must be set at time of assignment and before first use</li> <li>Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero</li> </ul> <hr/> <p><b>Note:</b> Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic</p>	“Code Generation for Complex Data” on page 6-4
Characters	Restricted to 8 bits of precision	“Code Generation for Characters” on page 6-6
Enumerated data	<ul style="list-style-type: none"> <li>Supports integer-based enumerated types only</li> <li>Restricted use in <code>switch</code> statements and <code>for</code>-loops</li> </ul>	“Enumerated Data”
Function handles	<ul style="list-style-type: none"> <li>Same bound variable cannot reference</li> </ul>	“Function Handles”

Data	What's Different	More Information
	<p data-bbox="647 300 862 357">different function handles</p> <ul data-bbox="609 374 911 604" style="list-style-type: none"><li data-bbox="609 374 911 496">• Cannot pass function handles to or from primary or extrinsic functions</li><li data-bbox="609 513 911 604">• Cannot view function handles from the debugger</li></ul>	

## Code Generation for Complex Data

### In this section...

“Restrictions When Defining Complex Variables” on page 6-4

“Expressions With Complex Operands Yield Complex Results” on page 6-4

### Restrictions When Defining Complex Variables

For code generation, you must set the complexity of variables at the time of assignment, either by assigning a complex constant or using the `complex` function, as in these examples:

```
x = 5 + 6i; % x is a complex number by assignment.
y = 7 + 8j; % y is a complex number by assignment.
x = complex(5,6); % x is the complex number 5 + 6i.
```

Once you set the type and size of a variable, you cannot cast it to another type or size. In the following example, the variable `x` is defined as complex and stays complex:

```
x = 1 + 2i; % Defines x as a complex variable.
y = int16(x); % Real and imaginary parts of y are int16.
x = 3; % x now has the value 3 + 0i.
```

Mismatches can also occur when you assign a real operand the complex result of an operation:

```
z = 3; % Sets type of z to double (real)
z = 3 + 2i; % ERROR: cannot recast z to complex
```

As a workaround, set the complexity of the operand to match the result of the operation:

```
m = complex(3); % Sets m to complex variable of value 3 + 0i
m = 5 + 6.7i; % Assigns a complex result to a complex number
```

### Expressions With Complex Operands Yield Complex Results

In general, expressions that contain one or more complex operands produce a complex result in generated code, even if the value of the result is zero. Consider the following example:

```
x = 2 + 3i;
```



```
y = 2 - 3i;
z = x + y; % z is 4 + 0i.
```

In MATLAB, this code generates the real result  $z = 4$ . During code generation, the types for  $x$  and  $y$  are known, but their values are not. Because either or both operands in this expression are complex,  $z$  is defined as a complex variable requiring storage for both a real and an imaginary part.  $z$  equals the complex result  $4 + 0i$  in generated code, not  $4$  as in MATLAB code.

Exceptions to this behavior are:

- Values returned by MEX functions are real when the imaginary part of the value is zero.

```
function y = foo()
    y = 1 + 0i; % y is complex with imaginary part equal to zero
end
```

The MEX function `foo_mex` returns the real value 1.

```
z = foo_mex
```

- Complex arguments to extrinsic functions are real when the imaginary part of the argument is zero.

```
function y = foo()
    coder.extrinsic('sqrt')
    x = 1 + 0i; % x is complex
    y = sqrt(x); % x is real, y is real
end
```

- Functions that take complex arguments but produce real results return real values.

```
y = real(x); % y is the real part of the complex number x.
y = imag(x); % y is the real-valued imaginary part of x.
y = isreal(x); % y is false (0) for a complex number x.
```

- Functions that take real arguments but produce complex results return complex values.

```
z = complex(x,y); % z is a complex number for a real x and y.
```

## Code Generation for Characters

The complete set of Unicode<sup>®</sup> characters is not supported for code generation. Characters are restricted to 8 bits of precision in generated code. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with characters if you intend to generate code from your MATLAB algorithm.

## Array Size Restrictions for Code Generation

For code generation, the maximum number of elements of an array is constrained by the code generation software and the target hardware.

For fixed-size arrays and variable-size arrays that use static memory allocation, the maximum number of elements is the smaller of:

- `intmax('int32')`.
- The largest integer that fits in the C `int` data type on the target hardware.

For variable-size arrays that use dynamic memory allocation, the maximum number of elements is the smaller of:

- `intmax('int32')`.
- The largest power of 2 that fits in the C `int` data type on the target hardware.

These restrictions apply even on a 64-bit platform.

For a fixed-size array, if the number of elements exceeds the maximum, the code generation software reports an error at compile time. For a variable-size array, if the number of elements exceeds the maximum during execution of the generated MEX in MATLAB, the MEX code reports an error. Generated standalone code cannot report array size violations.

### See Also

- “Variable-Size Data”
- `coder.HardwareImplementation`



# Code Generation for Variable-Size Data

---

- “What Is Variable-Size Data?” on page 7-2
- “Variable-Size Data Definition for Code Generation” on page 7-3
- “Bounded Versus Unbounded Variable-Size Data” on page 7-4
- “Control Memory Allocation of Variable-Size Data” on page 7-5
- “Specify Variable-Size Data Without Dynamic Memory Allocation” on page 7-6
- “Variable-Size Data in Code Generation Reports” on page 7-9
- “Define Variable-Size Data for Code Generation” on page 7-11
- “C Code Interface for Arrays” on page 7-17
- “Diagnose and Fix Variable-Size Data Errors” on page 7-21
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 7-25
- “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-33

## What Is Variable-Size Data?

Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.

For example, in the following MATLAB function `nway`, `B` is a variable-size array; its length is not known at compile time.

```
function B = nway(A,n)
% Compute average of every N elements of A and put them in B.
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
        B(i) = mean(A(k + (0:n-1)));
        k = k + n;
    end
else
    error('n <= 0 or does not divide number of elements evenly');
end
```

## Variable-Size Data Definition for Code Generation

In the MATLAB language, data can vary in size. By contrast, the semantics of generated code constrains the class, complexity, and shape of every expression, variable, and structure field. Therefore, for code generation, you must use each variable consistently. Each variable must:

- Be either complex or real (determined at first assignment)
- Have a consistent shape

For fixed-size data, the shape is the same as the size returned in MATLAB. For example, if `size(A) == [4 5]`, the shape of variable `A` is 4 x 5. For variable-size data, the shape can be abstract. That is, one or more dimensions can be unknown (such as `4x?` or `?x?`).

By default, the compiler detects code logic that attempts to change these fixed attributes after initial assignments, and flags these occurrences as errors during code generation. However, you can override this behavior by defining variables or structure fields as variable-size data.

For more information, see “Bounded Versus Unbounded Variable-Size Data” on page 7-4

### Bounded Versus Unbounded Variable-Size Data

You can generate code for bounded and unbounded variable-size data. *Bounded variable-size data* has fixed upper bounds; this data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds; this data *must* be allocated on the heap. If you use unbounded data, you must use dynamic memory allocation so that the compiler:

- Does not check for upper bounds
- Allocates memory on the heap instead of the stack

You can control the memory allocation of variable-size data. For more information, see “Control Memory Allocation of Variable-Size Data” on page 7-5.



## Control Memory Allocation of Variable-Size Data

Data whose size (in bytes) is greater than or equal to the dynamic memory allocation threshold is allocated on the heap. The default dynamic memory allocation threshold is 64 kilobytes. Data whose size is less than this threshold is allocated on the stack.

Dynamic memory allocation is an expensive operation; the performance cost might be too high for small data sets. If you use small variable-size data sets or data that does not change size at run time, disable dynamic memory allocation. See “Control Dynamic Memory Allocation”.

You can control memory allocation globally for your application by modifying the dynamic memory allocation threshold. See “Generate Code for a MATLAB Function That Expands a Vector in a Loop”. You can control memory allocation for individual variables by specifying upper bounds. See “Specifying Upper Bounds for Variable-Size Data” on page 7-6.

## Specify Variable-Size Data Without Dynamic Memory Allocation

### In this section...

“Fixing Upper Bounds Errors” on page 7-6

“Specifying Upper Bounds for Variable-Size Data” on page 7-6

### Fixing Upper Bounds Errors

If MATLAB cannot determine or compute the upper bound, you must specify an upper bound. See “Specifying Upper Bounds for Variable-Size Data” on page 7-6 and “Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 7-23

### Specifying Upper Bounds for Variable-Size Data

- “When to Specify Upper Bounds for Variable-Size Data” on page 7-6
- “Specifying Upper Bounds on the Command Line for Variable-Size Inputs” on page 7-6
- “Specifying Unknown Upper Bounds for Variable-Size Inputs” on page 7-7
- “Specifying Upper Bounds for Local Variable-Size Data” on page 7-7
- “Using a Matrix Constructor with Nonconstant Dimensions” on page 7-8

### When to Specify Upper Bounds for Variable-Size Data

When using static allocation on the stack during code generation, MATLAB must be able to determine upper bounds for variable-size data. Specify the upper bounds explicitly for variable-size data from external sources, such as inputs and outputs.

### Specifying Upper Bounds on the Command Line for Variable-Size Inputs

Use the `coder.typeof` construct with the `-args` option on the `codegen` command line (requires a MATLAB Coder license). For example:

```
codegen foo -args {coder.typeof(double(0),[3 100],1)}
```

This command specifies that the input to function `foo` is a matrix of real doubles with two variable dimensions. The upper bound for the first dimension is 3; the upper bound for the second dimension is 100. For a detailed explanation of this syntax, see `coder.typeof`.

### Specifying Unknown Upper Bounds for Variable-Size Inputs

If you use dynamic memory allocation, you can specify that you don't know the upper bounds of inputs. To specify an unknown upper bound, use the infinity constant `Inf` in place of a numeric value. For example:

```
codegen foo -args {coder.typeof(double(0), [1 Inf])}
```

In this example, the input to function `foo` is a vector of real doubles without an upper bound.

### Specifying Upper Bounds for Local Variable-Size Data

When using static allocation, MATLAB uses a sophisticated analysis to calculate the upper bounds of local data at compile time. However, when the analysis fails to detect an upper bound or calculates an upper bound that is not precise enough for your application, you need to specify upper bounds explicitly for local variables.

You do not need to specify upper bounds when using dynamic allocation on the heap. In this case, MATLAB assumes variable-size data is unbounded and does not attempt to determine upper bounds.

### Constraining the Value of a Variable That Specifies Dimensions of Variable-Size Data

Use the `assert` function with relational operators to constrain the value of variables that specify the dimensions of variable-size data. For example:

```
function y = dim_need_bound(n) %#codegen
assert (n <= 5);
L = ones(n,n);
M = zeros(n,n);
M = [L; M];
y = M;
```

This `assert` statement constrains input `n` to a maximum size of 5, defining `L` and `M` as variable-sized matrices with upper bounds of 5 for each dimension.

### Specifying the Upper Bounds for All Instances of a Local Variable

Use the `coder. varsizes` function to specify the upper bounds for all instances of a local variable in a function. For example:

```
function Y = example_bounds1(u) %#codegen
Y = [1 2 3 4 5];
```

```
coder.varsize('Y', [1 10]);  
if (u > 0)  
    Y = [Y Y+u];  
else  
    Y = [Y Y*u];  
end
```

The second argument of `coder.varsize` specifies the upper bound for each instance of the variable specified in the first argument. In this example, the argument `[1 10]` indicates that for every instance of `Y`:

- First dimension is fixed at size 1
- Second dimension can grow to an upper bound of 10

By default, `coder.varsize` assumes dimensions of 1 are fixed size. For more information, see the `coder.varsize` reference page.

### Using a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For example:

```
function y = var_by_assign(u) %#codegen  
if (u > 0)  
    y = ones(3,u);  
else  
    y = zeros(3,1);  
end
```

If you are not using dynamic memory allocation, you must also add an `assert` statement to provide upper bounds for the dimensions. For example:

```
function y = var_by_assign(u) %#codegen  
assert (u < 20);  
if (u > 0)  
    y = ones(3,u);  
else  
    y = zeros(3,1);  
end
```

## Variable-Size Data in Code Generation Reports

In this section...
“What Reports Tell You About Size” on page 7-9
“How Size Appears in Code Generation Reports” on page 7-10
“How to Generate a Code Generation Report” on page 7-10

### What Reports Tell You About Size

Code generation reports:

- Differentiate fixed-size from variable-size data
- Identify variable-size data with unknown upper bounds
- Identify variable-size data with fixed dimensions

If you define a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends \* to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions does not change during execution.

- Provide guidance on how to fix size mismatch and upper bounds errors.

## How Size Appears in Code Generation Reports

Variable	Type	Size
B	Output	1 x :?
A	Input	1 x :100
n	Input	1 x 1

:? means variable size, unknown upper bound

No colon prefix (:) means fixed size

:100 means variable size, upper bound = 100

Variable	Type	Size
y	Output	1 x 10 *

\* means that you declared y as variable size, but subsequently fixed its dimensions

## How to Generate a Code Generation Report

Add the `-report` option to your `codegen` command.

## Define Variable-Size Data for Code Generation

### In this section...

“When to Define Variable-Size Data Explicitly” on page 7-11

“Using a Matrix Constructor with Nonconstant Dimensions” on page 7-11

“Inferring Variable Size from Multiple Assignments” on page 7-12

“Defining Variable-Size Data Explicitly Using `coder.varsizes`” on page 7-13

### When to Define Variable-Size Data Explicitly

For code generation, you must assign variables to have a specific class, size, and complexity before using them in operations or returning them as outputs. Generally, you cannot reassign variable properties after the initial assignment. Therefore, attempts to grow a variable or structure field after assigning it a fixed size might cause a compilation error. In these cases, you must explicitly define the data as variable sized using one of these methods:

Method	See
Assign the data from a variable-size matrix constructor such as <ul style="list-style-type: none"> <li>• <code>ones</code></li> <li>• <code>zeros</code></li> <li>• <code>repmat</code></li> </ul>	“Using a Matrix Constructor with Nonconstant Dimensions” on page 7-11
Assign multiple, constant sizes to the same variable before using (reading) the variable.	“Inferring Variable Size from Multiple Assignments” on page 7-12
Define all instances of a variable to be variable sized	“Defining Variable-Size Data Explicitly Using <code>coder.varsizes</code> ” on page 7-13

### Using a Matrix Constructor with Nonconstant Dimensions

You can define a variable-size matrix by using a constructor with nonconstant dimensions. For example:

```
function y = var_by_assign(u) %#codegen
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

If you are not using dynamic memory allocation, you must also add an `assert` statement to provide upper bounds for the dimensions. For example:

```
function y = var_by_assign(u) %#codegen
assert (u < 20);
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

### Inferring Variable Size from Multiple Assignments

You can define variable-size data by assigning multiple, constant sizes to the same variable before you use (read) the variable in your code. When MATLAB uses static allocation on the stack for code generation, it infers the upper bounds from the largest size specified for each dimension. When you assign the same size to a given dimension across all assignments, MATLAB assumes that the dimension is fixed at that size. The assignments can specify different shapes as well as sizes.

When dynamic memory allocation is used, MATLAB does not check for upper bounds; it assumes variable-size data is unbounded.

### Inferring Upper Bounds from Multiple Definitions with Different Shapes

```
function y = var_by_multiassign(u) %#codegen
if (u > 0)
    y = ones(3,4,5);
else
    y = zeros(3,1);
end
```

When static allocation is used, this function infers that `y` is a matrix with three dimensions, where:

- First dimension is fixed at size 3



- Second dimension is variable with an upper bound of 4
- Third dimension is variable with an upper bound of 5

The code generation report represents the size of matrix `y` like this:

Variable	Type	Size
<code>y</code>	Output	<code>3 x :4 x :5</code>

When dynamic allocation is used, the function analyzes the dimensions of `y` differently:

- First dimension is fixed at size 3
- Second and third dimensions are unbounded

In this case, the code generation report represents the size of matrix `y` like this:

Variable	Type	Size
<code>y</code>	Output	<code>3 x :? x :?</code>

## Defining Variable-Size Data Explicitly Using `coder.varsize`

Use the function `coder. varsize` to define one or more variables or structure fields as variable-size data. Optionally, you can also specify which dimensions vary along with their upper bounds (see “Specifying Which Dimensions Vary” on page 7-14). For example:

- Define `B` as a variable-size 2-by-2 matrix, where each dimension has an upper bound of 64:

```
coder. varsize('B', [64 64]);
```

- Define `B` as a variable-size matrix:

```
coder. varsize('B');
```

When you supply only the first argument, `coder. varsize` assumes all dimensions of `B` can vary and that the upper bound is `size(B)`.

For more information, see the `coder. varsize` reference page.

### Specifying Which Dimensions Vary

You can use the function `coder. varsize` to specify which dimensions vary. For example, the following statement defines `B` as a row vector whose first dimension is fixed at 2, but whose second dimension can grow to an upper bound of 16:

```
coder. varsize('B', [2, 16], [0 1])
```

The third argument specifies which dimensions vary. This argument must be a logical vector or a double vector containing only zeros and ones. Dimensions that correspond to zeros or `false` have fixed size; dimensions that correspond to ones or `true` vary in size. `coder. varsize` usually treats dimensions of size 1 as fixed (see “Defining Variable-Size Matrices with Singleton Dimensions” on page 7-14).

For more information about the syntax, see the `coder. varsize` reference page.

### Allowing a Variable to Grow After Defining Fixed Dimensions

Function `var_by_if` defines matrix `Y` with fixed 2-by-2 dimensions before first use (where the statement `Y = Y + u` reads from `Y`). However, `coder. varsize` defines `Y` as a variable-size matrix, allowing it to change size based on decision logic in the `else` clause:

```
function Y = var_by_if(u) %#codegen
if (u > 0)
    Y = zeros(2,2);
    coder. varsize('Y');
    if (u < 10)
        Y = Y + u;
    end
else
    Y = zeros(5,5);
end
```

Without `coder. varsize`, MATLAB infers `Y` to be a fixed-size, 2-by-2 matrix and generates a size mismatch error during code generation.

### Defining Variable-Size Matrices with Singleton Dimensions

A singleton dimension is a dimension for which `size(A, dim) = 1`. Singleton dimensions are fixed in size when:

- You specify a dimension with an upper bound of 1 in `coder. varsize` expressions.

For example, in this function, `Y` behaves like a vector with one variable-size dimension:

```
function Y = dim_singleton(u) %#codegen
Y = [1 2];
coder. varsize('Y', [1 10]);
if (u > 0)
    Y = [Y 3];
else
    Y = [Y u];
end
```

- You initialize variable-size data with singleton dimensions using matrix constructor expressions or matrix functions.

For example, in this function, both `X` and `Y` behave like vectors where only their second dimensions are variable sized:

```
function [X,Y] = dim_singleton_vects(u) %#codegen
Y = ones(1,3);
X = [1 4];
coder. varsize('Y','X');
if (u > 0)
    Y = [Y u];
else
    X = [X u];
end
```

You can override this behavior by using `coder. varsize` to specify explicitly that singleton dimensions vary. For example:

```
function Y = dim_singleton_vary(u) %#codegen
Y = [1 2];
coder. varsize('Y', [1 10], [1 1]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

In this example, the third argument of `coder. varsize` is a vector of ones, indicating that each dimension of `Y` varies in size. For more information, see the `coder. varsize` reference page.

### Defining Variable-Size Structure Fields

To define structure fields as variable-size arrays, use colon (:) as the index expression. The colon (:) indicates that all elements of the array are variable sized. For example:

```
function y=struct_example() %#codegen

    d = struct('values', zeros(1,0), 'color', 0);
    data = repmat(d, [3 3]);
    coder. varsize('data(:).values');

    for i = 1:numel(data)
        data(i).color = rand-0.5;
        data(i).values = 1:i;
    end

    y = 0;
    for i = 1:numel(data)
        if data(i).color > 0
            y = y + sum(data(i).values);
        end;
    end
```

The expression `coder. varsize('data(:).values')` defines the field `values` inside each element of matrix `data` to be variable sized.

Here are other examples:

- `coder. varsize('data.A(:).B')`

In this example, `data` is a scalar variable that contains matrix `A`. Each element of matrix `A` contains a variable-size field `B`.

- `coder. varsize('data(:).A(:).B')`

This expression defines field `B` inside each element of matrix `A` inside each element of matrix `data` to be variable sized.

## C Code Interface for Arrays

### In this section...

“C Code Interface for Statically Allocated Arrays” on page 7-17

“C Code Interface for Dynamically Allocated Arrays” on page 7-18

“Utility Functions for Creating emxArray Data Structures” on page 7-19

### C Code Interface for Statically Allocated Arrays

In generated code, MATLAB contains two pieces of information about statically allocated arrays: the maximum size of the array and its actual size.

For example, consider the MATLAB function `uniquetol`:

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B');
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

Generate code for `uniquetol` specifying that input `A` is a variable-size real double vector whose first dimension is fixed at 1 and second dimension can vary up to 100 elements.

```
codegen -config:lib -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

In the generated code, the function declaration is:

```
extern void uniquetol(const double A_data[100], const int A_size[2],...
    double tol, emxArray_real_T *B);
```

There are two pieces of information about `A`:

- `double A_data[100]`: the maximum size of input `A` (where 100 is the maximum size specified using `coder.typeof`).
- `int A_size[2]`: the actual size of the input.

## C Code Interface for Dynamically Allocated Arrays

In generated code, MATLAB represents dynamically allocated data as a structure type called `emxArray`. An embeddable version of the MATLAB `mxArray`, the `emxArray` is a family of data types, specialized for all base types.

### `emxArray` Structure Definition

```
typedef struct emxArray_<baseTypedef>
{
    <baseType> *data;
    int *size;
    int allocatedSize;
    int numDimensions;
    boolean_T canFreeData;
} emxArray_<baseTypedef>;
```

where `baseTypedef` is the predefined type in `rtwtypes.h` corresponding to `baseType`. For example, here's the definition for an `emxArray` of base type `double` with unknown upper bounds:

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

Note that the predefined type corresponding to `double` is `real_T`. For more information on the correspondence between built-in data types and predefined types in `rtwtypes.h`, see “How MATLAB Coder Infers C/C++ Data Types”.

To define two variables, `in1` and `in2`, of this type, use this statement:

```
emxArray_real_T *in1, *in2;
```

### C Code Interface for Structure Fields

Field	Description
<code>*data</code>	Pointer to data of type <code>&lt;baseType&gt;</code>
<code>*size</code>	Pointer to first element of size vector. Length of the vector equals the number of dimensions.

Field	Description
<code>allocatedSize</code>	Number of elements currently allocated for the array. If the size changes, MATLAB reallocates memory based on the new size.
<code>numDimensions</code>	Number of dimensions of the size vector, that is, the number of dimensions you can access without crossing into unallocated or unused memory
<code>canFreeData</code>	Boolean flag indicating how to deallocate memory: <ul style="list-style-type: none"> <li>• <code>true</code> – MATLAB deallocates memory automatically</li> <li>• <code>false</code> – Calling program determines when to deallocate memory</li> </ul>

## Utility Functions for Creating `emxArray` Data Structures

When you generate code that uses variable-size data, the code generation software exports a set of utility functions that you can use to create and interact with `emxArrays` in your generated code. To call these functions in your main C function, include the generated header file. For example, when you generate code for function `foo`, include `foo_emxAPI.h` in your main C function. For more information, see the “Write a C Main Function” section in “Using Dynamic Memory Allocation for an "Atoms" Simulation”.

---

**Note:** The code generation software exports `emxArray` utility functions only for variable-size arrays that are entry-point function inputs or outputs.

---

Function	Arguments	Description
<code>emxArray_&lt;baseType&gt;</code> <code>*emxCreateWrapper_&lt;baseType&gt;</code> <code>(...)</code>	<code>*data</code> <code>num_rows</code> <code>num_cols</code>	Creates a new 2-dimensional <code>emxArray</code> , but does not allocate it on the heap. Instead uses memory provided by the user and sets <code>canFreeData</code> to

Function	Arguments	Description
		<code>false</code> so it does not inadvertently free user memory, such as the stack.
<code>emxArray_&lt;baseType&gt;</code> <code>*emxCreateWrapperND_&lt;baseType&gt;</code> <code>(...)</code>	<code>*data</code> <code>numDimensions</code> <code>*size</code>	Same as <code>emxCreateWrapper</code> , except it creates a new N-dimensional <code>emxArray</code> .
<code>emxArray_&lt;baseType&gt;</code> <code>*emxCreate_&lt;baseType&gt;</code> <code>(...)</code>	<code>num_rows</code> <code>num_cols</code>	Creates a new two-dimensional <code>emxArray</code> on the heap, initialized to zero. All data elements have the data type specified by <i>baseTypeName</i> .
<code>emxArray_&lt;baseType&gt;</code> <code>*emxCreateND_&lt;baseType&gt;</code> <code>(...)</code>	<code>numDimensions</code> <code>*size</code>	Same as <code>emxCreate</code> , except it creates a new N-dimensional <code>emxArray</code> on the heap.
<code>emxArray_&lt;baseType&gt;</code> <code>*emxDestroyArray_&lt;baseType&gt;</code> <code>(...)</code>	<code>*emxArray</code>	Frees dynamic memory allocated by <code>*emxCreate</code> and <code>*emxCreateND</code> functions.



## Diagnose and Fix Variable-Size Data Errors

### In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 7-21

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 7-23

### Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

### Assigning Variable-Size Matrices to Fixed-Size Matrices

You cannot assign variable-size matrices to fixed-size matrices in generated code. Consider this example:

```
function Y = example_mismatch1(n) %#codegen
assert(n<10);
B = ones(n,n);
A = magic(3);
A(1) = mean(A(:));
if (n == 3)
    A = B;
end
Y = A;
```

Compiling this function produces this error:

```
??? Dimension 1 is fixed on the left-hand side
but varies on the right ...
```

There are several ways to fix this error:

- Allow matrix A to grow by adding the `coder. varsize` construct:

```
function Y = example_mismatch1_fix1(n) %#codegen
coder. varsize('A');
assert(n<10);
B = ones(n,n);
A = magic(3);
```

```
A(1) = mean(A(:));  
if (n == 3)  
    A = B;  
end  
Y = A;
```

- Explicitly restrict the size of matrix **B** to 3-by-3 by modifying the `assert` statement:

```
function Y = example_mismatch1_fix2(n) %#codegen  
coder.varsize('A');  
assert(n==3)  
B = ones(n,n);  
A = magic(3);  
A(1) = mean(A(:));  
if (n == 3)  
    A = B;  
end  
Y = A;
```

- Use explicit indexing to make **B** the same size as **A**:

```
function Y = example_mismatch1_fix3(n) %#codegen  
assert(n<10);  
B = ones(n,n);  
A = magic(3);  
A(1) = mean(A(:));  
if (n == 3)  
    A = B(1:3, 1:3);  
end  
Y = A;
```

## Empty Matrix Reshaped to Match Variable-Size Specification

If you assign an empty matrix `[]` to variable-size data, MATLAB might silently reshape the data in generated code to match a `coder.varsize` specification. For example:

```
function Y = test(u) %#codegen  
Y = [];  
coder.varsize('Y', [1 10]);  
If u < 0  
    Y = [Y u];  
end
```

In this example, `coder. varsize` defines `Y` as a column vector of up to 10 elements, so its first dimension is fixed at size 1. The statement `Y = []` designates the first dimension of `Y` as 0, creating a mismatch. The right hand side of the assignment is an empty matrix and the left hand side is a variable-size vector. In this case, MATLAB reshapes the empty matrix `Y = []` in generated code to `Y = zeros(1,0)` so it matches the `coder. varsize` specification.

## Performing Binary Operations on Fixed and Variable-Size Operands

You cannot perform binary operations on operands of different sizes. Operands have different sizes if one has fixed dimensions and the other has variable dimensions. For example:

```
function z = mismatch_operands(n) %#codegen
    assert(n>=3 && n<10);
    x = ones(n,n);
    y = magic(3);
    z = x + y;
```

When you compile this function, you get an error because `y` has fixed dimensions (3 x 3), but `x` has variable dimensions. Fix this problem by using explicit indexing to make `x` the same size as `y`:

```
function z = mismatch_operands_fix(n) %#codegen
    assert(n>=3 && n<10);
    x = ones(n,n);
    y = magic(3);
    z = x(1:3,1:3) + y;
```

## Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

## Using Nonconstant Dimensions in a Matrix Constructor

You can define variable-size data by assigning a variable to a matrix with nonconstant dimensions. For example:

```
function y = dims_vary(u) %#codegen
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

However, compiling this function generates an error because you did not specify an upper bound for `u`.

There are several ways to fix the problem:

- Enable dynamic memory allocation and recompile. During code generation, MATLAB does not check for upper bounds when it uses dynamic memory allocation for variable-size data.
- If you do not want to use dynamic memory allocation, add an `assert` statement before the first use of `u`:

```
function y = dims_vary_fix(u) %#codegen
assert (u < 20);
if (u > 0)
    y = ones(3,u);
else
    y = zeros(3,1);
end
```

## Incompatibilities with MATLAB in Variable-Size Support for Code Generation

### In this section...

“Incompatibility with MATLAB for Scalar Expansion” on page 7-25

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-27

“Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 7-28

“Incompatibility with MATLAB in Determining Class of Empty Arrays” on page 7-29

“Incompatibility with MATLAB in Vector-Vector Indexing” on page 7-30

“Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-30

“Incompatibility with MATLAB in Concatenating Variable-Size Matrices” on page 7-31

“Dynamic Memory Allocation Not Supported for MATLAB Function Blocks” on page 7-32

### Incompatibility with MATLAB for Scalar Expansion

Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

During code generation, the standard MATLAB scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Instead, it generates a size mismatch error at run time for MEX functions. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior.

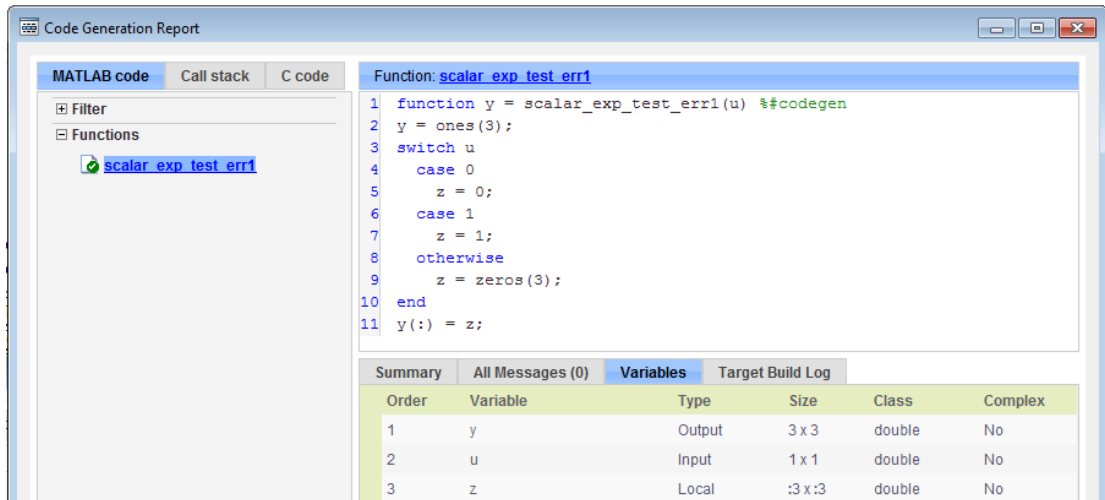
For example, in the following function, `z` is scalar for the `switch` statement `case 0` and `case 1`. MATLAB applies scalar expansion when evaluating `y(:) = z;` for these two cases.

```

function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z;

```

When you generate code for this function, the code generation software determines that  $z$  is variable size with an upper bound of 3.



If you run the MEX function with  $u$  equal to zero or one, even though  $z$  is scalar at run time, the generated code does not perform scalar expansion and a run-time error occurs.

```

scalar_exp_test_err1_mex(0)
Sizes mismatch: 9 ~= 1.

```

```

Error in scalar_exp_test_err1 (line 11)
y(:) = z;

```

### Workaround

Use indexing to force  $z$  to be a scalar value:

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z(1);
```

## Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays

For variable-size N-D arrays, the `size` function can return a different result in generated code than in MATLAB. In generated code, `size(A)` returns a fixed-length output because it does not drop trailing singleton dimensions of variable-size N-D arrays. By contrast, `size(A)` in MATLAB returns a variable-length output because it drops trailing singleton dimensions.

For example, if the shape of array `A` is `:?x:?x:?` and `size(A,3)==1`, `size(A)` returns:

- Three-element vector in generated code
- Two-element vector in MATLAB code

### Workarounds

If your application requires generated code to return the same size of variable-size N-D arrays as MATLAB code, consider one of these workarounds:

- Use the two-argument form of `size`.

For example, `size(A,n)` returns the same answer in generated code and MATLAB code.

- Rewrite `size(A)`:

```
B = size(A);
X = B(1:ndims(A));
```

This version returns `X` with a variable-length output. However, you cannot pass a variable-size `X` to matrix constructors such as `zeros` that require a fixed-size argument.

## Incompatibility with MATLAB in Determining Size of Empty Arrays

The size of an empty array in generated code might be different from its size in MATLAB source code. The size might be `1x0` or `0x1` in generated code, but `0x0` in MATLAB. Therefore, you should not write code that relies on the specific size of empty matrices.

For example, consider the following code:

```
function y = foo(n) %#codegen
x = [];
i=0;
    while (i<10)
        x = [5, x];
        i=i+1;
    end
if n > 0
    x = [];
end
y=size(x);
end
```

Concatenation requires its operands to match on the size of the dimension that is not being concatenated. In the preceding concatenation the scalar value has size `1x1` and `x` has size `0x0`. To support this use case, the code generation software determines the size for `x` as `[1 x :?]`. Because there is another assignment `x = []` after the concatenation, the size of `x` in the generated code is `1x0` instead of `0x0`.

### Workaround

If your application checks whether a matrix is empty, use one of these workarounds:

- Rewrite your code to use the `isempty` function instead of the `size` function.
- Instead of using `x=[]` to create empty arrays, create empty arrays of a specific size using `zeros`. For example:

```
function y = test_empty(n) %#codegen
x = zeros(1,0);
```



```

i=0;
while (i<10)
    x = [5, x];
    i=i+1;
end
if n > 0
    x = zeros(1,0);
end
y=size(x);
end

```

## Incompatibility with MATLAB in Determining Class of Empty Arrays

The class of an empty array in generated code can be different from its class in MATLAB source code. Therefore, do not write code that relies on the class of empty matrices.

For example, consider the following code:

```

function y = fun(n)
    x = [];
    if n > 1
        x = ['a', x];
    end
    y=class(x);
end

```

`fun(0)` returns `double` in MATLAB, but `char` in the generated code. When the statement `n > 1` is false, MATLAB does not execute `x = ['a', x]`. The class of `x` is `double`, the class of the empty array. However, the code generation software considers all execution paths. It determines that based on the statement `x = ['a', x]`, the class of `x` is `char`.

### Workaround

Instead of using `x=[]` to create an empty array, create an empty array of a specific class. For example, use `blanks(0)` to create an empty array of characters.

```

function y = fun(n)
    x = blanks(0);
    if n > 1
        x = ['a', x];
    end
    y=class(x);
end

```

## Incompatibility with MATLAB in Vector-Vector Indexing

In vector-vector indexing, you use one vector as an index into another vector. When either vector is variable sized, you might get a run-time error during code generation. Consider the index expression `A(B)`. The general rule for indexing is that `size(A(B)) == size(B)`. However, when both `A` and `B` are vectors, MATLAB applies a special rule: use the orientation of `A` as the orientation of the output. For example, if `size(A) == [1 5]` and `size(B) == [3 1]`, then `size(A(B)) == [1 3]`.

In this situation, if the code generation software detects that both `A` and `B` are vectors at compile time, it applies the special rule and gives the same result as MATLAB. However, if either `A` or `B` is a variable-size matrix (has shape `?x?`) at compile time, the code generation software applies only the general indexing rule. Then, if both `A` and `B` become vectors at run time, the code generation software reports a run-time error when you run the MEX function. Run-time error checking does not occur for non-MEX builds; the generated code will have unspecified behavior. It is best practice to generate and test a MEX function before generating C code.

### Workaround

Force your data to be a vector by using the colon operator for indexing: `A(B(:))`. For example, suppose your code intentionally toggles between vectors and regular matrices at run time. You can do an explicit check for vector-vector indexing:

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
    D = A(B);
end
...
```

The indexing in the first branch specifies that `C` and `B(:)` are compile-time vectors. As a result, the code generation software applies the standard vector-vector indexing rule.

## Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

The following limitation applies to matrix indexing operations for code generation:

- Initialization of the following style:

```
for i = 1:10
    M(i) = 5;
end
```

In this case, the size of `M` changes as the loop is executed. Code generation does not support increasing the size of an array over time.

For code generation, preallocate `M` as highlighted in the following code.

```
M=zeros(1,10);
for i = 1:10
    M(i) = 5;
end
```

The following limitation applies to matrix indexing operations for code generation when dynamic memory allocation is disabled:

- `M(i:j)` where `i` and `j` change in a loop

During code generation, memory is not dynamically allocated for the size of the expressions that change as the program executes. To implement this behavior, use `for`-loops as shown in the following example:

```
...
M = ones(10,10);
for i=1:10
    for j = i:10
        M(i,j) = 2 * M(i,j);
    end
end
...
```

---

**Note:** The matrix `M` must be defined before entering the loop, as shown in the highlighted code.

---

## Incompatibility with MATLAB in Concatenating Variable-Size Matrices

For code generation, when you concatenate variable-sized arrays, the dimensions that are not being concatenated must match exactly.

## **Dynamic Memory Allocation Not Supported for MATLAB Function Blocks**

You cannot use dynamic memory allocation for variable-size data in MATLAB Function blocks. Use bounded instead of unbounded variable-size data.

# Variable-Sizing Restrictions for Code Generation of Toolbox Functions

**In this section...**

“Common Restrictions” on page 7-33

“Toolbox Functions with Variable Sizing Restrictions” on page 7-34

## Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in “Toolbox Functions with Variable Sizing Restrictions” on page 7-34.

### Variable-length vector restriction

Inputs to the library function must be variable-length vectors or fixed-size vectors. A variable-length vector is a variable-size array that has the shape  $1 \times n$  or  $n \times 1$  (one dimension is variable sized and the other is fixed at size 1). Other shapes are not permitted, even if they are vectors at run time.

### Automatic dimension restriction

When the function selects the working dimension automatically, it bases the selection on the upper bounds for the dimension sizes. In the case of the `sum` function, `sum(X)` selects its working dimension automatically, while `sum(X, dim)` uses `dim` as the explicit working dimension.

For example, if `X` is a variable-size matrix with dimensions  $1 \times 3 \times 5$ , `sum(x)` behaves like `sum(X,2)` in generated code. In MATLAB, it behaves like `sum(X,2)` provided `size(X,2)` is not 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`. Consequently, you get a run-time error if an automatically selected working dimension assumes a length of 1 at run time.

To avoid the issue, specify the intended working dimension explicitly as a constant value.

### Array-to-vector restriction

The function issues an error when a variable-size array that is not a variable-length vector assumes the shape of a vector at run time. To avoid the issue, specify the input explicitly as a variable-length vector instead of a variable-size array.

**Array-to-scalar restriction**

The function issues an error if a variable-size array assumes a scalar value at run time. To avoid this issue, specify scalars as fixed size.

**Toolbox Functions with Variable Sizing Restrictions**

The following restrictions apply to specific toolbox functions, but only for code generation.

Function	Restrictions with Variable-Size Data
all	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>
any	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>
bsxfun	<ul style="list-style-type: none"> <li>• Dimensions expand only where one input array or the other has a fixed length of 1.</li> </ul>
cat	<ul style="list-style-type: none"> <li>• Dimension argument must be a constant.</li> <li>• An error occurs if variable-size inputs are empty at run time.</li> </ul>
conv	<ul style="list-style-type: none"> <li>• See “Variable-length vector restriction” on page 7-33.</li> <li>• Input vectors must have the same orientation, either both row vectors or both column vectors.</li> </ul>
cov	<ul style="list-style-type: none"> <li>• For <code>cov(X)</code>, see “Array-to-vector restriction” on page 7-33.</li> </ul>
cross	<ul style="list-style-type: none"> <li>• Variable-size array inputs that become vectors at run time must have the same orientation.</li> </ul>
deconv	<ul style="list-style-type: none"> <li>• For both arguments, see “Variable-length vector restriction” on page 7-33.</li> </ul>
detrend	<ul style="list-style-type: none"> <li>• For first argument for row vectors only, see “Array-to-vector restriction” on page 7-33 .</li> </ul>
diag	<ul style="list-style-type: none"> <li>• See “Array-to-vector restriction” on page 7-33 .</li> </ul>
diff	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> </ul>

Function	Restrictions with Variable-Size Data
	<ul style="list-style-type: none"> <li>Length of the working dimension must be greater than the difference order input when the input is variable sized. For example, if the input is a variable-size matrix that is 3-by-5 at run time, <code>diff(x,2,1)</code> works but <code>diff(x,5,1)</code> generates a run-time error.</li> </ul>
fft	<ul style="list-style-type: none"> <li>See “Automatic dimension restriction” on page 7-33.</li> </ul>
filter	<ul style="list-style-type: none"> <li>For first and second arguments, see “Variable-length vector restriction” on page 7-33.</li> <li>See “Automatic dimension restriction” on page 7-33.</li> </ul>
hist	<ul style="list-style-type: none"> <li>For second argument, see “Variable-length vector restriction” on page 7-33.</li> <li>For second input argument, see “Array-to-scalar restriction” on page 7-34.</li> </ul>
histc	<ul style="list-style-type: none"> <li>See “Automatic dimension restriction” on page 7-33.</li> </ul>
ifft	<ul style="list-style-type: none"> <li>See “Automatic dimension restriction” on page 7-33.</li> </ul>
ind2sub	<ul style="list-style-type: none"> <li>First input (the size vector input) must be fixed size.</li> </ul>
interp1	<ul style="list-style-type: none"> <li>For the Y input and xi input, see “Array-to-vector restriction” on page 7-33.</li> <li>Y input can become a column vector dynamically.</li> <li>A run-time error occurs if Y input is not a variable-length vector and becomes a row vector at run time.</li> </ul>
ipermute	<ul style="list-style-type: none"> <li>Order input must be fixed size.</li> </ul>
issorted	<ul style="list-style-type: none"> <li>For optional rows input, see “Variable-length vector restriction” on page 7-33.</li> </ul>
magic	<ul style="list-style-type: none"> <li>Argument must be a constant.</li> <li>Output can be fixed-size matrices only.</li> </ul>
max	<ul style="list-style-type: none"> <li>See “Automatic dimension restriction” on page 7-33.</li> </ul>
mean	<ul style="list-style-type: none"> <li>See “Automatic dimension restriction” on page 7-33.</li> <li>An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>

Function	Restrictions with Variable-Size Data
median	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>
min	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> </ul>
mode	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>
mtimes	<ul style="list-style-type: none"> <li>• When an input is variable-size, MATLAB determines whether to generate code for a general matrix*matrix multiplication or a scalar*matrix multiplication, based on whether one of the arguments is a fixed-size scalar. If neither argument is a fixed-size scalar, the inner dimensions of the two arguments must agree even if a variable-size matrix input is a scalar at run time.</li> </ul>
nchoosek	<ul style="list-style-type: none"> <li>• The second input, k, must be a fixed-size scalar.</li> <li>• The second input, k, must be a constant for static allocation. If you enable dynamic allocation, the second input can be a variable.</li> <li>• You cannot create a variable-size array by passing in a variable, k, unless you enable dynamic allocation.</li> </ul>
permute	<ul style="list-style-type: none"> <li>• Order input must be fixed-size.</li> </ul>
planerot	<ul style="list-style-type: none"> <li>• Input must be a fixed-size, two-element column vector. It cannot be a variable-size array that takes on the size 2-by-1 at run time.</li> </ul>
poly	<ul style="list-style-type: none"> <li>• See “Variable-length vector restriction” on page 7-33.</li> </ul>
polyfit	<ul style="list-style-type: none"> <li>• For first and second arguments, see “Variable-length vector restriction” on page 7-33.</li> </ul>
prod	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>



Function	Restrictions with Variable-Size Data
rand	<ul style="list-style-type: none"> <li>• For an upper-bounded variable <math>N</math>, <code>rand(1,N)</code> produces a variable-length vector of <math>1 \times M</math> where <math>M</math> is the upper bound on <math>N</math>.</li> <li>• For an upper-bounded variable <math>N</math>, <code>rand([1,N])</code> may produce a variable-length vector of <math>:1 \times M</math> where <math>M</math> is the upper bound on <math>N</math>.</li> </ul>
Generated fixed-point code enhancements	<ul style="list-style-type: none"> <li>• For an upper-bounded variable <math>N</math>, <code>randn(1,N)</code> produces a variable-length vector of <math>1 \times M</math> where <math>M</math> is the upper bound on <math>N</math>.</li> <li>• For an upper-bounded variable <math>N</math>, <code>randn([1,N])</code> may produce a variable-length vector of <math>:1 \times M</math> where <math>M</math> is the upper bound on <math>N</math>.</li> </ul>
Generated fixed-point code enhancements	<ul style="list-style-type: none"> <li>• For an upper-bounded variable <math>N</math>, <code>randn(1,N)</code> produces a variable-length vector of <math>1 \times M</math> where <math>M</math> is the upper bound on <math>N</math>.</li> <li>• For an upper-bounded variable <math>N</math>, <code>randn([1,N])</code> may produce a variable-length vector of <math>:1 \times M</math> where <math>M</math> is the upper bound on <math>N</math>.</li> </ul>
reshape	<ul style="list-style-type: none"> <li>• If the input is a variable-size array and the output array has at least one fixed-length dimension, do not specify the output dimension sizes in a size vector <code>SZ</code>. Instead, specify the output dimension sizes as scalar values, <code>sz1, . . . , szN</code>. Specify fixed-size dimensions as constants.</li> <li>• When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.</li> </ul>
roots	<ul style="list-style-type: none"> <li>• See “Variable-length vector restriction” on page 7-33.</li> </ul>

Function	Restrictions with Variable-Size Data
shiftdim	<ul style="list-style-type: none"> <li>• If you do not supply the second argument, the number of shifts is determined at compilation time by the upper bounds of the dimension sizes. Consequently, at run time the number of shifts is constant.</li> <li>• An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant).</li> <li>• First input argument must have the same number of dimensions when you supply a positive number of shifts.</li> </ul>
std	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.</li> </ul>
sub2ind	<ul style="list-style-type: none"> <li>• First input (the size vector input) must be fixed size.</li> </ul>
sum	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>
trapz	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.</li> </ul>
typecast	<ul style="list-style-type: none"> <li>• See “Variable-length vector restriction” on page 7-33 on first argument.</li> </ul>
var	<ul style="list-style-type: none"> <li>• See “Automatic dimension restriction” on page 7-33.</li> <li>• An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.</li> </ul>

# Code Generation for MATLAB Structures

---

- “Structure Definition for Code Generation” on page 8-2
- “Structure Operations Allowed for Code Generation” on page 8-3
- “Define Scalar Structures for Code Generation” on page 8-4
- “Define Arrays of Structures for Code Generation” on page 8-7
- “Make Structures Persistent” on page 8-9
- “Index Substructures and Fields” on page 8-10
- “Assign Values to Structures and Fields” on page 8-12
- “Pass Structure Arguments by Reference or by Value” on page 8-14

## Structure Definition for Code Generation

To generate efficient standalone code for structures, you must define and use structures differently than you normally would when running your code in the MATLAB environment:

<b>What's Different</b>	<b>More Information</b>
Use a restricted set of operations.	“Structure Operations Allowed for Code Generation” on page 8-3
Observe restrictions on properties and values of scalar structures.	“Define Scalar Structures for Code Generation” on page 8-4
Make structures uniform in arrays.	“Define Arrays of Structures for Code Generation” on page 8-7
Reference structure fields individually during indexing.	“Index Substructures and Fields” on page 8-10
Avoid type mismatch when assigning values to structures and fields.	“Assign Values to Structures and Fields” on page 8-12

## Structure Operations Allowed for Code Generation

To generate efficient standalone code for MATLAB structures, you are restricted to the following operations:

- Define structures as local and persistent variables by assignment and using the `struct` function
- Index structure fields using dot notation
- Define primary function inputs as structures
- Pass structures to local functions

## Define Scalar Structures for Code Generation

### In this section...

“Restriction When Using struct” on page 8-4

“Restrictions When Defining Scalar Structures by Assignment” on page 8-4

“Adding Fields in Consistent Order on Each Control Flow Path” on page 8-4

“Restriction on Adding New Fields After First Use” on page 8-5

### Restriction When Using struct

When you use the `struct` function to create scalar structures for code generation, you cannot create structures of cell arrays.

### Restrictions When Defining Scalar Structures by Assignment

When you define a scalar structure by assigning a variable to a preexisting structure, you do not need to define the variable before the assignment. However, if you already defined that variable, it must have the same class, size, and complexity as the structure you assign to it. In the following example, `p` is defined as a structure that has the same properties as the predefined structure `S`:

```
...
S = struct('a', 0, 'b', 1, 'c', 2);
p = S;
...
```

### Adding Fields in Consistent Order on Each Control Flow Path

When you create a structure, you must add fields in the same order on each control flow path. For example, the following code generates a compiler error because it adds the fields of structure `x` in a different order in each `if` statement clause:

```
function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.b = 30; % Generates an error (on variable x)
```

```

    x.a = 40;
end
y = x.a + x.b;

```

In this example, the assignment to `x.a` comes before `x.b` in the first `if` statement clause, but the assignments appear in reverse order in the `else` clause. Here is the corrected code:

```

function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.a = 40;
    x.b = 30;
end
y = x.a + x.b;

```

## Restriction on Adding New Fields After First Use

You cannot add fields to a structure after you perform the following operations on the structure:

- Reading from the structure
- Indexing into the structure array
- Passing the structure to a function

For example, consider this code:

```

...
x.c = 10; % Defines structure and creates field c
y = x; % Reads from structure
x.d = 20; % Generates an error
...

```

In this example, the attempt to add a new field `d` after reading from structure `x` generates an error.

This restriction extends across the structure hierarchy. For example, you cannot add a field to a structure after operating on one of its fields or nested structures, as in this example:

```

function y = fcn(u) %#codegen

```

```
x.c = 10;  
y = x.c;  
x.d = 20; % Generates an error
```

In this example, the attempt to add a new field `d` to structure `x` after reading from the structure's field `c` generates an error.



## Define Arrays of Structures for Code Generation

### In this section...

“Ensuring Consistency of Fields” on page 8-7

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 8-7

“Defining an Array of Structures Using Concatenation” on page 8-8

### Ensuring Consistency of Fields

When you create an array of MATLAB structures with the intent of generating code, you must be sure that each structure field in the array has the same size, type, and complexity.

Once you have created the array of structures, you can make the structure fields variable-size using `coder. varsizes`. For more information, see “Declare a variable-size structure field.”

### Using repmat to Define an Array of Structures with Consistent Field Properties

You can create an array of structures from a scalar structure by using the MATLAB `repmat` function, which replicates and tiles an existing scalar structure:

- 1 Create a scalar structure, as described in “Define Scalar Structures for Code Generation” on page 8-4.
- 2 Call `repmat`, passing the scalar structure and the dimensions of the array.
- 3 Assign values to each structure using standard array indexing and structure dot notation.

For example, the following code creates `X`, a 1-by-3 array of scalar structures. Each element of the array is defined by the structure `s`, which has two fields, `a` and `b`:

```
...  
s.a = 0;  
s.b = 0;  
X = repmat(s,1,3);  
X(1).a = 1;
```

```
X(2).a = 2;  
X(3).a = 3;  
X(1).b = 4;  
X(2).b = 5;  
X(3).b = 6;  
...
```

## Defining an Array of Structures Using Concatenation

To create a small array of structures, you can use the concatenation operator, square brackets ( `[ ]` ), to join one or more structures into an array (see “Concatenating Matrices”). For code generation, the structures that you concatenate must have the same size, class, and complexity.

For example, the following code uses concatenation and a local function to create the elements of a 1-by-3 structure array:

```
...  
W = [ sab(1,2) sab(2,3) sab(4,5) ];  
  
function s = sab(a,b)  
    s.a = a;  
    s.b = b;  
...
```

## Make Structures Persistent

To make structures persist, you define them to be persistent variables and initialize them with the `isempty` statement, as described in “Define and Initialize Persistent Variables” on page 5-10.

For example, the following function defines structure `X` to be persistent and initializes its fields `a` and `b`:

```
function f(u) %#codegen
persistent X

if isempty(X)
    X.a = 1;
    X.b = 2;
end
```

## Index Substructures and Fields

Use these guidelines when indexing substructures and fields for code generation:

### Reference substructure field values individually using dot notation

For example, the following MATLAB code uses dot notation to index fields and substructures:

```
...
substruct1.a1 = 15.2;
substruct1.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                 'ele3',substruct1);

substruct2 = mystruct;
substruct2.ele3.a2 = 2*(substruct1.a2);
...
```

The generated code indexes elements of the structures in this example by resolving symbols as follows:

Dot Notation	Symbol Resolution
substruct1.a1	Field a1 of local structure substruct1
substruct2.ele3.a1	Value of field a1 of field ele3, a substructure of local structure substruct2
substruct2.ele3.a2(1,1)	Value in row 1, column 1 of field a2 of field ele3, a substructure of local structure substruct2

### Reference field values individually in structure arrays

To reference the value of a field in a structure array, you must index into the array to the structure of interest and then reference that structure's field individually using dot notation, as in this example:

```
...  
y = X(1).a % Extracts the value of field a  
           % of the first structure in array X  
...
```

To reference all the values of a particular field for each structure in an array, use this notation in a `for` loop, as in this example:

```
...  
s.a = 0;  
s.b = 0;  
X = repmat(s,1,5);  
for i = 1:5  
    X(i).a = i;  
    X(i).b = i+1;  
end
```

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 8-7 for more information.

## Do not reference fields dynamically

You cannot reference fields in a structure by using dynamic names, which express the field as a variable expression that MATLAB evaluates at run time (see “Generate Field Names from Variables”).

## Assign Values to Structures and Fields

Use these guidelines when assigning values to a structure, substructure, or field for code generation:

### Field properties must be consistent across structure-to-structure assignments

If:	Then:
Assigning one structure to another structure.	Define each structure with the same number, type, and size of fields.
Assigning one structure to a substructure of a different structure and vice versa.	Define the structure with the same number, type, and size of fields as the substructure.
Assigning an element of one structure to an element of another structure.	The elements must have the same type and size.

### Do not use field values as constants

The values stored in the fields of a structure are not treated as constant values in generated code. Therefore, you cannot use field values to set the size or class of other data. For example, the following code generates a compiler error if variable-sizing is disabled:

```
...  
Y.a = 3;  
Y.b = 5;  
X = zeros(Y.a,Y.b); % Generates an error
```

In this example, even though you set fields `a` and `b` of structure `Y` to the values 3 and 5 respectively, `Y.a` and `Y.b` are not constants in generated code. Therefore, they are not valid arguments to pass to the function `zeros`.

---

**Note:** An exception to this behavior occurs if the structure is declared completely using the `struct` function

```
...  
Y = struct('a',3,'b',5);  
X = zeros(Y.a,Y.b); % Generates a fixed-size 3 X 5 matrix
```

---

## Do not assign mxArray's to structures

You cannot assign mxArray's to structure elements; convert mxArray's to known types before code generation (see “Working with mxArray's” on page 13-17).

## Pass Structure Arguments by Reference or by Value

### In this section...

“Specify Pass by Reference or by Value Using a Project” on page 8-14

“Specify Pass by Reference or by Value Using the Command-Line Interface” on page 8-15

“Pass Input Structure Argument by Reference” on page 8-15

“Pass Input Structure Argument by Value” on page 8-16

“Pass Output Structure Argument by Reference” on page 8-16

“Pass Output Structure Argument by Value” on page 8-17

“Pass Input and Output Structure Argument by Reference” on page 8-18

For standalone C code generation, you can control whether a generated entry-point function passes structure arguments by reference or by value. Passing by reference uses a pointer to access the structure arguments. If the function writes to an element of the input structure, it overwrites the input value. Passing by value makes a copy of the input or output structure argument. To reduce memory usage and execution time, use pass by reference.

If a structure argument is both an input and output, the generated entry-point function passes the argument by reference.

Generated MEX functions pass structure arguments by reference. For MEX function output, you cannot specify that you want to pass structure arguments by value.

### Specify Pass by Reference or by Value Using a Project

- 1 On the **Build** tab, set the **Output Type** to one of the following:
  - C/C++ Static Library
  - C/C++ Dynamic Library
  - C/C++ Executable
- 2 To open the **Project Settings** dialog box, click the **More settings** link.
- 3 On the **All Settings** tab, set the **Pass structures by reference to entry-point functions** option.



Set To	For
Yes	Pass by reference (default)
No	Pass by value

## Specify Pass by Reference or by Value Using the Command-Line Interface

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

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

- 2 Set the PassStructByReference property.

Set To	For
true	Pass by reference
false	Pass by value (default)

For example,

```
cfg.PassStructByReference = true;
```

## Pass Input Structure Argument by Reference

This example shows how to generate an entry-point function that passes an input structure argument by reference.

- 1 Write the function `my_struct_in`.

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

- 2 Declare a structure variable `str` in the MATLAB workspace.

```
str = struct('f', 1:4);
```

- 3 Create a code configuration object to generate a C static library.

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

- 4 Specify that you want to pass structure arguments by reference.

```
cfg.PassStructByReference = true;
```

- 5 Generate code. Specify that the input argument has the type of the variable `str`.

```
codegen -config cfg -args {str} my_struct_in -report
```

- 6 To view the generated C code, click **View report**.

The generated function signature for `my_struct_in` is:

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

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

## Pass Input Structure Argument by Value

This example shows how to generate an entry-point function that passes an input structure argument by value.

- 1 Write the function `my_struct_in`.

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

- 2 Declare a structure variable `str` in the MATLAB workspace.

```
str = struct('f', 1:4);
```

- 3 Create a code configuration object to generate a C static library.

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

- 4 Specify that you want to pass structure arguments by value.

```
cfg.PassStructByReference = false;
```

By default, the `PassStructByReference` parameter is `false`.

- 5 Generate code. Specify that the input argument has the type of the variable `str`.

```
codegen -config cfg -args {str} my_struct_in -report
```

- 6 To view the generated C code, click **View report**.

The generated function signature for `my_struct_in` is:

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

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

## Pass Output Structure Argument by Reference

This example shows how to generate an entry-point function that passes an output structure argument by reference.

- 1 Write the function `my_struct_out`.

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

- 2 Declare a variable `a` in the MATLAB workspace.

```
a = 1:4;
```

- 3 Create a code configuration object to generate a C static library.

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

- 4 Specify that you want to pass structure arguments by reference.

```
cfg.PassStructByReference = true;
```

- 5 Generate code. Specify that the input argument has the type of the variable `a`.

```
codegen -config cfg -args {a} my_struct_out -report
```

- 6 To view the generated C code, click **View report**.

The generated function signature for `my_struct_out` is:

```
void my_struct_out(const double x[4], struct0_T *s)
```

`my_struct_out` passes the output structure `s` by reference.

## Pass Output Structure Argument by Value

This example shows how to generate an entry-point function that passes an output structure argument by value.

- 1 Write the function `my_struct_out`.

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

- 2 Declare a variable `a` in the MATLAB workspace.

```
a = 1:4;
```

- 3 Create a code configuration object to generate a C static library.

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

- 4 Specify that you want to pass structure arguments by value.

```
cfg.PassStructByReference = false;
```

By default, the `PassStructByReference` parameter is `false`

- 5 Generate code. Specify that the input argument has the type of the variable `a`.

```
codegen -config cfg -args {a} my_struct_out -report
```

- 6 To view the generated C code, click **View report**.

The generated function signature for `my_struct_out` is:

```
struct0_T my_struct_out(const double x[4])
```

`my_struct_out` returns an output structure.

## Pass Input and Output Structure Argument by Reference

This example shows how a generated entry-point function passes a structure argument by reference when the structure argument is both an input and an output. In this case, the function passes the structure argument by reference even though you set `PassStructByReference` to `false`.

- 1 Write the function `my_struct_inout`.

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

- 2 Define the variable `a` and structure `str` in the MATLAB workspace.

```
a = 1:4;  
str = struct('f',a);
```

- 3 Create a code configuration object to generate a C static library.

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

- 4 Specify that you want to pass structure arguments by value.

```
cfg.PassStructByReference = false;
```

By default, the `PassStructByReference` parameter is `false`.

- 5 Generate code. Specify that the first input has the type of `a` and the second input has the type of `str`.

```
codegen -config cfg -args {a, str} my_struct_inout -report
```

- 6 To view the generated C code, click **View report**.

The generated function signature for `my_struct_inout` is:

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

`my_struct_inout` passes the structure `s` by reference even though `PassStructByReference` is `false`.



# Code Generation for Enumerated Data

---

- “Enumerated Data Definition for Code Generation” on page 9-2
- “Enumerated Types Supported for Code Generation” on page 9-3
- “When to Use Enumerated Data for Code Generation” on page 9-6
- “Generate Code for Enumerated Data from MATLAB Algorithms” on page 9-7
- “Define Enumerated Data for Code Generation” on page 9-8
- “Operations on Enumerated Data for Code Generation” on page 9-10
- “Include Enumerated Data in Control Flow Statements” on page 9-13
- “Customize Enumerated Types for Code Generation” on page 9-19
- “Use Enumerated Types in LED Control Function” on page 9-23
- “Control Names of Enumerated Type Values in Generated Code” on page 9-26
- “Change and Reload Enumerated Data Types” on page 9-29
- “Restrictions on Use of Enumerated Data in for-Loops” on page 9-30
- “Toolbox Functions That Support Enumerated Types for Code Generation” on page 9-31

## Enumerated Data Definition for Code Generation

To generate efficient standalone code for enumerated data, you must define and use enumerated types differently than you do in the MATLAB environment:

Difference	More Information
Supports integer-based enumerated types only	“Enumerated Types Supported for Code Generation” on page 9-3
Name of each enumerated data type must be unique	“Naming Enumerated Types for Code Generation” on page 9-9
Each enumerated data type must be defined in a separate file on the MATLAB path	“Define Enumerated Data for Code Generation” on page 9-8 and “Generate Code for Enumerated Data from MATLAB Algorithms” on page 9-7
Restricted set of operations	“Operations on Enumerated Data for Code Generation” on page 9-10
Restricted use in <code>for</code> -loops	“Restrictions on Use of Enumerated Data in <code>for</code> -Loops” on page 9-30



## Enumerated Types Supported for Code Generation

An enumerated type is a user-defined type whose values belong to a predefined set of enumerated values. Each enumerated value consists of a name and an underlying numeric value.

You define an enumerated data type in an enumeration class definition file. For code generation, you must base the class on `int8`, `uint8`, `int16`, `uint16`, or `int32`. For example:

```
classdef(Enumeration) PrimaryColors < int32
    enumeration
        Red(1),
        Blue(2),
        Yellow(4)
    end
end
```

In this example, the statement `classdef(Enumeration) PrimaryColors < int32` means that the enumerated type `PrimaryColors` is based on the built-in type `int32`. `PrimaryColors` inherits the characteristics of the `int32` type. It also defines its own unique characteristics. For example, `PrimaryColors` is restricted to three enumerated values:

Enumerated Value	Enumerated Name	Underlying Numeric Value
Red(1)	Red	1
Blue(2)	Blue	2
Yellow(4)	Yellow	4

### Enumeration Class Base Types for Code Generation

For code generation, you must 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 an enumerated type in generated C/C++ code. You can:

- 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 type in generated C/C++ code.

## C Code Representation for Base Type `int32`

If the base type is `int32`, the code generation software generates a C enumeration type. Consider the following MATLAB enumerated type definition:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

This enumerated type definition results in the following C code:

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

typedef enum LEDcolor LEDcolor;
```

## C Code Representation for Base Type Other Than `int32`

For built-in integer base types other than `int32`, the code generation software generates a `typedef` statement for the enumerated type and `#define` statements for the enumerated values. Consider the following MATLAB enumerated type definition:

```
classdef(Enumeration) LEDcolor < int16
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

end

This enumerated type definition results in the following C code:

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

The C type in the `typedef` statement depends on:

- The integer sizes defined for the production hardware in the Hardware Implementation object or the project settings. See `coder.HardwareImplementation`.
- The setting that determines use of built-in C types or MathWorks typedefs in the generated code. See “Specify Data Type Used in Generated Code” and “How MATLAB Coder Infers C/C++ Data Types” .

## When to Use Enumerated Data for Code Generation

You can use enumerated types to represent program states and to control program logic, especially when you restrict data to a predetermined set of values and refer to these values by name. You can sometimes achieve these goals by using integers or strings, however, enumerated types offer the following advantages:

- More readable code than integers.
- More robust error checking than integers or strings.

For example, if you mistype the name of an element in the enumerated type, you get a compile-time error that the element does not belong to the set of allowable values.

- More efficient code than strings.

For example, comparisons of enumerated values execute faster than comparisons of strings.

## Generate Code for Enumerated Data from MATLAB Algorithms

The basic workflow for generating code for enumerated types in MATLAB code is:

- 1 Define an enumerated data type that inherits from a base type that code generation supports. See “Enumerated Types Supported for Code Generation”.
- 2 Save the enumerated data type in a file on the MATLAB path.
- 3 Write a MATLAB function that uses the enumerated type.
- 4 Specify enumerated type inputs using the project or the command-line interface.
- 5 Generate code.

### See Also

- “Use Enumerated Types in LED Control Function”
- “Define Enumerated Data for Code Generation”
- “Specifying an Enumerated Type Input Parameter by Example”
- “Specifying an Enumerated Type Input Parameter by Type”

## Define Enumerated Data for Code Generation

To define enumerated data for code generation from MATLAB algorithms:

- 1 Create a class definition file.

In the Command Window, select **File > New > Class**.

- 2 Enter the class definition:

```
classdef(Enumeration) EnumTypeName < BaseType
```

*EnumTypeName* is a case-sensitive string that must be unique among data type names and workspace variable names. *BaseType* must be `int8`, `uint8`, `int16`, `uint16`, or `int32`.

For example, the following code defines an enumerated type called `sysMode` that inherits from the built-in type `int32`:

```
classdef(Enumeration) sysMode < int32
    ...
end
```

- 3 Define enumerated values in an enumeration section:

```
classdef(Enumeration) EnumTypeName < BaseType
    enumeration
        EnumName(N)
        ...
    end
end
```

For example, the following code defines a set of two values for enumerated type `sysMode`:

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0),
        ON(1)
    end
end
```

Each enumerated value consists of a string *EnumName* and an underlying integer *N*. Each *EnumName* must be unique within its type. If the enumerated value name

does not include the class name prefix, *EnumName* must be unique across enumerated types. See “Control Names of Enumerated Type Values in Generated Code” on page 9-26.

The underlying integers do not have to be consecutive or ordered, or unique within or across types.

#### **4** Save the file on the MATLAB path.

The name of the file must match the name of the enumerated data type. The match is case sensitive.

For examples, see “Include Enumerated Data in Control Flow Statements” on page 9-13.

## **Naming Enumerated Types for Code Generation**

You must use a unique name for each enumerated data type. Do not use the name of:

- A toolbox function supported for code generation.
- Another data type.
- A variable in the MATLAB base workspace.

For example, you cannot name an enumerated data type `mode` because MATLAB for code generation provides a toolbox function of the same name.

For a list of toolbox functions supported for code generation, see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List”.

## Operations on Enumerated Data for Code Generation

To generate efficient standalone code for enumerated data, you are restricted to the following operations. The examples use the following enumerated class definition:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

### Assignment Operator, =

Example	Result
<pre>xon = LEDcolor.GREEN xoff = LEDcolor.RED</pre>	<pre>xon =     GREEN xoff =     RED</pre>

### Relational Operators, < > <= >= == ~=

Example	Result
<pre>xon == xoff</pre>	<pre>ans =     0</pre>
<pre>xon &lt;= xoff</pre>	<pre>ans =     1</pre>
<pre>xon &gt; xoff</pre>	<pre>ans =     0</pre>

### Cast Operation

Example	Result
<pre>double(LEDcolor.RED)</pre>	<pre>ans =</pre>



Example	Result
	2
<pre>z = 2 y = LEDcolor(z)</pre>	<pre>z =   2  y =   RED</pre>

## Indexing Operation

Example	Result
<pre>m = [1 2] n = LEDcolor(m) p = n(LEDcolor.GREEN)</pre>	<pre>m =   1    2  n =   GREEN  RED  p =   GREEN</pre>

## Control Flow Statements: if, switch, while

Statement	Example	Executable Example
if	<pre>if state == sysMode.ON     led = LEDcolor.GREEN; else     led = LEDcolor.RED; end</pre>	“if Statement with Enumerated Data Types” on page 9-13
switch	<pre>switch button case VCRButton.Stop     state = VCRState.Stop;</pre>	“switch Statement with Enumerated Data Types” on page 9-14

Statement	Example	Executable Example
	<pre> case VCRButton.PlayOrPause     state = VCRState.Play; case VCRButton.Next     state = VCRState.Forward; case VCRButton.Previous     state = VCRState.Rewind; otherwise     state = VCRState.Stop; end                     </pre>	
while	<pre> while state ~= State.Ready     switch state         case State.Standby             initialize();             state = State.Boot;         case State.Boot             boot();             state = State.Ready;     end end                     </pre>	<p>“while Statement with Enumerated Data Types” on page 9-16</p>

## Include Enumerated Data in Control Flow Statements

The following examples define enumerated types that use the base type `int32`. You can base an enumerated type on one of these built-in integer types:

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

The base type determines the representation of the enumerated type in the generated C/C++ code. See “Enumerated Types Supported for Code Generation” on page 9-3.

### if Statement with Enumerated Data Types

This example defines the enumeration types `LEDcolor` and `sysMode`. The function `displayState` uses these enumerated data types to activate an LED display.

#### Class Definition: `sysMode`

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0),
        ON(1)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `sysMode.m`.

#### Class Definition: `LEDcolor`

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

This definition must reside on the MATLAB path in a file called `LEDcolor.m`.

### **MATLAB Function: displayState**

This function uses enumerated data to activate an LED display, based on the state of a device. It lights a green LED display to indicate the ON state and lights a red LED display to indicate the OFF state.

```
function led = displayState(state)
%#codegen

if state == sysMode.ON
    led = LEDcolor.GREEN;
else
    led = LEDcolor.RED;
end
```

### **Build and Test a MEX Function for displayState**

- 1 Generate a MEX function for `displayState`. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen displayState -args {sysMode.ON}
```

- 2 Test the function. For example,

```
displayState(sysMode.OFF)
```

```
ans =
```

```
    RED
```

## **switch Statement with Enumerated Data Types**

This example is based on the definition of the enumeration types `VCRState` and `VCRButton`. The function `VCR` uses these enumerated data types to set the state of the VCR.

### **Class Definition: VCRState**

```
classdef(Enumeration) VCRState < int32
    enumeration
        Stop(0),
        Pause(1),
        Play(2),
        Forward(3),
```

```

        Rewind(4)
    end
end

```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRState.m`.

### Class Definition: `VCRButton`

```

classdef(Enumeration) VCRButton < int32
    enumeration
        Stop(1),
        PlayOrPause(2),
        Next(3),
        Previous(4)
    end
end

```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRButton.m`.

### MATLAB Function: `VCR`

This function uses enumerated data to set the state of a VCR, based on the initial state of the VCR and the state of the VCR button.

```

function s = VCR(button)
    %#codegen

    persistent state

    if isempty(state)
        state = VCRState.Stop;
    end

    switch state
        case {VCRState.Stop, VCRState.Forward, VCRState.Rewind}
            state = handleDefault(button);
        case VCRState.Play
            switch button
                case VCRButton.PlayOrPause, state = VCRState.Pause;
                otherwise, state = handleDefault(button);
            end
        case VCRState.Pause

```

```
        switch button
            case VCRButton.PlayOrPause, state = VCRState.Play;
            otherwise, state = handleDefault(button);
        end
    end
end
s = state;

function state = handleDefault(button)
switch button
    case VCRButton.Stop, state = VCRState.Stop;
    case VCRButton.PlayOrPause, state = VCRState.Play;
    case VCRButton.Next, state = VCRState.Forward;
    case VCRButton.Previous, state = VCRState.Rewind;
    otherwise, state = VCRState.Stop;
end
```

### Build and Test a MEX Function for VCR

- 1 Generate a MEX function for VCR. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen -args {VCRButton.Stop} VCR
```

- 2 Test the function. For example,

```
s = VCR(VCRButton.Stop)
```

```
s =
```

```
    Stop
```

### while Statement with Enumerated Data Types

This example is based on the definition of the enumeration type `State`. The function `Setup` uses this enumerated data type to set the state of a device.

#### Class Definition: State

```
classdef(Enumeration) State < int32
    enumeration
        Standby(0),
        Boot(1),
        Ready(2)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `State.m`.

### **MATLAB Function: Setup**

The following function `Setup` uses enumerated data to set the state of a device.

```
function s = Setup(initState)
%#codegen

state = initState;

if isempty(state)
    state = State.Standby;
end

while state ~= State.Ready
    switch state
        case State.Standby
            initialize();
            state = State.Boot;
        case State.Boot
            boot();
            state = State.Ready;
    end
end
s = state;

function initialize()
% Perform initialization.

function boot()
% Boot the device.
```

### **Build and Test a MEX Executable for Setup**

- 1 Generate a MEX executable for `Setup`. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen Setup -args {State.Standby}
```

- 2 Test the function. For example,

```
s = Setup(State.Standby)
```

```
s =
```

Ready



# Customize Enumerated Types for Code Generation

## Customizing Enumerated Types

For code generation, you can customize an enumerated type by using the same techniques that work with MATLAB classes, as described in “Modifying Superclass Methods and Properties”. You can customize an enumerated type by including a methods section in the enumerated class definition. You can override the following methods to customize the behavior of an enumerated type. To override a method, include a customized version of the method in the methods section in the enumerated class definition. If you do not want to override the inherited methods, omit the methods section.

Method	Description	Default Value Returned or Specified	When to Use
<code>addClassNameToEnumNames</code>	Specifies whether the class name becomes a prefix in the generated code.	<code>false</code> — prefix is not used.	If you want the class name to become a prefix in the generated code, override this method to set the return value to <code>true</code> . See “Control Names of Enumerated Type Values in Generated Code” on page 9-26.
<code>getDefaultValue</code>	Returns the default enumerated value.	First value in the enumerated class definition.	If you want the default value for the enumerated type to be something other than the first value listed in the enumerated class definition, override this method to specify a default value. See “Specify a Default

Method	Description	Default Value Returned or Specified	When to Use
			Enumerated Value” on page 9-20.
getHeaderFile	Specifies the file in which the enumerated class is defined for code generation.	''	If you want to use an enumerated class definition that is specified in a custom header file, override this method to return the path to this header file. In this case, the code generation software does not generate the class definition. See “Specify a Header File” on page 9-21

## Specify a Default Enumerated Value

If the value of a variable that is cast to an enumerated type does not match one of the enumerated type values:

- Generated MEX reports an error.
- Generated C/C++ code replaces the value of the variable with the enumerated type default value.

Unless you specify otherwise, the default value for an enumerated type is the first value in the enumerated class definition. To specify a different default value, add your own `getDefaultValue` method to the methods section. The following code shows a shell for the `getDefaultValue` method:

```
function retVal = getDefaultValue()
% GETDEFAULTVALUE Returns the default enumerated value.
% This value must be an instance of the enumerated class.
% If this method is not defined, the first enumerated value is used.
    retVal = ThisClass.EnumName;
end
```

To customize this method, provide a value for `ThisClass.EnumName` that specifies the default that you want. `ThisClass` must be the name of the class within which the method exists. `EnumName` must be the name of an enumerated value defined in that class. For example:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end

    methods (Static)
        function y = getDefaultvalue()
            y = LEDcolor.RED;
        end
    end
end
```

This example defines the default as `LEDcolor.RED`. If this method does not appear, the default value is `LEDcolor.GREEN`, because that value is the first value listed in the enumerated class definition.

## Specify a Header File

To prevent the declaration of an enumerated type from being embedded in the generated code, allowing you to provide the declaration in an external file, include the following method in the enumerated class methods section:

```
function y = getHeaderFile()
% GETHEADERFILE File where type is defined for generated code.
% If specified, this file is #included where required in the code.
% Otherwise, the type is written out in the generated code.
y = 'filename';
end
```

Substitute a legal filename for `filename`. Be sure to provide a filename suffix, typically `.h`. Providing the method replaces the declaration that appears in the generated code with an `#include` statement like:

```
#include "imported_enum_type.h"
```

The `getHeaderFile` method does not create the declaration file itself. You must provide a file of the specified name that declares the enumerated data type. The file can also contain definitions of enumerated types that you do not use in your MATLAB code.

For example, to use the definition of `LEDcolor` in `my_LEDcolor.h`:

- 1 Modify the definition of `LEDcolor` to override the `getHeaderFile` method to return the name of the external header file:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end

    methods(Static)
        function y=getHeaderFile()
            y='my_LEDcolor.h';
        end
    end
end
```

- 2 In the current folder, provide a header file, `my_LEDcolor.h`, that contains the definition:

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

typedef enum LEDcolor LEDcolor;
```

- 3 Generate a library for the function `displayState` that takes one input of enumerated data type `sysMode`.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

`codegen` generates a C static library with the default name, `displayState`, and supporting files in the default folder, `codegen/lib/displayState`.

- 4 Click the **View Report** link.
- 5 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The header file contains a `#include` statement for the external header file.

```
#include "my_LEDcolor.h"
```

It does not include a declaration for the enumerated class.

## Use Enumerated Types in LED Control Function

This example shows how to define, use, and generate code for enumerated types in a function that controls an LED. In this example, the enumerated types inherit from base type `int32`. The base type can be `int8`, `uint8`, `int16`, `uint16`, or `int32`.

- 1 Define the enumerated type `sysMode`. Store it in `sysMode.m` on the MATLAB path.

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0),
        ON(1)
    end
end
```

- 2 Define the enumerated type `LEDcolor`. Store it in `LEDcolor.m` on the MATLAB path.

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

- 3 Define the function `displayState`, which uses enumerated data to activate an LED display, based on the state of a device. `displayState` lights a green LED display to indicate the ON state. It lights a red LED display to indicate the OFF state. Store `displayState` in `displayState.m` on the MATLAB path.

```
function led = displayState(state)
    %#codegen

    if state == sysMode.ON
        led = LEDcolor.GREEN;
    else
        led = LEDcolor.RED;
    end
```

- 4 Generate the MEX function `displayState_mex`. Specify that `displayState` has one input that is an enumerated data type `sysMode`.

```
codegen -report displayState -args {sysMode.ON}
```

- 5 Test the MEX function.

```
displayState_mex(sysMode.OFF)
```

```
ans =
```

```
    RED
```

- 6 Generate a static library for the function `displayState` that takes one input of enumerated data type `sysMode`.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

`codegen` generates a C static library with the default name, `displayState`. It generates supporting files in the default folder, `codegen/lib/displayState`.

- 7 Click the **View Report** link.

- 8 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The report displays the header file containing the enumerated data type definition.

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

```
typedef enum LEDcolor LEDcolor;
```

The enumerated type `LEDcolor` is represented as a C enumerated type because the base type in the class definition for `LEDcolor` is `int32`. When the base type is `int8`, `uint8`, `int16`, or `uint16`, the code generation software generates a `typedef` for the enumerated type. It generates `#define` statements for the enumerated type values. For example:

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

## Related Examples

- “Generate Code for Enumerated Data from MATLAB Algorithms” on page 9-7
- “Customize Enumerated Types for Code Generation” on page 9-19

## **More About**

- “Enumerated Data Definition for Code Generation” on page 9-2
- “Enumerated Types Supported for Code Generation” on page 9-3

## Control Names of Enumerated Type Values in Generated Code

This example shows how to control whether generated enumerated type value names include the class name prefix from the enumerated type definition. By default, the generated enumerated type value name does not include the class name prefix.

- 1 Define the enumerated type `sysMode`. Store it in `sysMode.m` on the MATLAB path.

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0),
        ON(1)
    end
end
```

- 2 Define the enumerated type `LEDcolor`. Store it in `LEDcolor.m` on the MATLAB path.

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end
end
```

- 3 Define the function `displayState`, which uses enumerated data to activate an LED display, based on the state of a device. `displayState` lights a green LED display to indicate the ON state and lights a red LED display to indicate the OFF state. Store in `displayState.m` on the MATLAB path.

```
function led = displayState(state)
    %#codegen

    if state == sysMode.ON
        led = LEDcolor.GREEN;
    else
        led = LEDcolor.RED;
    end
end
```

- 4 Generate a library for the function `displayState` that takes one input of enumerated data type `sysMode`.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```



codegen generates a C static library with the default name, `displayState`, and supporting files in the default folder, `codegen/lib/displayState`.

- 5 Click the **View Report** link.
- 6 In the report, on the **C Code** tab, click the link to the `displayState_types.h` file.

The report displays the header file containing the enumerated data type definition.

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

typedef enum LEDcolor LEDcolor;
```

The enumerated value names do not include the class name prefix `LEDcolor_`.

- 7 Modify the definition of `LEDcolor` to add the `addClassNameToEnumNames` method. Set the return value to `true` so that the enumerated value names in the generated code contain the class prefix.

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end

    methods(Static)
        function y=addClassNameToEnumNames()
            y=true;
        end
    end
end
```

- 8 Clear existing class instances.

```
clear classes
```

- 9 Generate code.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

- 10 Open the code generation report and look at the enumerated type definition in `displayState_types.h`.

```
enum LEDcolor
{
    LEDcolor_GREEN = 1,
    LEDcolor_RED
};
```

```
typedef enum LEDcolor LEDcolor;
```

The enumerated value names include the class name prefix.

## Change and Reload Enumerated Data Types

You can change the definition of an enumerated data type by editing and saving the file that contains the definition. You do not need to inform MATLAB that a class definition has changed. MATLAB automatically reads the modified definition when you save the file. However, the class definition changes do not take full effect if class instances (enumerated values) exist that reflect the previous class definition. Such instances might exist in the base workspace or might be cached. The following table explains options for removing instances of an enumerated data type from the base workspace and cache.

If In Base Workspace...	If In Cache...
<p>Do one of the following:</p> <ul style="list-style-type: none"><li>• Locate and delete specific obsolete instances.</li><li>• Delete the classes from the workspace by using the <code>clear classes</code> command. For more information, see <code>clear</code>.</li></ul>	<ul style="list-style-type: none"><li>• Clear MEX functions that are caching instances of the class.</li></ul>

## Restrictions on Use of Enumerated Data in for-Loops

### Do not use enumerated data as the loop counter variable in for - loops

To iterate over a range of enumerated data with consecutive values, in the loop counter, cast the enumerated data to a built-in integer type. The size of the built-in integer type must be big enough to contain the enumerated value.

For example, suppose you define an enumerated type `ColorCodes` as follows:

```
classdef(Enumeration) ColorCodes < int32
    enumeration
        Red(1),
        Blue(2),
        Green(3),
        Yellow(4),
        Purple(5)
    end
end
```

Because the enumerated values are consecutive, you can use `ColorCodes` data in a for-loop like this:

```
...
for i = int32(ColorCodes.Red):int32(ColorCodes.Purple)
    c = ColorCodes(i);
    ...
end
```

## Toolbox Functions That Support Enumerated Types for Code Generation

The following MATLAB toolbox functions support enumerated types for code generation:

- `cast`
- `cat`
- `circshift`
- `flipdim`
- `fliplr`
- `flipud`
- `histc`
- `intersect`
- `ipermute`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismember`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `setdiff`
- `setxor`
- `shiftdim`
- `sort`
- `sortrows`

- squeeze
- union
- unique

# Code Generation for MATLAB Classes

---

- “MATLAB Classes Definition for Code Generation” on page 10-2
- “Classes That Support Code Generation” on page 10-7
- “Generate Code for MATLAB Value Classes” on page 10-8
- “Generate Code for MATLAB Handle Classes and System Objects” on page 10-13
- “MATLAB Classes in Code Generation Reports” on page 10-15
- “Troubleshooting Issues with MATLAB Classes” on page 10-18

## MATLAB Classes Definition for Code Generation

To generate efficient standalone code for MATLAB classes, you must use classes differently than when running your code in the MATLAB environment.

What's Different	More Information
Class must be in a single file. Because of this limitation, code generation is not supported for a class definition that uses an @-folder.	"Creating a Single, Self-Contained Class Definition File"
Restricted set of language features.	"Language Limitations" on page 10-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 10-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 10-4
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 10-13
Calls to base class constructor.	"Calls to Base Class Constructor" on page 10-5
Global variables containing MATLAB objects are not supported for code generation.	N/A
Inheritance from built-in MATLAB classes is not supported.	"Inheritance from Built-In MATLAB Classes Not Supported" on page 10-6

### Language Limitations

Although code generation support is provided for common features of classes such as properties and methods, there are a number of advanced features which are not supported, such as:

- Events
- Listeners
- Arrays of objects



- Recursive data structures
  - Linked lists
  - Trees
  - Graphs
- Overloadable operators `subsref`, `subsassign`, and `subsindex`

In MATLAB, classes can define their own versions of the `subsref`, `subsassign`, and `subsindex` methods. Code generation does not support classes that have their own definitions of these methods.

- The `empty` method

In MATLAB, classes have a built-in static method, `empty`, which creates an empty array of the class. Code generation does not support this method.

- The following MATLAB handle class methods:
  - `addlistener`
  - `delete`
  - `eq`
  - `findobj`
  - `findpro`
- The `AbortSet` property attribute

## Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

```
codegen ClassNameA
```

- If an entry-point MATLAB function has an input or output that is a MATLAB class, you cannot generate code for this function.

For example, if function `f00` takes one input, `a`, that is a MATLAB object, you cannot generate code for `f00` by executing:

```
codegen foo -args {a}
```

- Code generation does not support assigning an object of a value class into a nontunable property. For example, `obj.prop=v;` is invalid when `prop` is a nontunable property and `v` is an object based on a value class.
- You cannot use `coder.extrinsic` to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to the `coder.ceval` function.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.
- The `coder.nullcopy` function does not support MATLAB classes as inputs.

## Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you normally would when running your code in the MATLAB environment:

- After defining a property, do not assign it an incompatible type. Do not use a property before attempting to grow it.

When you define class properties for code generation, consider the same factors that you take into account when defining variables. In the MATLAB language, variables can change their class, size, or complexity dynamically at run time so you can use the same variable to hold a value of varying class, size, or complexity. C and C++ use static typing. Before using variables, to determine their type, the code generation software requires a complete assignment to each variable. Similarly, before using properties, you must explicitly define their class, size, and complexity.

- Initial values:
  - If the property does not have an explicit initial value, the code generation software assumes that it is undefined at the beginning of the constructor. The code generation software does not assign an empty matrix as the default.
  - If the property does not have an initial value and the code generation software cannot determine that the property is assigned prior to first use, the software generates a compilation error.
  - For System objects, if a nontunable property is a structure, you must completely assign the structure. You cannot do partial assignment using subscripting.

For example, for a nontunable property, you can use the following assignment:

```
mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
```

You cannot use the following partial assignments:

```
mySystemObject.nonTunableProperty.fieldA = a;
mySystemObject.nonTunableProperty.fieldB = b;
```

- If dynamic memory allocation is enabled, code generation supports variable-size properties for handle classes. Without dynamic memory allocation, you cannot generate code for handle classes that have variable-size properties.
- `coder. varsize` is not supported for class properties.
- MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns `true` (1).

## Calls to Base Class Constructor

If a class constructor contains a call to the constructor of the base class, the call to the base class constructor must come before `for`, `if`, `return`, `switch` or `while` statements.

For example, if you define a class B based on class A:

```
classdef B < A
    methods
        function obj = B(varargin)
            if nargin == 0
                a = 1;
                b = 2;
            elseif nargin == 1
                a = varargin{1};
                b = 1;
            elseif nargin == 2
                a = varargin{1};
                b = varargin{2};
            end
            obj = obj@A(a,b);
        end
    end
end
```

Because the class definition for **B** uses an `if` statement before calling the base class constructor for **A**, you cannot generate code for function `callB`:

```
function [y1,y2] = callB
x = B;
y1 = x.p1;
y2 = x.p2;
end
```

However, you can generate code for `callB` if you define class **B** as:

```
classdef B < A
    methods
        function obj = NewB(varargin)
            [a,b] = getaandb(varargin{:});
            obj = obj@A(a,b);
        end
    end
end
```

```
function [a,b] = getaandb(varargin)
if nargin == 0
    a = 1;
    b = 2;
elseif nargin == 1
    a = varargin{1};
    b = 1;
elseif nargin == 2
    a = varargin{1};
    b = varargin{2};
end
end
```

## Inheritance from Built-In MATLAB Classes Not Supported

You cannot generate code for classes that inherit from built-in MATLAB classes. For example, you cannot generate code for the following class:

```
classdef myclass < double
```

## Classes That Support Code Generation

You can generate code for MATLAB value and handle classes and user-defined System objects. Your class can have multiple methods and properties and can inherit from multiple classes.

To generate code for:	Example:
Value classes	“Generate Code for MATLAB Value Classes” on page 10-8
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 10-13

For more information, see:

- “Classes in the MATLAB Language”
- “MATLAB Classes Definition for Code Generation” on page 10-2

## Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

- 1 In a writable folder, create a MATLAB value class, `Shape`. Save the code as `Shape.m`.

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out = obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
end
methods(Abstract = true)
    getarea(obj);
end
methods(Static)
    function d = distanceBetweenShapes(shape1,shape2)
        xDist = abs(shape1.centerX - shape2.centerX);
        yDist = abs(shape1.centerY - shape2.centerY);
        d = sqrt(xDist^2 + yDist^2);
    end
end
end
```

- 2 In the same folder, create a class, `Square`, that is a subclass of `Shape`. Save the code as `Square.m`.

```
classdef Square < Shape
% Create a Square at coordinates center X and center Y
```

```

% with sides of length of side
properties
    side;
end
methods
    function obj = Square(side,centerX,centerY)
        obj@Shape(centerX,centerY);
        obj.side = side;
    end
    function Area = getarea(obj)
        Area = obj.side^2;
    end
end
end

```

- 3 In the same folder, create a class, Rhombus, that is a subclass of Shape. Save the code as Rhombus.m.

```

classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)
            Area = 0.5*obj.diag1*obj.diag2;
        end
    end
end
end

```

- 4 Write a function that uses this class.

```

function [TotalArea, Distance] = use_shape
    %#codegen
    s = Square(2,1,2);
    r = Rhombus(3,4,7,10);
    TotalArea = s.area + r.area;
    Distance = Shape.distanceBetweenShapes(s,r);

```

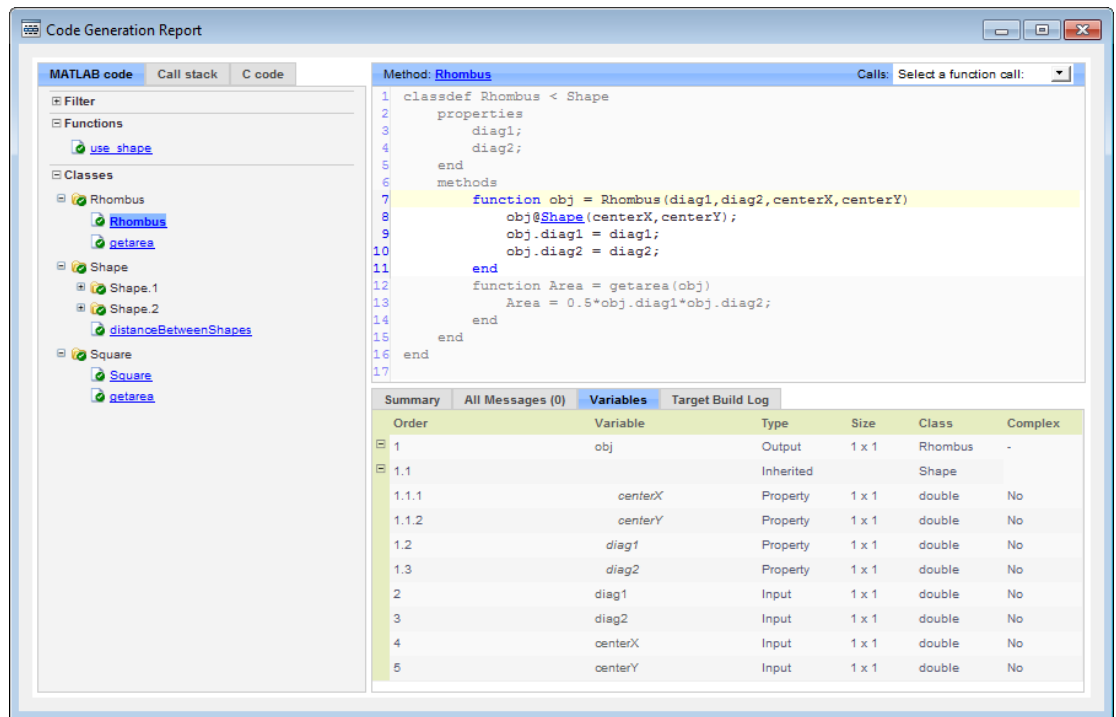
- 5 Generate a static library for use\_shape and generate a code generation report.

codegen -config:lib -report use\_shape

codegen generates a C static library with the default name, use\_shape, and supporting files in the default folder, codegen/lib/use\_shape.

- 6 Click the **View report** link.
- 7 In the report, on the **MATLAB code** tab, click the link to the Rhombus class.

The report displays the class definition of the Rhombus class and highlights the class constructor. On the **Variables** tab, it provides details of the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties. To view the list of properties, expand the list. Within the list of properties, the list of inherited properties is collapsed. In the following report, the lists of properties and inherited properties are expanded.



The screenshot shows the Code Generation Report window with the following content:

**MATLAB code** Call stack C code

Method: Rhombus Calls: Select a function call:

```

1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape(centerX,centerY);
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17

```

**Summary** All Messages (0) **Variables** Target Build Log

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1		Inherited		Shape	
1.1.1	centerX	Property	1 x 1	double	No
1.1.2	centerY	Property	1 x 1	double	No
1.2	diag1	Property	1 x 1	double	No
1.3	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

- 8 At the top right side of the report, expand the **Calls** list.



The **Calls** list shows that there is a call to the Rhombus constructor from `use_shape` and that this constructor calls the Shape constructor.

The screenshot shows the Code Generation Report window with the following components:

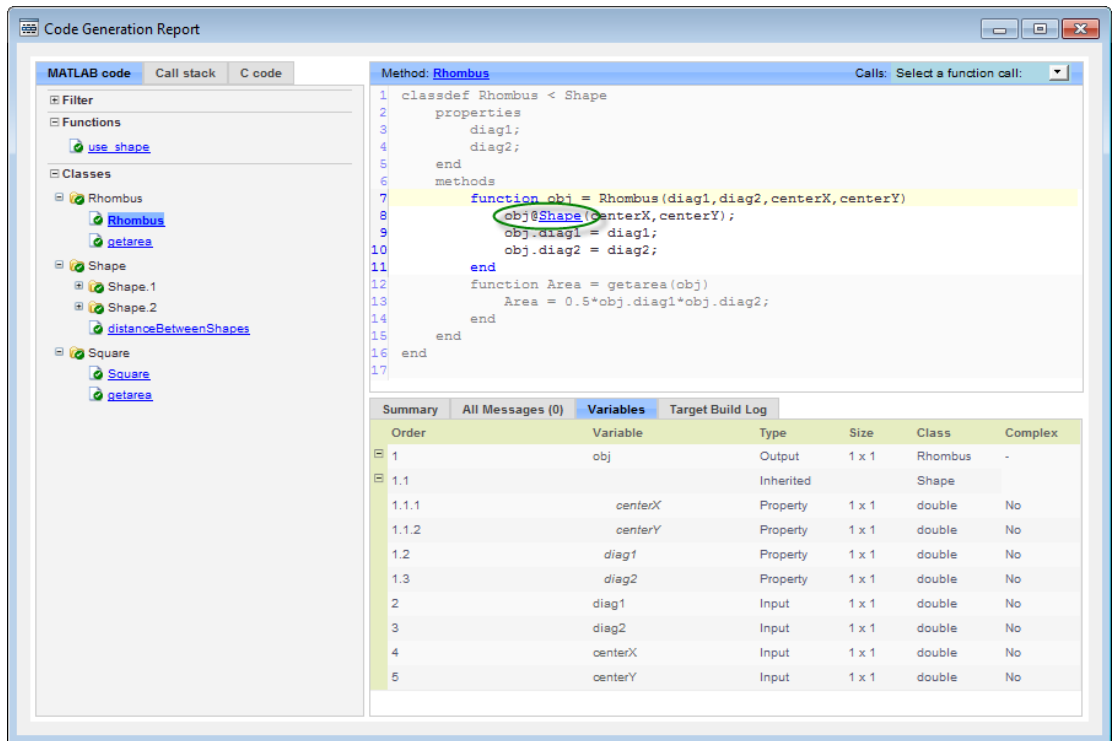
- Left Panel (MATLAB code):** A tree view showing the class hierarchy: Rhombus (with links to Rhombus and getarea), Shape (with links to Shape.1, Shape.2, and distanceBetweenShapes), and Square (with links to Square and getarea).
- Method View:** The `Rhombus` method is selected, showing the following code:
 

```

1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape(centerX,centerY);
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17
```
- Call Stack:** A dropdown menu is open, showing the following calls:
  - Select a function call:
  - Select a function call:
  - Calls to this function
  - From use\_shape line 4
  - Calls from this function
  - From line 8 to Shape
- Summary Table:** A table with columns: Order, Variable, Type, Size, Class, and Complex.
 

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1		Inherited		Shape	
1.1.1	centerX	Property	1 x 1	double	No
1.1.2	centerY	Property	1 x 1	double	No
1.2	diag1	Property	1 x 1	double	No
1.3	diag2	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

- 9 The constructor for the Rhombus class calls the Shape method of the base Shape class: `obj@Shape`. In the report, click the Shape link in this call.



The link takes you to the Shape method in the Shape class definition.

## Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report.

- 1 In a writable folder, create a System object, `AddOne`, which subclasses from `matlab.System`. Save the code as `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access=protected)
        % stepImpl method is called by the step method
        function y = stepImpl(~,x)
            y = x+1;
        end
    end
end
```

- 2 Write a function that uses this System object.

```
function y = testAddOne(x)
%#codegen
    p = AddOne();
    y = p.step(x);
end
```

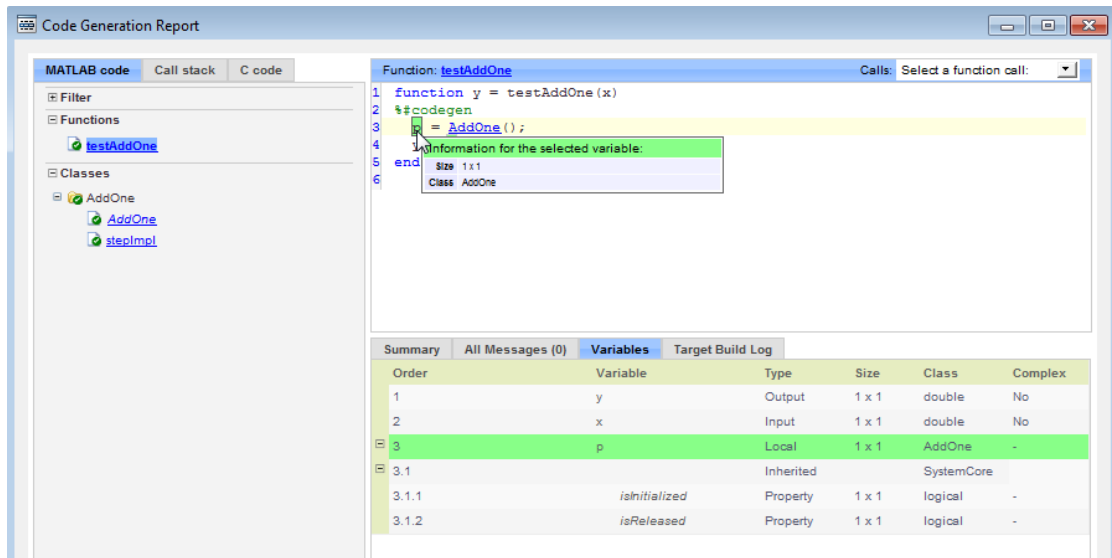
- 3 Generate a MEX function for this code.

```
codegen -report testAddOne -args {0}
```

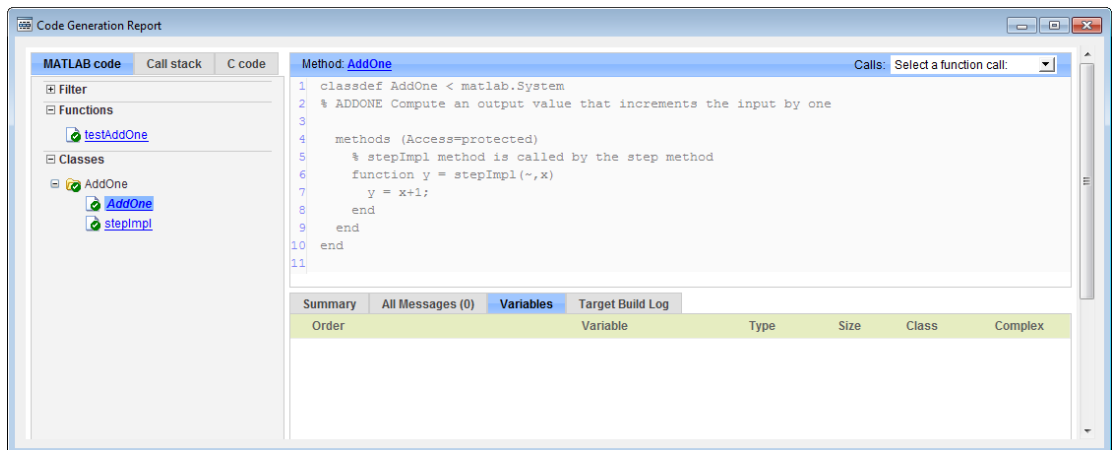
The `-report` option instructs `codegen` to generate a code generation report, even if no errors or warnings occur. The `-args` option specifies that the `testAddOne` function takes one scalar double input.

```
>> codegen -report testAddOne -args {0}
Code generation successful: View report
```

- 4 Click the **View report** link.
- 5 In the report, on the **MATLAB Code** tab **Functions** panel, click `testAddOne`, then click the **Variables** tab. You can view information about the variable `p` on this tab.



6 To view the class definition, on the **Classes** panel, click **AddOne**.



# MATLAB Classes in Code Generation Reports

## What Reports Tell You About Classes

Code generation reports:

- Provide a hierarchical tree of the classes used in your MATLAB code.
- Display a list of methods for each class in the MATLAB code tab.
- Display the objects used in your MATLAB code together with their properties on the **Variables** tab.
- Provide a filter so that you can sort methods by class, size, and complexity.
- List the set of calls from and to the selected method in the **Calls** list.

## How Classes Appear in Code Generation Reports

### In the MATLAB Code Tab

The report displays an alphabetical hierarchical list of the classes used in the your MATLAB code. For each class, you can:

- Expand the class information to view the class methods.
- View a class method by clicking its name. The report displays the methods in the context of the full class definition.
- Filter the methods by size, complexity, and class by using the **Filter functions and methods** option.

### Default Constructors

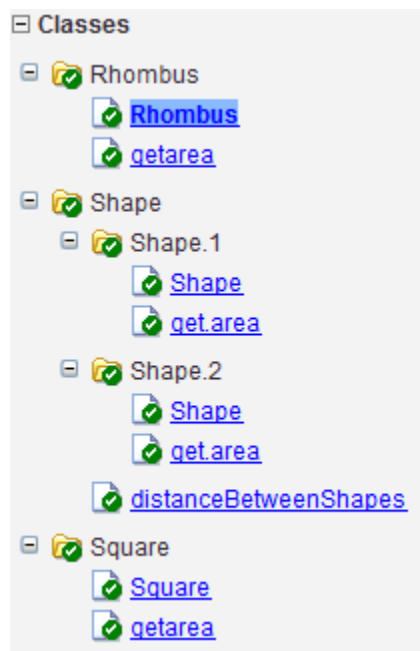
If a class has a default constructor, the report displays the constructor in italics.

### Specializations

If the same class is specialized into multiple different classes, the report differentiates the specializations by grouping each one under a single node in the tree. The report associates the class definition functions and static methods with the primary node. It associates the instance-specific methods with the corresponding specialized node.

For example, consider a base class, **Shape** that has two specialized subclasses, **Rhombus** and **Square**. The **Shape** class has an abstract method, `getarea`, and a static method, `distanceBetweenShapes`. The code generation report, displays a

node for the specialized `Rhombus` and `Square` classes with their constructors and `getarea` method. It displays a node for the `Shape` class and its associated static method, `distanceBetweenShapes`, and two instances of the `Shape` class, `Shape1` and `Shape2`.



### Packages

If you define classes as part of a package, the report displays the package in the list of classes. You can expand the package to view the classes that it contains. For more information about packages, see “Packages Create Namespaces”.

### In the Variables Tab

The report displays the objects in the selected function or class. By default, for classes that have properties, the list of properties is collapsed. To expand the list, click the + symbol next to the object name. Within the list of properties, the list of inherited properties is collapsed. To expand the list of inherited properties, click the + symbol next to `Inherited`.

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable `obj1` that is a MATLAB object

with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

### **In the Call Stack**

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

## **How to Generate a Code Generation Report**

Add the `-report` option to your `codegen` command (requires a MATLAB Coder license)

## Troubleshooting Issues with MATLAB Classes

### Class *cClass* does not have a property with name *name*

If a MATLAB class has a method, `mymethod`, that returns a handle class with a property, `myprop`, you cannot generate code for the following type of assignment:

```
obj.mymethod().myprop=...
```

For example, consider the following classes:

```
classdef MyClass < handle
    properties
        myprop
    end
    methods
        function this = MyClass
            this.myprop = MyClass2;
        end
        function y = mymethod(this)
            y = this.myprop;
        end
    end
end
```

```
classdef MyClass2 < handle
    properties
        aa
    end
end
```

You cannot generate code for function `foo`.

```
function foo
```

```
h = MyClass;
```

```
h.mymethod().aa = 12;
```

In this function, `h.mymethod()` returns a handle object of type `MyClass2`. In MATLAB, the assignment `h.mymethod().aa = 12;` changes the property of that object. Code generation does not support this assignment.



**Workaround**

Rewrite the code to return the object and then assign a value to a property of the object.

```
function foo  
  
h = MyClass;  
  
b=h.mymethod();  
b.aa=12;
```



# Code Generation for Function Handles

---

- “Function Handle Definition for Code Generation” on page 11-2
- “Define and Pass Function Handles for Code Generation” on page 11-3
- “Function Handle Limitations for Code Generation” on page 11-5

## Function Handle Definition for Code Generation

You can use function handles to invoke functions indirectly and parameterize operations that you repeat frequently. You can perform the following operations with function handles:

- Define handles that reference user-defined functions and built-in functions supported for code generation (see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List”)

---

**Note:** You cannot define handles that reference extrinsic MATLAB functions.

---

- Define function handles as scalar values
- Define structures that contain function handles
- Pass function handles as arguments to other functions (excluding extrinsic functions)

To generate efficient standalone code for enumerated data, you are restricted to using a subset of the operations you can perform with function handles in MATLAB, as described in “Function Handle Limitations for Code Generation” on page 11-5

## Define and Pass Function Handles for Code Generation

The following code example shows how to define and call function handles for code generation. You can copy the example to a MATLAB Function block in Simulink or MATLAB function in Stateflow. To convert this function to a MEX function using `codegen`, uncomment the two calls to the `assert` function, highlighted below:

```
function addval(m)
%#codegen

% Define class and size of primary input m
% Uncomment next two lines to build MEX function with codegen
% assert(isa(m,'double'));
% assert(all (size(m) == [3 3]));

% Pass function handle to addone
% to add one to each element of m
m = map(@addone, m);
disp(m);

% Pass function handle to addtwo
% to add two to each element of m
m = map(@addtwo, m);
disp(m);

function y = map(f,m)
    y = m;
    for i = 1:numel(y)
        y(i) = f(y(i));
    end

function y = addone(u)
    y = u + 1;

function y = addtwo(u)
    y = u + 2;
```

This code passes function handles `@addone` and `@addtwo` to the function `map` which increments each element of the matrix `m` by the amount prescribed by the referenced function. Note that `map` stores the function handle in the input variable `f` and then uses `f` to invoke the function — in this case `addone` first and then `addtwo`.

If you have MATLAB Coder, you can use the function `codegen` to convert the function `addval` to a MEX executable that you can run in MATLAB. Follow these steps:

- 1 At the MATLAB command prompt, issue this command:

```
codegen addval
```

- 2 Define and initialize a 3-by-3 matrix by typing a command like this at the MATLAB prompt:

```
m = zeros(3)
```

- 3 Execute the function by typing this command:

```
addval(m)
```

You should see the following result:

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

1    1    1
1    1    1
1    1    1

3    3    3
3    3    3
3    3    3
```

For more information, see “MEX Function Generation at the Command Line”.

## Function Handle Limitations for Code Generation

### You cannot use the same bound variable to reference different function handles.

After you bind a variable to a specific function, you cannot use the same variable to reference two different function handles, as in this example:

```
%Incorrect code
...
x = @plus;
x = @minus;
...
```

This code produces a compilation error.

### You cannot pass function handles to or from `coder.ceval`.

You cannot pass function handles as inputs to or outputs from `coder.ceval`. For example, suppose that `f` and `str.f` are function handles:

```
f = @sin;
str.x = pi;
str.f = f;
```

The following statements result in compilation errors:

```
coder.ceval('foo', @sin);
coder.ceval('foo', f);
coder.ceval('foo', str);
```

### You cannot pass function handles to or from extrinsic functions.

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions. For more information, see “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12.

## You cannot pass function handles to or from primary functions.

You cannot pass function handles as inputs to or outputs from primary functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle,'function_handle') && isa(data,'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as primary inputs. `plotFcn` attempts to call the function referenced by the `fhandle` with the input `data` and plot the results. However, this code generates a compilation error. The error indicates that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of primary inputs.



# Defining Functions for Code Generation

---

- “Specify Variable Numbers of Arguments” on page 12-2
- “Supported Index Expressions” on page 12-3
- “Apply Operations to a Variable Number of Arguments” on page 12-4
- “Implement Wrapper Functions” on page 12-6
- “Pass Property/Value Pairs” on page 12-7
- “Variable Length Argument Lists for Code Generation” on page 12-9

## Specify Variable Numbers of Arguments

You can use `varargin` in a function definition to specify that the function accepts a variable number of input arguments for a given input argument. You can use `varargout` in a function definition to specify that the function returns a variable number of arguments for a given output argument.

When you use `varargin` and `varargout` for code generation, there are the following limitations:

- You cannot use `varargout` in the function definition for a top-level function.
- You cannot use `varargin` in the function definition for a top-level function in a MATLAB Function block in a Simulink model, or in a MATLAB function in a Stateflow diagram.
- If you use `varargin` to define an argument to a top-level function, the code generation software generates the function with a fixed number of arguments. This fixed number of arguments is based on the number of example arguments that you provide on the command line or in a MATLAB Coder project test file.

Common applications of `varargin` and `varargout` for code generation are to:

- “Apply Operations to a Variable Number of Arguments” on page 12-4
- “Implement Wrapper Functions” on page 12-6
- “Pass Property/Value Pairs” on page 12-7

Code generation relies on loop unrolling to produce simple and efficient code for `varargin` and `varargout`. This technique permits most common uses of `varargin` and `varargout`, but some uses are not allowed (see “Variable Length Argument Lists for Code Generation” on page 12-9).

For more information about using `varargin` and `varargout` in MATLAB functions, see “Passing Variable Numbers of Arguments”.

## Supported Index Expressions

In MATLAB, `varargin` and `varargout` are cell arrays. Generated code does not support cell arrays, but does allow you to use the most common syntax — curly braces `{}` — for indexing into `varargin` and `varargout` arrays, as in this example:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i};
end

```

You can use the following index expressions. The *exp* arguments must be constant expressions or depend on a loop index variable.

Expression		Description
<code>varargin</code> ( <i>read only</i> )	<code>varargin{exp}</code>	Read the value of element <i>exp</i>
	<code>varargin{exp1:exp2}</code>	Read the values of elements <i>exp1</i> through <i>exp2</i>
	<code>varargin{:}</code>	Read the values of all elements
<code>varargout</code> ( <i>read and write</i> )	<code>varargout{exp}</code>	Read or write the value of element <i>exp</i>

**Note:** The use of `()` is not supported for indexing into `varargin` and `varargout` arrays.

## Apply Operations to a Variable Number of Arguments

You can use `varargin` and `varargout` in `for`-loops to apply operations to a variable number of arguments. To index into `varargin` and `varargout` arrays in generated code, the value of the loop index variable must be known at compile time. Therefore, during code generation, the compiler attempts to automatically unroll these `for`-loops. Unrolling eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant. For example, the following function automatically unrolls its `for`-loop in the generated code:

```
 %#codegen
function [cmLen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmLen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

## When to Force Loop Unrolling

To automatically unroll `for`-loops containing `varargin` and `varargout` expressions, the relationship between the loop index expression and the index variable must be determined at compile time.

In the following example, the function `fcn` cannot detect a logical relationship between the index expression `j` and the index variable `i`:

```
 %#codegen
function [x,y,z] = fcn(a,b,c)

[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;
for i = 1:length(varargin)
    j = j+1;
    varargout{j} = varargin{j};
end
```

As a result, the function does not unroll the loop and generates a compilation error:

Nonconstant expression or empty matrix.  
 This expression must be constant because  
 its value determines the size or class of some expression.

To fix the problem, you can force loop unrolling by wrapping the loop header in the function `coder.unroll`, as follows:

```

%#codegen
function [x,y,z] = fcn(a,b,c)
    [x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
    j = 0;
    for i = coder.unroll(1:length(varargin))
        j = j + 1;
        varargout{j} = varargin{j};
    end;
  
```

## Using Variable Numbers of Arguments in a for-Loop

The following example multiplies a variable number of input dimensions in inches by 2.54 to convert them to centimeters:

```

%#codegen
function [cmlen,cmwth,cmhgt] = conv_2_metric(inlen,inwth,inhgt)

[cmlen,cmwth,cmhgt] = inch_2_cm(inlen,inwth,inhgt);

function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
  
```

### Key Points About the Example

- `varargin` and `varargout` appear in the local function `inch_2_cm`, not in the top-level function `conv_2_metric`.
- The index into `varargin` and `varargout` is a for-loop variable

For more information, see “Variable Length Argument Lists for Code Generation” on page 12-9.

## Implement Wrapper Functions

You can use `varargin` and `varargout` to write wrapper functions that accept up to 64 inputs and pass them directly to another function.

### Passing Variable Numbers of Arguments from One Function to Another

The following example passes a variable number of inputs to different optimization functions, based on a specified input method:

```
%#codegen
function answer = fcn(method,a,b,c)
answer = optimize(method,a,b,c);

function answer = optimize(method,varargin)
    if strcmp(method,'simple')
        answer = simple_optimization(varargin{:});
    else
        answer = complex_optimization(varargin{:});
    end
    ...
end
```

#### Key Points About the Example

- You can use `{:}` to read all elements of `varargin` and pass them to another function.
- You can mix variable and fixed numbers of arguments.

For more information, see “Variable Length Argument Lists for Code Generation” on page 12-9.

## Pass Property/Value Pairs

You can use `varargin` to pass property/value pairs in functions. However, for code generation, you must take precautions to avoid type mismatch errors when evaluating `varargin` array elements in a `for`-loop:

If	Do This:
You assign <code>varargin</code> array elements to local variables in the <code>for</code> -loop	Verify that for all pairs, the size, type, and complexity are the same for each property and the same for each value
Properties or values have different sizes, types, or complexity	Do not assign <code>varargin</code> array elements to local variables in a <code>for</code> -loop; reference the elements directly

For example, in the following function `test1`, the sizes of the property strings and numeric values are not the same in each pair:

```

%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        name = varargin{i};
        value = varargin{i+1};
        switch name
            case 'size'
                v = set_size(v, value);
            case 'rgb'
                v = set_color(v, value);
            otherwise
            end
        end
    end
end
...

```

Generated code determines the size, type, and complexity of a local variable based on its first assignment. In this example, the first assignments occur in the first iteration of the `for`-loop:

- Defines local variable `name` with size equal to 4
- Defines local variable `value` with a size of scalar

However, in the second iteration, the size of the property string changes to 3 and the size of the numeric value changes to a vector, resulting in a type mismatch error. To avoid such errors, reference `varargin` array values directly, not through local variables, as highlighted in this code:

```

%#codegen
function test1
    v = create_value('size', 18, 'rgb', [240 9 44]);
end

function v = create_value(varargin)
    v = new_value();
    for i = 1 : 2 : length(varargin)
        switch varargin{i}
            case 'size'
                v = set_size(v, varargin{i+1});
            case 'rgb'
                v = set_color(v, varargin{i+1});
            otherwise
                end
        end
    end
end
...

```



## Variable Length Argument Lists for Code Generation

### Use variable length argument lists in top-level functions according to guidelines

When you use `varargin` and `varargout` for code generation, there are the following limitations:

- You cannot use `varargout` in the function definition for a top-level function.
- You cannot use `varargin` in the function definition for a top-level function in a MATLAB Function block in a Simulink model, or in a MATLAB function in a Stateflow diagram.
- If you use `varargin` to define an argument to a top-level function, the code generation software generates the function with a fixed number of arguments. This fixed number of arguments is based on the number of example arguments that you provide on the command line or in a MATLAB Coder project test file.

A *top-level function* is:

- The function called by Simulink in a MATLAB Function block or by Stateflow in a MATLAB function.
- The function that you provide on the command line to `codegen` or `fiaccl`.

For example, the following code generates compilation errors:

```
 %#codegen
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

To fix the problem, write a top-level function that specifies a fixed number of inputs and outputs. Then call `inch_2_cm` as an external function or local function, as in this example:

```
 %#codegen
function [cmL, cmW, cmH] = conv_2_metric(inL, inW, inH)
[cmL, cmW, cmH] = inch_2_cm(inL, inW, inH);
```

```
function varargout = inch_2_cm(varargin)
for i = 1:length(varargin)
    varargout{i} = varargin{i} * 2.54;
end
```

## Use curly braces {} to index into the argument list

For code generation, you can use curly braces {}, but not parentheses (), to index into `varargin` and `varargout` arrays. For more information, see “Supported Index Expressions” on page 12-3.

## Verify that indices can be computed at compile time

If you use an expression to index into `varargin` or `varargout`, make sure that the value of the expression can be computed at compile time. For examples, see “Apply Operations to a Variable Number of Arguments” on page 12-4.

## Do not write to varargin

Generated code treats `varargin` as a read-only variable. If you want to write to input arguments, copy the values into a local variable.

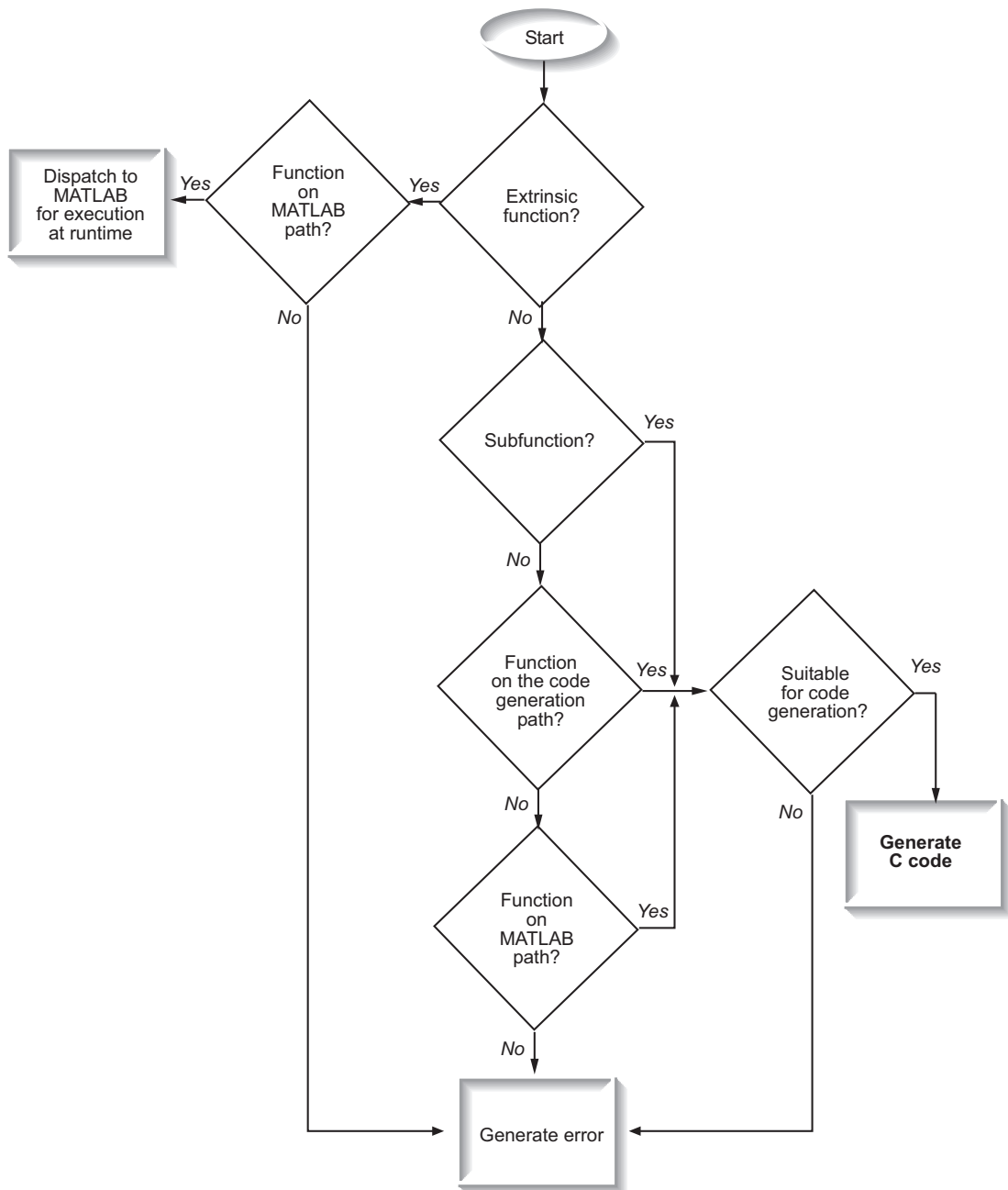
# Calling Functions for Code Generation

---

- “Resolution of Function Calls for Code Generation” on page 13-2
- “Resolution of File Types on Code Generation Path” on page 13-6
- “Compilation Directive `%#codegen`” on page 13-8
- “Call Local Functions” on page 13-9
- “Call Supported Toolbox Functions” on page 13-10
- “Call MATLAB Functions” on page 13-11

### Resolution of Function Calls for Code Generation

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:



## Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path

See “Compile Path Search Order” on page 13-4.

- Attempts to compile functions unless the code generation software determines that it should not compile them or you explicitly declare them to be extrinsic.

If a MATLAB function is not supported for code generation, you can declare it to be extrinsic by using the construct `coder.extrinsic`, as described in “Declaring MATLAB Functions as Extrinsic Functions”. During simulation, the code generation software generates code for the call to an extrinsic function, but does not generate the function's internal code. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, compilation errors occur.

The code generation software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using the `coder.extrinsic` function.

- Resolves file type based on precedence rules described in “Resolution of File Types on Code Generation Path” on page 13-6

## Compile Path Search Order

During code generation, function calls are resolved on two paths:

### 1 Code generation path

MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.

### 2 MATLAB path

If the function is not on the code generation path, MATLAB searches this path.

MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order”).

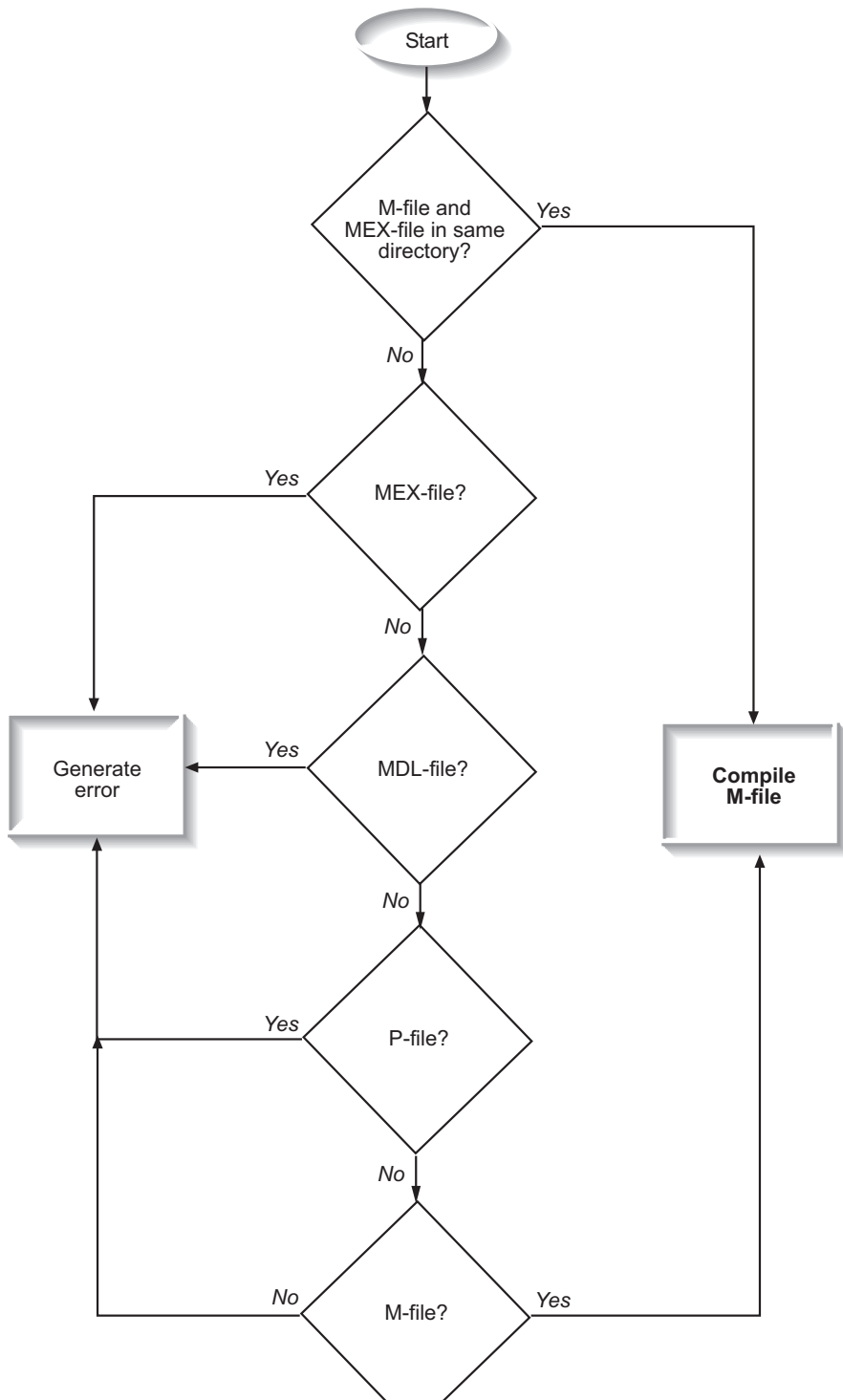
## **When to Use the Code Generation Path**

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

## **Resolution of File Types on Code Generation Path**

MATLAB uses the following precedence rules for code generation:





### Compilation Directive `%#codegen`

Add the `%#codegen` directive (or pragma) to your function after the function signature to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB code analyzer to help you diagnose and fix violations that would result in errors during code generation.

```
function y = my_fcn(x) %#codegen
```

```
.....
```

## Call Local Functions

Local functions are functions defined in the body of MATLAB function. They work the same way for code generation as they do when executing your algorithm in the MATLAB environment.

The following example illustrates how to define and call a local function `mean`:

```
function [mean, stdev] = stats(vals)
%#codegen

% Calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals, '-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

### Call Supported Toolbox Functions

You can call toolbox functions directly if they are supported for code generation. For a list of supported functions, see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List” on page 4-2.

## Call MATLAB Functions

The code generation software attempts to generate code for functions, even if they are not supported for C code generation. The software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using `coder.extrinsic`. During simulation, the code generation software generates code for these functions, but does not generate their internal code. During standalone code generation, MATLAB attempts to determine whether the visualization function affects the output of the function in which it is called. Provided that the output does not change, MATLAB proceeds with code generation, but excludes the visualization function from the generated code. Otherwise, compilation errors occur.

For example, you might want to call `plot` to visualize your results in the MATLAB environment. If you generate a MEX function from a function that calls `plot` and then run the generated MEX function, the code generation software dispatches calls to the `plot` function to MATLAB. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.

The screenshot shows the MATLAB code editor with the 'MATLAB code' tab selected. On the left, a 'Filter' pane shows 'Functions' with 'stats' and 'stats > avg' listed. The main editor displays the following code:

```

Function: stats
Calls: Select a function call:

1 function [mean, stdev] = stats(vals)
2 %#codegen
3
4 % Calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 len = length(vals);
8 mean = avg(vals, len);
9 stdev = sqrt(sum((vals-avg(vals, len)).^2)/len);
10 plot(vals, '-+');
11

```

A tooltip points to the `plot` function call on line 10, stating: "Only supported within the MATLAB environment."

For unsupported functions other than common visualization functions, you must declare the functions (like `pause`) to be extrinsic (see “Resolution of Function Calls for Code Generation” on page 13-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 13-16).

There are two ways to declare a function to be extrinsic:

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 13-12).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 13-16).

## Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

```
coder.extrinsic('function_name_1', ... , 'function_name_n');
```

### Declaring Extrinsic Functions

The following code declares the MATLAB `patch` function extrinsic in the local function `create_plot`:

```
function c = pythagoras(a,b,color) %#codegen
% Calculates the hypotenuse of a right triangle
% and displays the triangle.
```

```
c = sqrt(a^2 + b^2);
create_plot(a, b, color);
```

```
function create_plot(a, b, color)
%Declare patch and axis as extrinsic
```

```
coder.extrinsic('patch');
```

```
x = [0;a;a];
y = [0;0;b];
patch(x, y, color);
axis('equal');
```

The code generation software detects that `axis` is not supported for code generation and automatically treats it as an extrinsic function. The compiler does not generate code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

To test the function, follow these steps:

- 1 Convert `pythagoras` to a MEX function by executing this command at the MATLAB prompt:

```
codegen -report pythagoras -args {1, 1, [.3 .3 .3]}
```

- Click the link to the code generation report and then, in the report, view the MATLAB code for `create_plot`.

The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.

The screenshot shows the MATLAB code generation report for the `create_plot` function. The code is as follows:

```

7
8
9 function create_plot(a, b, color)
10 %Declare patch and axis as extrinsic
11
12 coder.extrinsic('patch'); % , 'axis');
13
14 x = [0;a;a];
15 y = [0;0;b];
16 patch(x, y, color);
17 axis('equal');

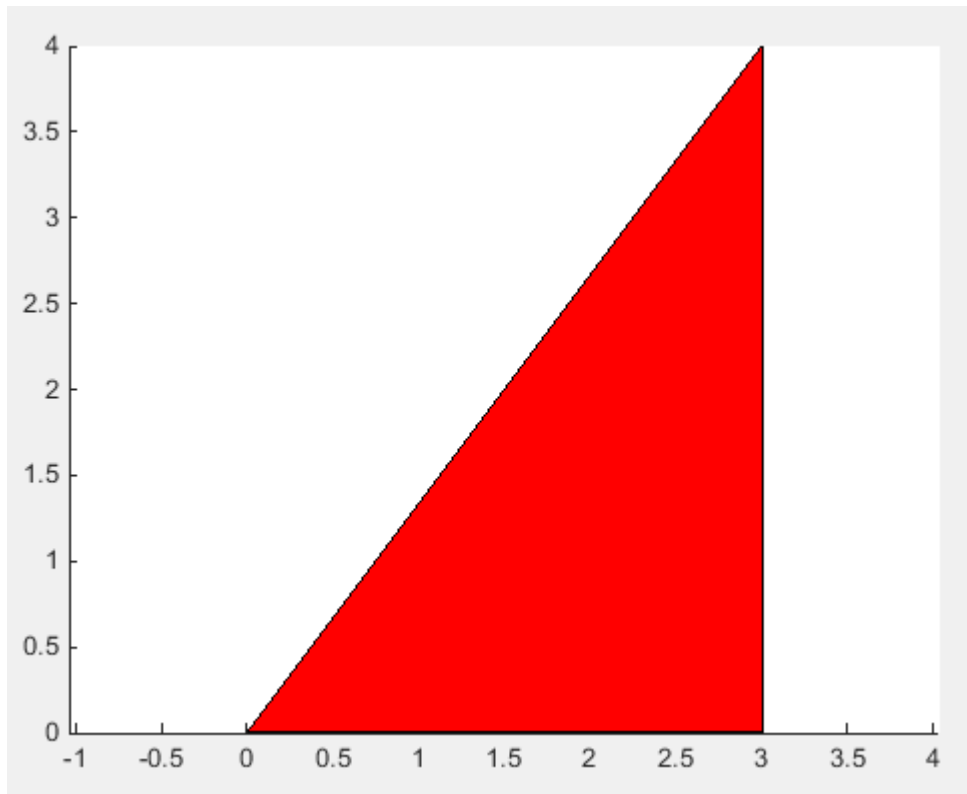
```

A tooltip points to the `axis('equal')` call, with the text: "Only supported within the MATLAB environment."

- Run the MEX function by executing this command:

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays a plot of the right triangle as a red patch object:



### When to Use the `coder.extrinsic` Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output — such as `pause` — during simulation, without generating unnecessary code (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 13-16).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with `mxArrays`” on page 13-17).



- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see “Scope of Extrinsic Function Declarations” on page 13-15).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 13-15). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 13-16).

### Rules for Extrinsic Function Declarations

Observe the following rules when declaring functions extrinsic for code generation:

- Declare the function extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.

### Scope of Extrinsic Function Declarations

The `coder.extrinsic` construct has function scope. For example, consider the following code:

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

In this example, `rat` and `min` are treated as extrinsic every time they are called in the main function `foo`. There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a local function, as in this example:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = mymin(N, D);

function y = mymin(a,b)
coder.extrinsic('min');
y = min(a,b);
```

Here, the function `rat` is extrinsic every time it is called inside the main function `foo`, but the function `min` is extrinsic only when called inside the local function `mymin`.

- Call the MATLAB function using `feval`, as described in “Calling MATLAB Functions Using `feval`” on page 13-16.

### Calling MATLAB Functions Using `feval`

The function `feval` is automatically interpreted as an extrinsic function during code generation. Therefore, you can use `feval` to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

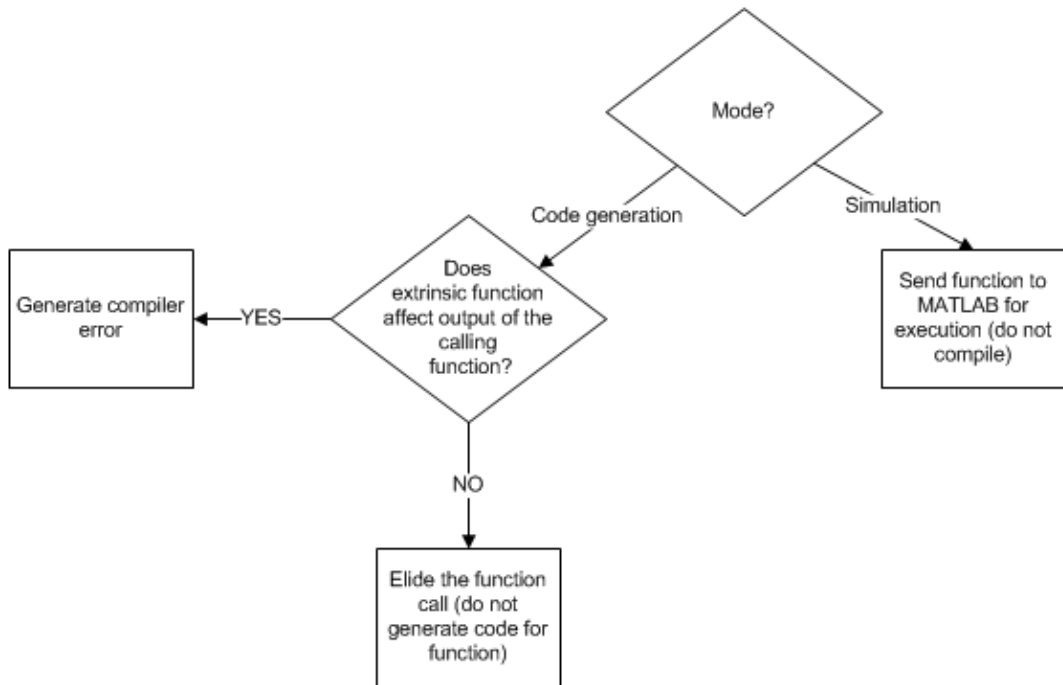
Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because `feval` is extrinsic, the statement `feval('min', N, D)` is evaluated by MATLAB — not compiled — which has the same result as declaring the function `min` extrinsic for just this one call. By contrast, the function `rat` is extrinsic throughout the function `foo`.

### How MATLAB Resolves Extrinsic Functions During Simulation

MATLAB resolves calls to extrinsic functions — functions that do not support code generation — as follows:



During simulation, MATLAB generates code for the call to an extrinsic function, but does not generate the function's internal code. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, MATLAB attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see “Working with `mxArrays`” on page 13-17). Provided that the output does not change, MATLAB proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, MATLAB issues a compiler error.

## Working with `mxArrays`

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables

- Passing `mxArrays` to functions and returning them from functions
- Converting `mxArrays` to known types at run time

To use `mxArrays` returned by extrinsic functions in other operations, you must first convert them to known types, as described in “Converting `mxArrays` to Known Types” on page 13-18.

### Converting `mxArrays` to Known Types

To convert an `mxArray` to a known type, assign the `mxArray` to a variable whose type is defined. At run time, the `mxArray` is converted to the type of the variable assigned to it. However, if the data in the `mxArray` is not consistent with the type of the variable, you get a run-time error.

For example, consider this code:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = min(N, D);
```

Here, the top-level function `foo` calls the extrinsic MATLAB function `rat`, which returns two `mxArrays` representing the numerator `N` and denominator `D` of the rational fraction approximation of `pi`. Although you can pass these `mxArrays` to another MATLAB function — in this case, `min` — you cannot assign the `mxArray` returned by `min` to the output `y`.

If you run this function `foo` in a MATLAB Function block in a Simulink model, the code generates the following error during simulation:

```
Function output 'y' cannot be of MATLAB type.
```

To fix this problem, define `y` to be the type and size of the value that you expect `min` to return — in this case, a scalar double — as follows:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```

## Restrictions on Extrinsic Functions for Code Generation

The full MATLAB run-time environment is not supported during code generation. Therefore, the following restrictions apply when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller, or read or write to the caller's workspace do not work during code generation. Such functions include:
  - `dbstack`
  - `evalin`
  - `assignin`
  - `save`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code may produce unpredictable results if your extrinsic function performs the following actions at run time:
  - Change folders
  - Change the MATLAB path
  - Delete or add MATLAB files
  - Change warning states
  - Change MATLAB preferences
  - Change Simulink parameters

## Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.



# Fixed-Point Conversion

---

- “Convert MATLAB Code to Fixed-Point C Code” on page 14-3
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-4
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-18
- “Type Proposal Settings” on page 14-34
- “Detect Overflows” on page 14-38
- “Replace the exp Function with a Lookup Table” on page 14-45
- “Replace a Custom Function with a Lookup Table” on page 14-51
- “Enable Plotting Using the Simulation Data Inspector” on page 14-58
- “Log Data for Histogram” on page 14-59
- “View and Modify Variable Information” on page 14-62
- “Build Instrumented MEX Function” on page 14-66
- “Propose Fixed-Point Data Types” on page 14-67
- “Apply Fixed-Point Data Types” on page 14-77
- “Modify Data Type Proposal Settings” on page 14-82
- “Modify Instrumentation Report Settings” on page 14-85
- “Automated Fixed-Point Conversion” on page 14-86
- “Instrumented MEX Functions” on page 14-105
- “Convert Fixed-Point Conversion Project to MATLAB Scripts” on page 14-107
- “Generated Fixed-Point Code” on page 14-110
- “Fixed-Point Code for MATLAB Classes” on page 14-116
- “Automated Fixed-Point Conversion Best Practices” on page 14-119
- “Replacing Functions Using Lookup Table Approximations” on page 14-128
- “MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 14-129
- “Inspecting Data Using the Simulation Data Inspector” on page 14-131

- “Custom Plot Functions” on page 14-134
- “Data Type Issues in Generated Code” on page 14-136



## Convert MATLAB Code to Fixed-Point C Code

- 1** Create a MATLAB Coder project, add the entry-point function from which you want to generate code, and then define entry-point input types.
- 2** On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**. Then on the project **Fixed-Point Conversion** pane, click **Define and validate fixed-point types** to open the Fixed-Point Conversion tool.
- 3** Compute ranges by either simulating using a test file, using static analysis to derive ranges from design ranges, or both.
- 4** Validate the proposed data types. See “Validating Types” on page 14-103.
- 5** Test numerics. See “Testing Numerics” on page 14-103.
- 6** In the MATLAB Coder project, select the **Build** tab, set the **Output type** to build a static or dynamic library, or executable, and then click **Build**.

MATLAB Coder generates fixed-point C code for your entry-point MATLAB function.

For more information, see “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-4 and “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-18.

## Propose Fixed-Point Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:
 

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```
- 3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>fun_with_matlab.m</code>	Entry-point MATLAB function
Test file	<code>fun_with_matlab_test.m</code>	MATLAB script that tests <code>fun_with_matlab.m</code>

## The `fun_with_matlab` Function

```
function y = fun_with_matlab(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, 0.333333333333333];

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
end

```

## The fun\_with\_matlab\_test Script

The test script runs the fun\_with\_matlab function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```

% fun_with_matlab_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
    y(i,:) = fun_with_matlab(x(i,:));
end

% Plot the results
titles = {'Chirp', 'Step', 'Impulse'}
clf

```

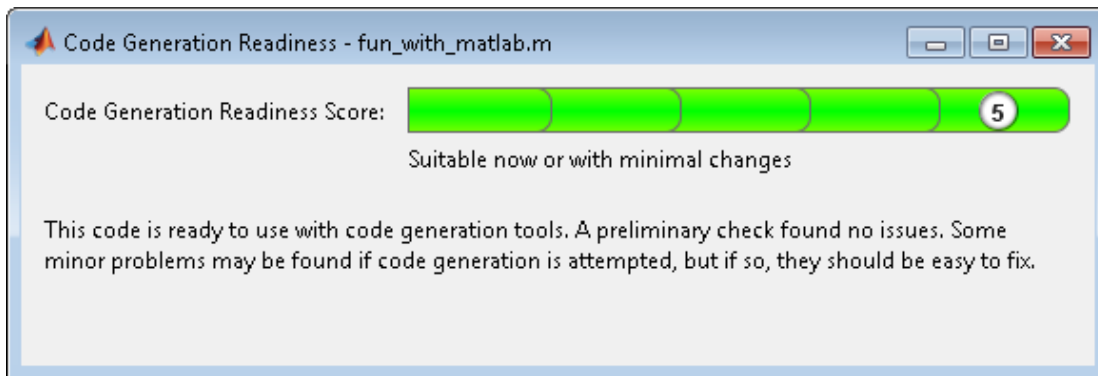
```
for i = 1:size(x,1)
    subplot(size(x,1),1,i)
    plot(t,x(i,:),t,y(i,:))
    title(titles{i})
    legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

### Check Code Generation Readiness

In the current working folder, right-click the `fun_with_matlab.m` function. From the context menu, select **Check Code Generation Readiness**.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `fun_with_matlab.m` function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis”.

### Create and set up a MATLAB Coder Project

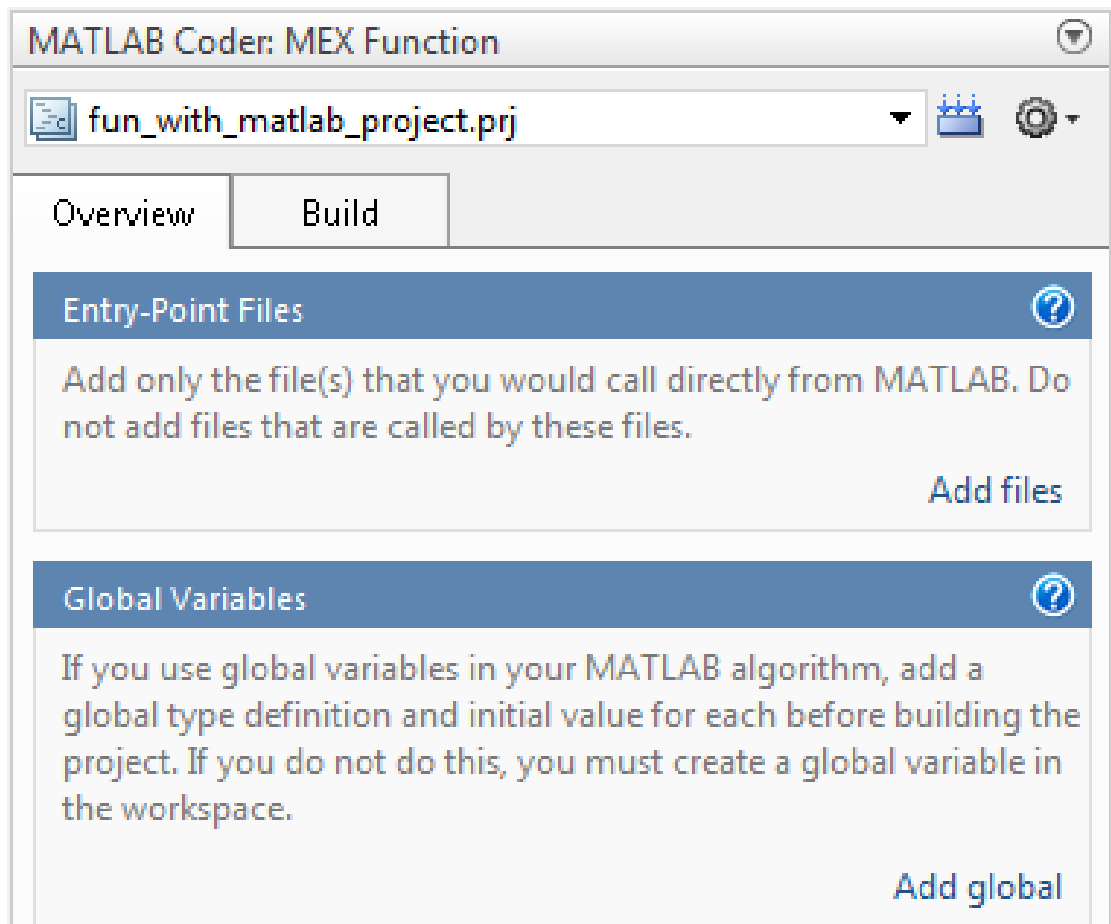
- 1 Navigate to the work folder that contains the file for this example.

- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `fun_with_matlab_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_matlab_project.prj
```

By default, the project opens in the MATLAB workspace.

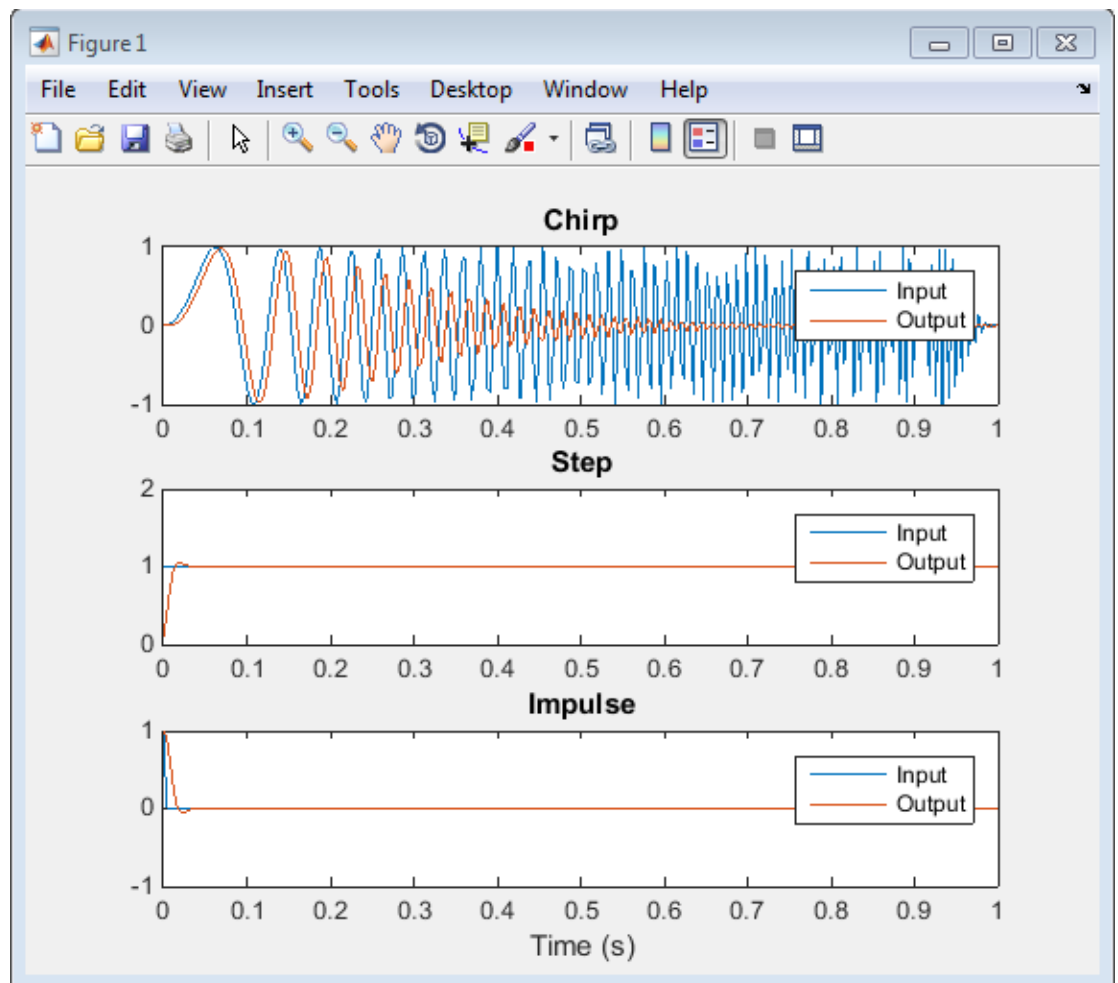


- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

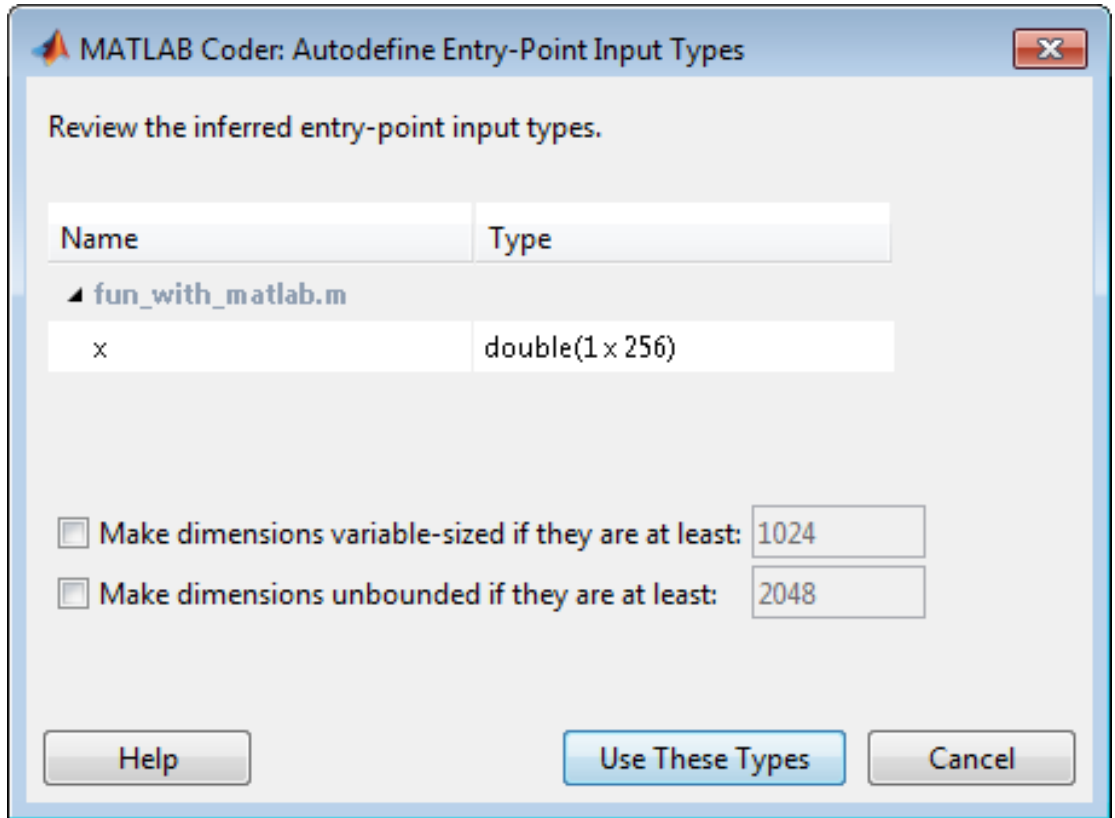
### Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `fun_with_matlab_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



MATLAB Coder determines the input types from the test file and then displays them.

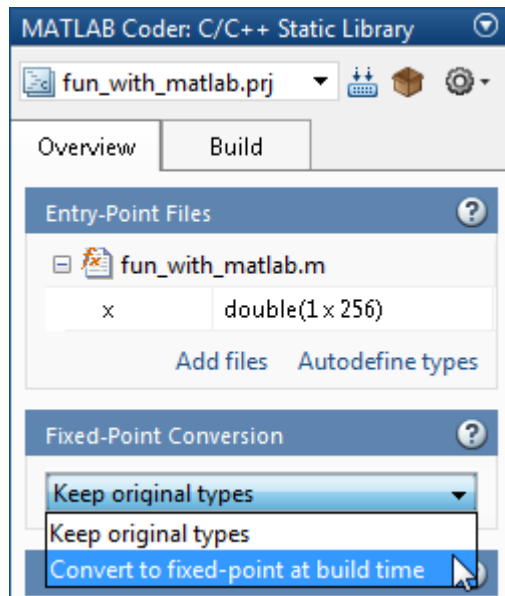


- 3 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x256)`.

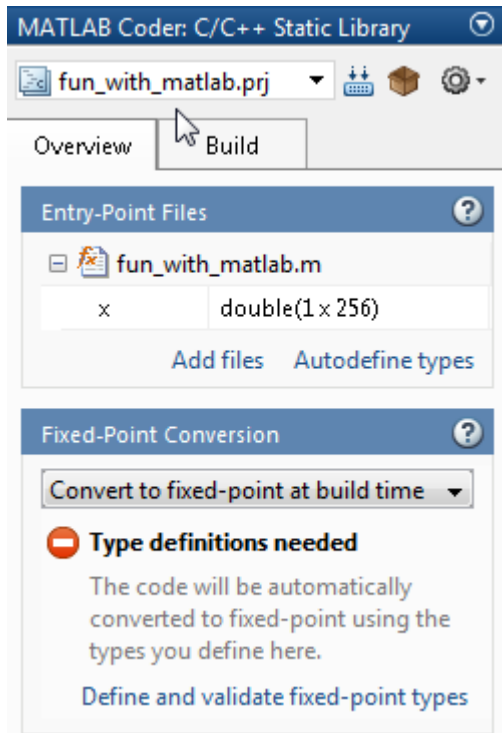
### Fixed-Point Conversion

- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.



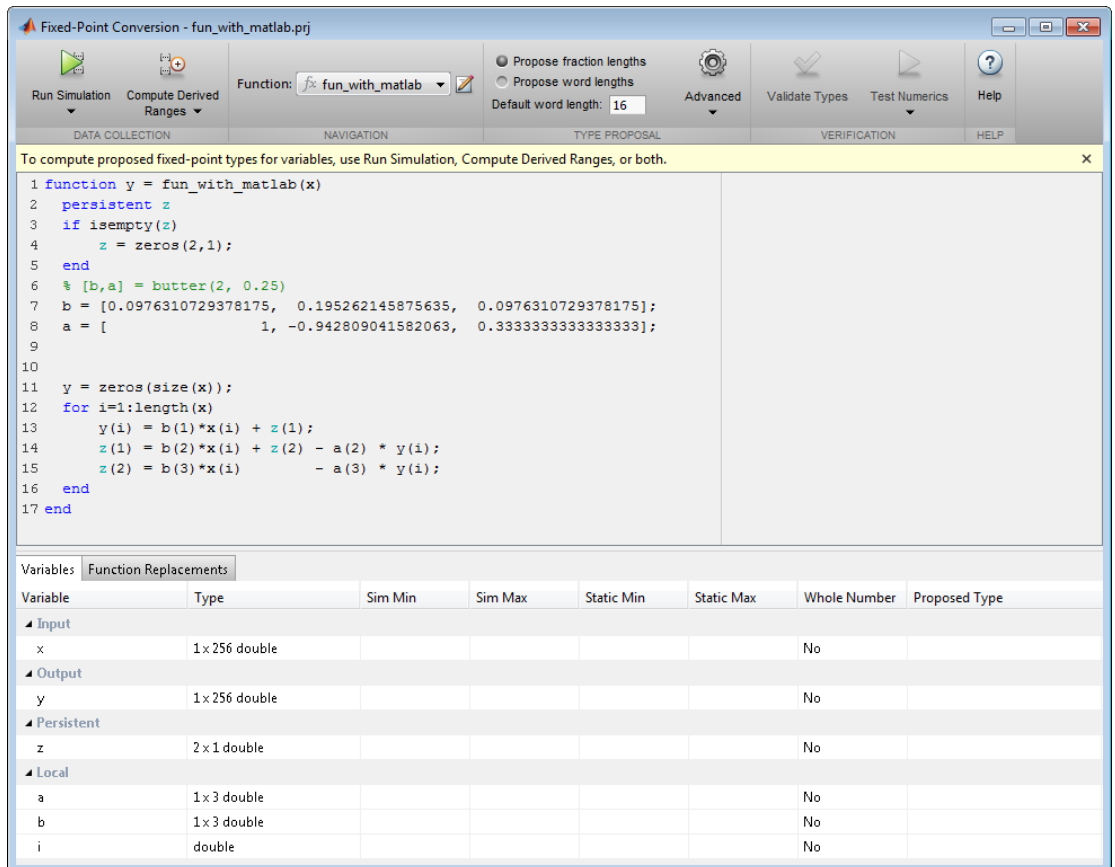
The project indicates that you must first define the fixed-point data types.





- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information”.

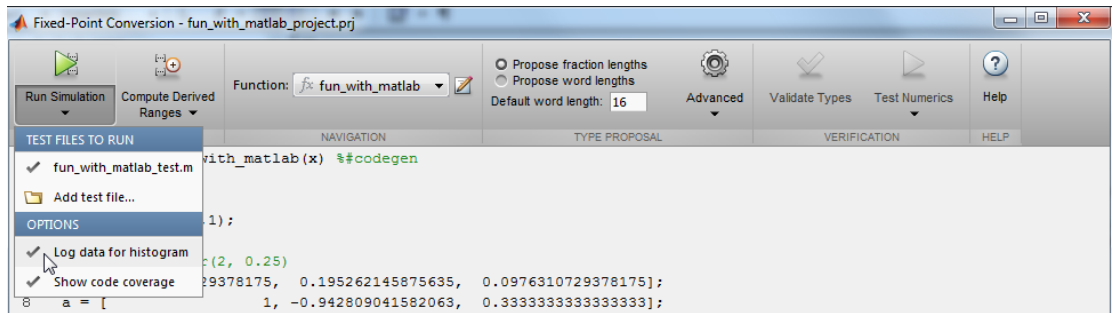


The screenshot shows the Fixed-Point Conversion tool interface. The main window displays a MATLAB function named `fun_with_matlab`. Below the code, there is a table titled "Variables" with a sub-tab "Function Replacements". The table lists variables and their proposed types.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1x256 double					No	
Output							
y	1x256 double					No	
Persistent							
z	2x1 double					No	
Local							
a	1x3 double					No	
b	1x3 double					No	
i	double					No	

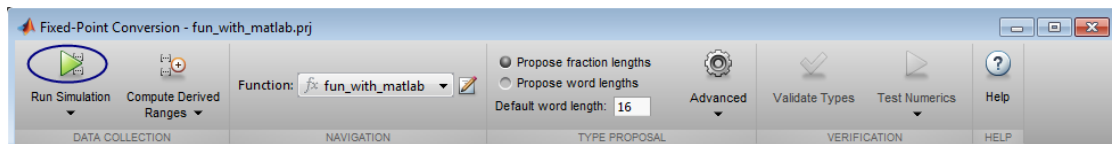
If the MEX function generation fails, the tool provides error message links to help you navigate to the code that caused the build issues. If your code contains functions that are not supported for fixed-point conversion, the tool displays these on the **Function Replacements** tab. For more information, see “Running a Simulation”.

- 3 Click **Run Simulation** and verify that the `fun_with_matlab_test` file is selected as a test file to run. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the conversion tool merges the simulation results. To clear results, right-click the **Variables** tab and select **Reset entire table**.
- 4 Click **Run Simulation** and select **Log data** for histogram.



By default, the Show code coverage option is selected. This option provides code coverage information that helps you verify that your test file is testing your algorithm over the intended operating range.

- 5 Click the Run Simulation button.



The simulation runs and the conversion tool displays a color-coded code coverage bar to the left of the MATLAB code. Review this information to verify that the test file is testing the algorithm adequately. Here, the dark green line to the left of the code indicates that the code is run every time the algorithm is executed. The orange bar indicates that the code next to it is executed only once. In this example, this is the expected behavior because the code is initializing a persistent variable. If your test file is not covering all your code, update the test or add more test files.

The screenshot shows the Fixed-Point Conversion tool interface. At the top, there are tabs for DATA COLLECTION, NAVIGATION, TYPE PROPOSAL, VERIFICATION, and HELP. The main area displays MATLAB code for a function named `fun_with_matlab`. Below the code, there is a table showing the results of the simulation and the proposed fixed-point types for various variables.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1x256 double	-0.9999756307053946	1			No	numerictype(1, 16, 14)
Output							
y	1x256 double	-0.97	1.06			No	numerictype(1, 16, 14)
Persistent							
z	2x1 double	-0.89	0.96			No	numerictype(1, 16, 15)
Local							
a	1x3 double	-0.94	1			No	numerictype(1, 16, 14)
b	1x3 double	0.1	0.2			No	numerictype(0, 16, 18)
i	double		1	256		Yes	numerictype(0, 9, 0)

If a value has . . . next to it, the value is rounded. Place your cursor over the . . . to view the actual value.

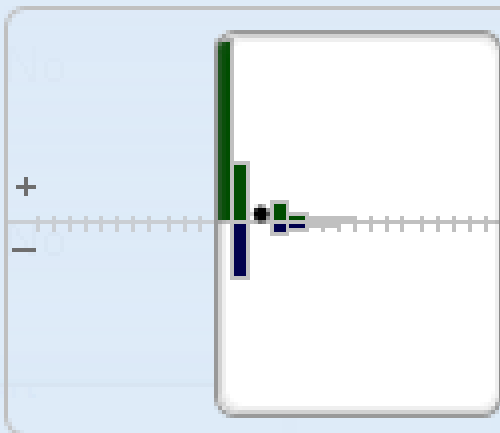
The tool displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

**Note:** If you manually enter static ranges, these manually-entered ranges take precedence over simulation ranges and the tool uses them to propose data types. In addition, you can also modify and lock the proposed type.

- 6 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.



Whole Number Proposed Type

No `numerictype(1, 16, 14)`



Sim values covered **99%**  Signed

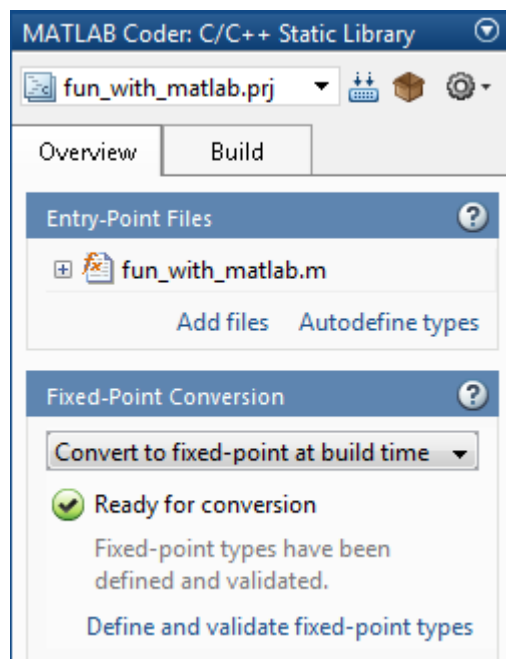
Supported range **-2 : 1.9999**

To modify the proposed data types, either enter the required type into the **ProposedType** field or use the histogram controls. For more information about the histogram, see “Histogram”.

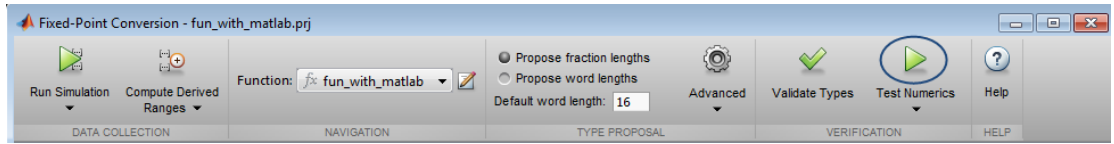
- 7 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a **Validation succeeded** message, and enables the **Test Numerics** option. The project indicates that you have validated the fixed-point data types.



If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see “Validating Types”.

- 8 Click **Test Numerics**, select Log inputs and outputs for comparison plots, and then click the Test Numerics button.



The tool runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable  $y$ . Because you selected to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output.

The maximum error is less than 0.03%. For the purpose of this example, this margin of error is acceptable, so you are ready to generate fixed-point C code.

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics”.

- 9 Return to the MATLAB Coder project.

### Generate Fixed-Point C Code

- 1 In the MATLAB Coder project, verify that the **Fixed-Point Conversion** pane displays **Ready for conversion**, and then select the **Build** tab.
- 2 On this tab, set the **Output type** to **C/C++ Static library**.

The default output file name is `fun_with_matlab`.

- 3 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/fun_with_matlab_fixpt`.

- 4 To view the generated code, click **View report**.

## Propose Fixed-Point Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges that you specify. The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time so you can save time by deriving ranges instead.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\dti`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

- 3 Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>dti.m</code>	Entry-point MATLAB function
Test file	<code>dti_test.m</code>	MATLAB script that tests <code>dti.m</code>



## The dti Function

The dti function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;
```

```
function b = subFunction(a)
b = a*a;
```

## The dti\_test Function

The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10)

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii = 1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;

    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);
end

figure('Name', [mfilename, '_plot'])
subplot(2,1,1)
plot(1:len,x_in)
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

subplot(2,1,2); plot(1:len,y_out)
xlabel('Time')
```

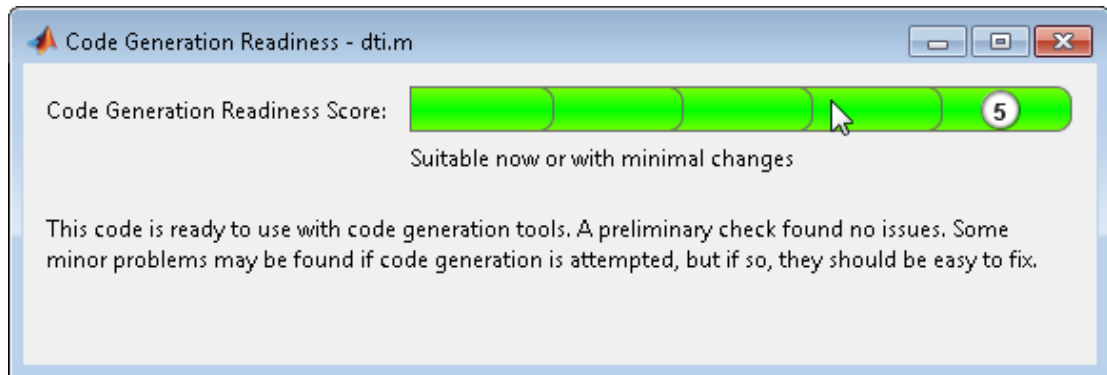
```
ylabel('Amplitude')
title('Output Signal (DTI)')

disp('Test complete.')
```

### Check Code Generation Readiness

In the current working folder, right-click the `dti.m` function. From the context menu, select **Check Code Generation Readiness**.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `dti.m` function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis”.

### Create and set up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `dti.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new dti.prj
```

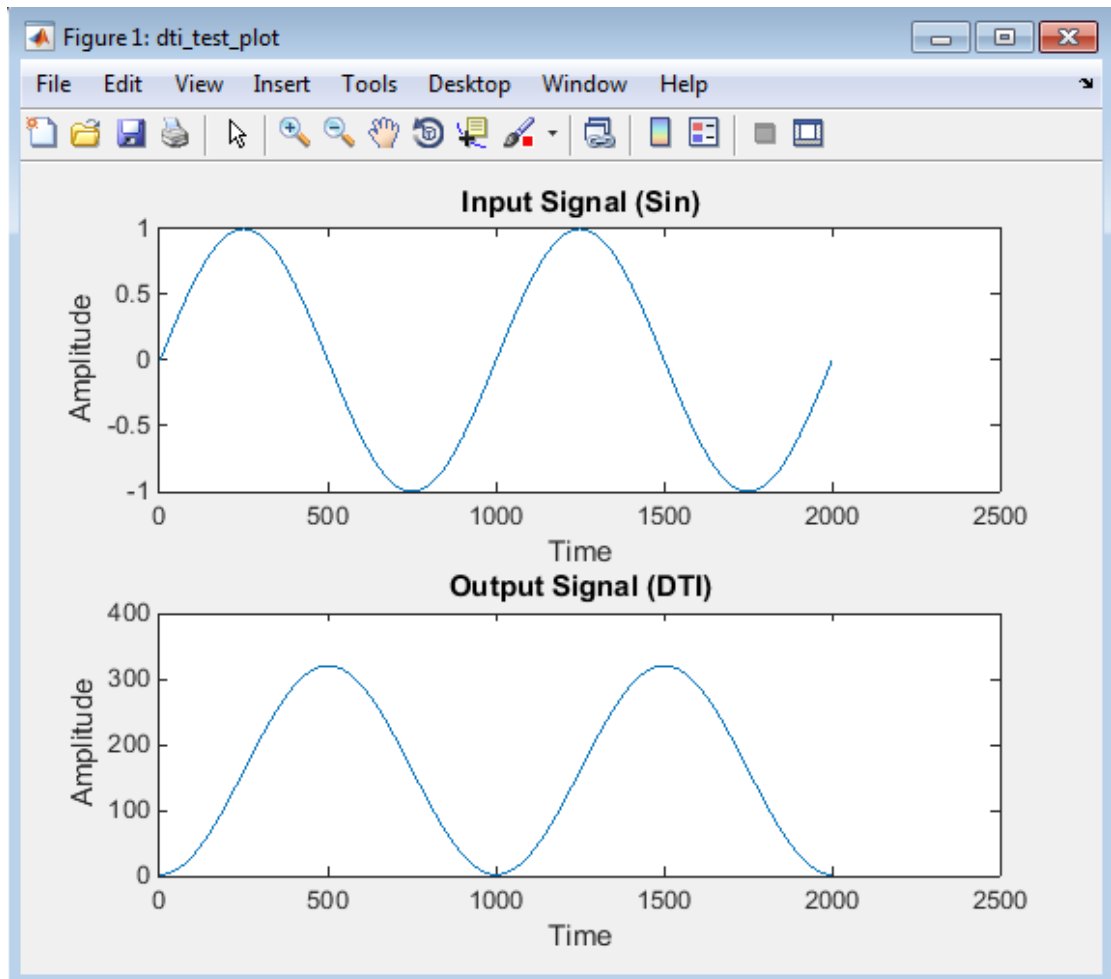
By default, the project opens in the MATLAB workspace.

- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `dti.m` and then click **OK** to add the file to the project.

### Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `dti_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



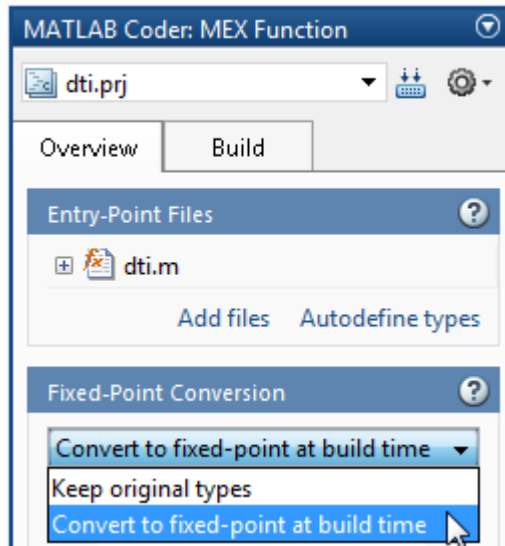
MATLAB Coder determines the input types from the test file and then displays them.

- 3 In the Autodefine Input Types dialog box, click **Use These Types**.

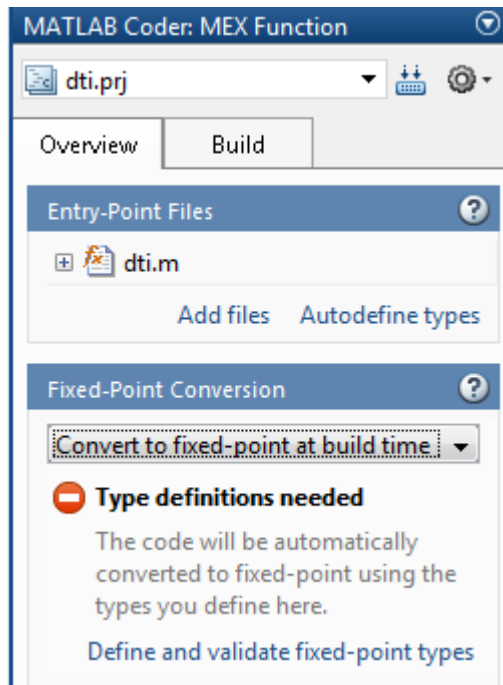
MATLAB Coder sets the type of `x` to `double(1x1)`.

## Fixed-Point Conversion

- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.



The project indicates that you must first define the fixed-point data types.



- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information”.

Fixed-Point Conversion - dti.prj

Run Simulation Compute Derived Ranges Function: dti

Propose fraction lengths  
Propose word lengths  
Default word length: 16

Advanced Validate Types Test Numerics Help

DATA COLLECTION NAVIGATION TYPE PROPOSAL VERIFICATION HELP

To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.

```

1 function [y, clip_status] = dti(u_in)
2 % Discrete Time Integrator in MATLAB
3 %
4 % Forward Euler method, also known as Forward Rectangular, or left-hand
5 % approximation. The resulting expression for the output of the block at
6 % step 'n' is y(n) = y(n-1) + K * u(n-1)
7 %
8 init_val = 1;
9 gain_val = 1;
10 limit_upper = 500;
11 limit_lower = -500;
12
13 % variable to hold state between consecutive calls to this block
14 persistent u_state;

```

Variables		Function Replacements					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
u_in	double					No	
Output							
clip_status	double					No	
y	double					No	
Persistent							
u_state	double					No	
Local							
gain_val	double					No	
init_val	double					No	
limit_lower	double					No	
limit_upper	double					No	
tprod	double					No	

If the MEX function generation fails, the tool provides error message links to help you navigate to the code that caused the build issues. If your code contains functions that are not supported for fixed-point conversion, the tool displays these on the **Function Replacements** tab. For more information, see “Running a Simulation”.

- 3 In the Fixed-Point Conversion window, on the **Variables** tab, for input `u_in`, select **Static Min** and set it to -1. Then set **Static Max** to 1.

To compute derived range information, at a minimum you must specify static minimum and maximum values or proposed data types for all input variables.



---

**Note:** If you manually enter static ranges, these manually-entered ranges take precedence over simulation ranges and the tool uses them to propose data types. In addition, you can also modify and lock the proposed type.

---

- 4 Click the Compute Derived Ranges button.



Range analysis computes the derived ranges and displays them in the **Variables** tab. Using these derived ranges, the analysis proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

In the `dti` function, the `clip_status` output has a minimum value of -2 and a maximum of 2.

```
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```

When you derive ranges, the Fixed-Point Conversion tool analyses the function and computes these minimum and maximum values for `clip_status`.

The screenshot shows the 'Fixed-Point Conversion - dti.prj' window. The top toolbar includes 'Run Simulation', 'Compute Derived Ranges', 'Function: dti', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. Below the toolbar are tabs for 'DATA COLLECTION', 'NAVIGATION', 'TYPE PROPOSAL', 'VERIFICATION', and 'HELP'. The main area contains MATLAB code for a discrete-time integrator. Below the code is a 'Static Analysis Output' table.

```

1 function [y, clip_status] = dti(u_in) %#codegen
2 % Discrete Time Integrator in MATLAB
3 %
4 % Forward Euler method, also known as Forward Rectangular, or left-hand
5 % approximation. The resulting expression for the output of the block at
6 % step 'n' is y(n) = y(n-1) + K * u(n-1)
7 %
8 init_val = 1;
9 gain_val = 1;
10 limit_upper = 500;
11 limit_lower = -500;
12
13 % variable to hold state between consecutive calls to this block
14 persistent u_state;
15 if isempty(u_state)
16     u_state = init_val+1;

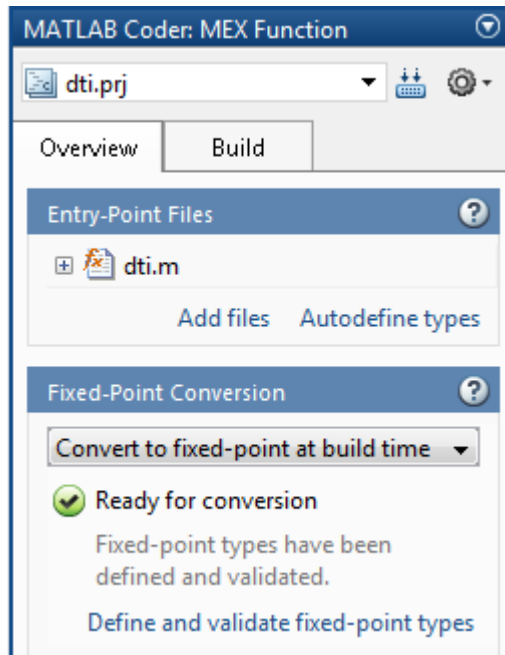
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
<b>Input</b>							
u_in	double			-1	1	No	numerictype(1, 16, 14)
<b>Output</b>							
clip_status	double			-2	2	No	numerictype(1, 16, 13)
y	double			-500	500	No	numerictype(1, 16, 6)
<b>Persistent</b>							
u_state	double			-501	501	No	numerictype(1, 16, 6)
<b>Local</b>							
gain_val	double			1	1	Yes	numerictype(0, 1, 0)
init_val	double			1	1	Yes	numerictype(0, 1, 0)
limit_lower	double			-500	-500	Yes	numerictype(1, 10, 0)
limit_upper	double			500	500	Yes	numerictype(0, 9, 0)
tprod	double			-1	1	No	numerictype(1, 16, 14)


The tool provides a **Quick derived range analysis** option and the option to specify a timeout in case the analysis takes a very long time. For more information, see “Computing Derived Ranges”

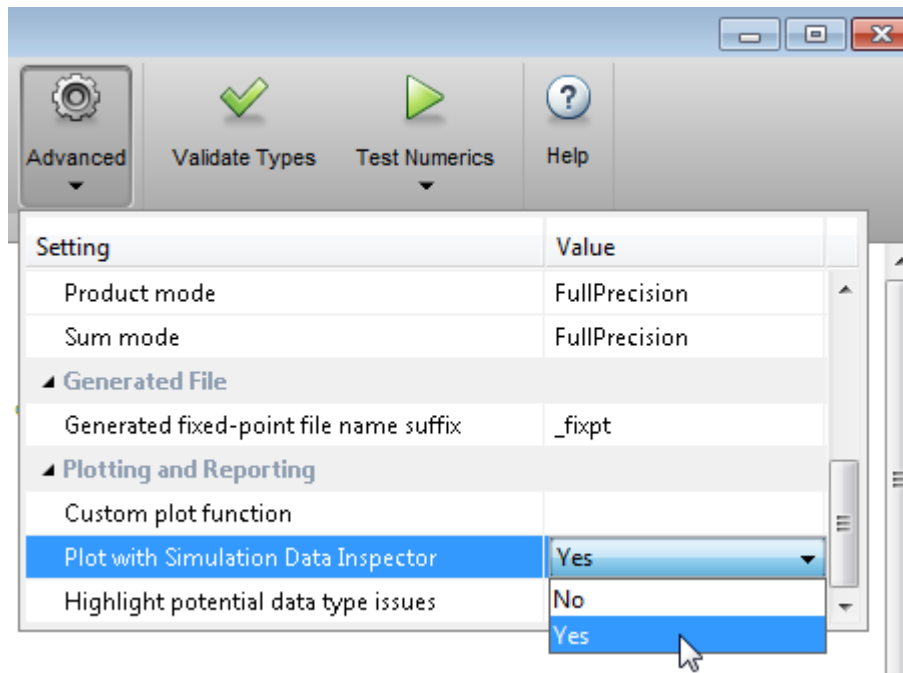
- 5 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a **Validation succeeded** message, and enables the **Test Numerics** option. The project indicates that you have validated the fixed-point data types.

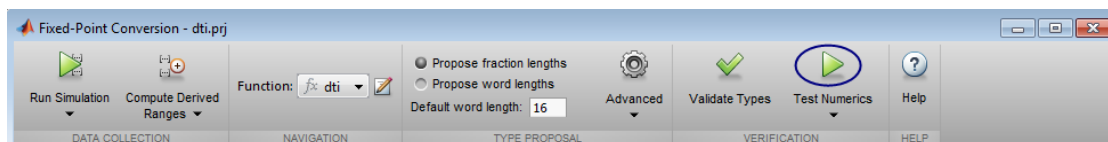


If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see “Validating Types”.

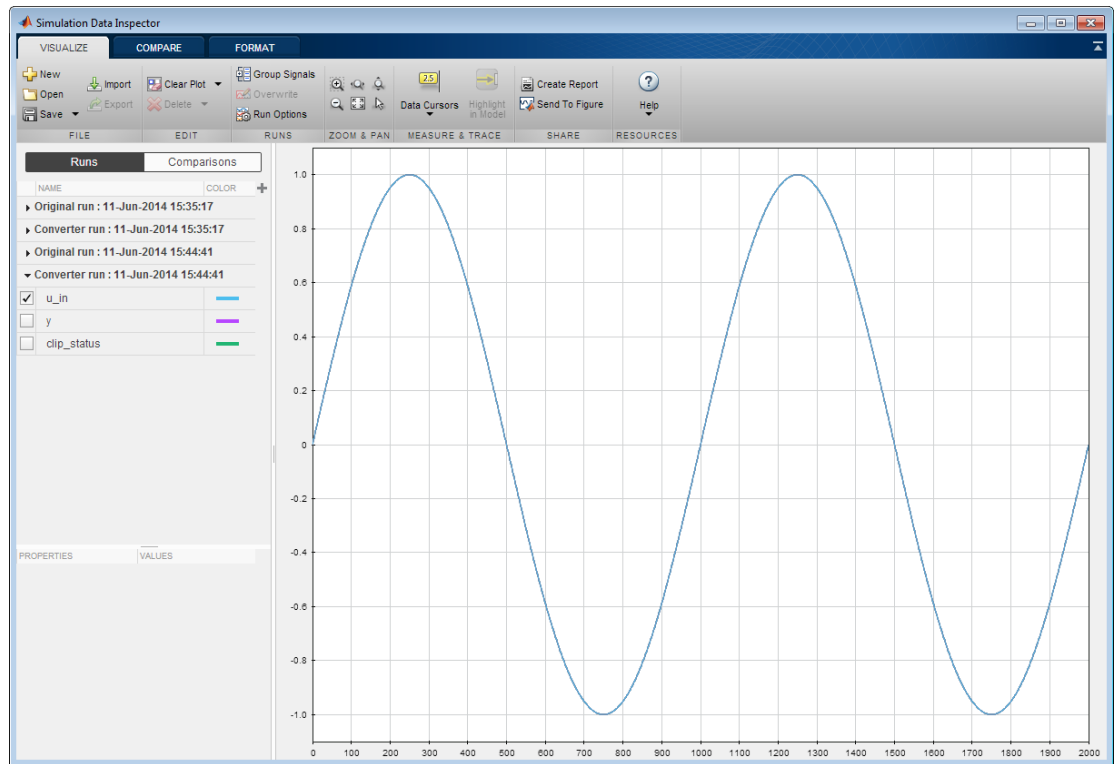
- 6 Use the Simulation Data Inspector to plot the floating-point and fixed-point results.
  - a Click 
  - b Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to Yes.



- 7 Click **Test Numerics**, select Log inputs and outputs for comparison plots, and then click the **Test Numerics** button.



The tool runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable  $y$ . Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.



On the **Verification Output** tab, the tool provides a link to a type proposal report.

The following table shows fixed point instrumentation results

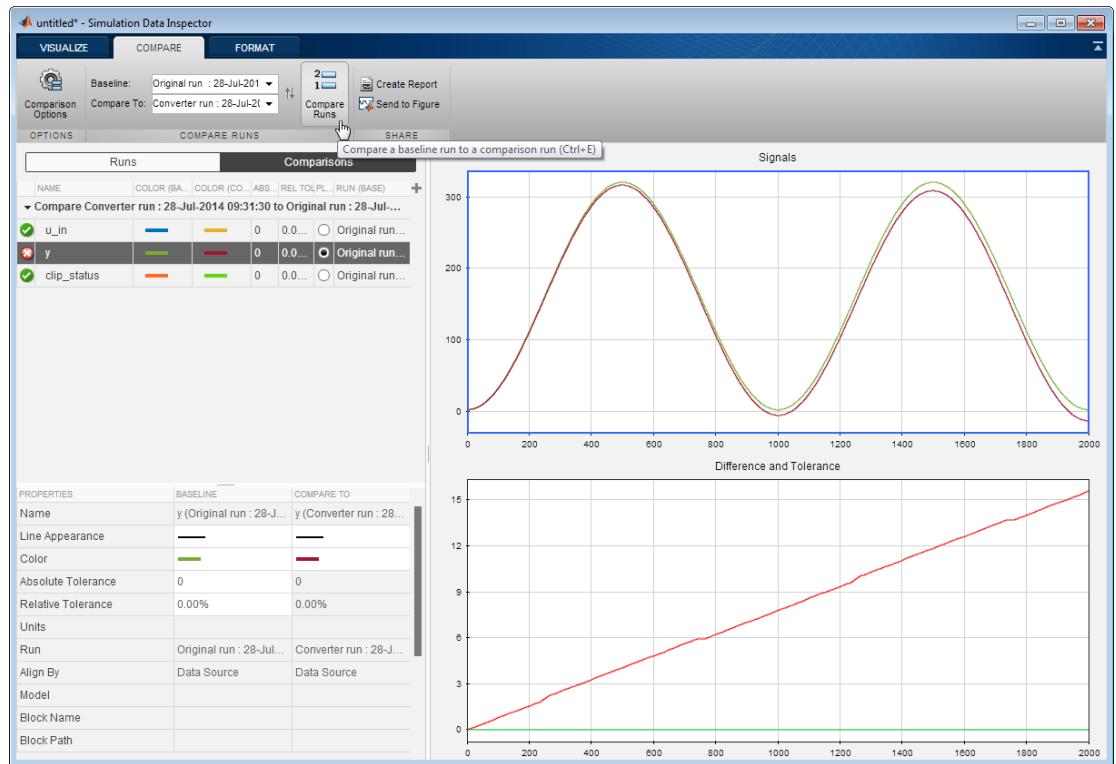
### Fixed-Point Report dti\_fixpt

```
function [y,clip_status] = dti_fixpt(u_in)
    fm = fimath('RoundingMethod','Floor','OverflowAction','Wrap','ProductMode','FullPrecision','SumMode','FullPrecision');
    %#codegen
    % Discrete Time Integrator in MATLAB
    %
    % Forward Euler method, also known as Forward Rectangular, or left-hand
    % approximation. The resulting expression for the output of the block at
    % step 'n' is y(n) = y(n-1) + K * u(n-1)
    %
    init_val = fi(1,0,1,0,fm);
    gain_val = fi(1,0,1,0,fm);
    limit_upper = fi(500,0,9,0,fm);
    limit_lower = fi(-500,1,10,0,fm);
    % variable to hold state between consecutive calls to this block
    persistent u_state
    if isempty(u_state)
        u_state = fi(init_val + fi(1,0,1,0,fm),1,16,6,fm);
    end
    % Compute Output
    if (u_state > limit_upper)
        y = fi(limit_upper,1,16,6,fm);
        clip_status = fi(-2,1,16,13,fm);
    elseif (u_state == limit_upper)
        y = fi(limit_upper,1,16,6,fm);
        clip_status = fi(-1,1,16,13,fm);
    elseif (u_state < limit_lower)
        y = fi(limit_lower,1,16,6,fm);
        clip_status = fi(2,1,16,13,fm);
    elseif (u_state == limit_lower)
        y = fi(limit_lower,1,16,6,fm);
        clip_status = fi(1,1,16,13,fm);
    else
        y = fi(u_state,1,16,6,fm);
        clip_status = fi(0,1,16,13,fm);
    end
    % Update State
    tprod = fi(gain_val*u_in,1,16,14,fm);
    u_state(:) = y + tprod;
end
```

Variable Name	Type	Sim Min	Sim Max
clip_status	numerictype(1, 16, 13)	0	0
gain_val	numerictype(0, 1, 0)	1	1
init_val	numerictype(0, 1, 0)	1	1
limit_lower	numerictype(1, 10, 0)	-500	-500
limit_upper	numerictype(0, 9, 0)	500	500
tprod	numerictype(1, 16, 14)	-1	1
u_in	numerictype(1, 16, 14)	-1	1
u_state	numerictype(1, 16, 6)	-13.59375	316.28125
y	numerictype(1, 16, 6)	-13.578125	316.28125

- You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-point values for the output **y**, on the **Compare** tab, select **y**, and then click **Compare Runs**.

The Simulation Data Inspector displays a plot of the baseline floating-point run against the fixed-point run and the difference between them.



## Generate Fixed-Point C Code

- 1 In the MATLAB Coder project, select the **Build** tab.
- 2 On this tab, set the **Output type** to **C/C++ Static library**.  
The default output file name is `dti`.
- 3 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/dti_fixpt`.

- 4 To view the generated code, click **View report**.

The code generation report opens and displays the generated code for `dti_fixpt.c`. In the generated C code, variables are assigned fixed-point data types.

## Type Proposal Settings

Basic Type Proposal Settings	Values	Description
Fixed-point type proposal mode	Propose fraction lengths for specified word length	Use the specified word length for data type proposals and propose the minimum fraction lengths to avoid overflows.
	Propose word lengths for specified fraction length (default)	Use the specified fraction length for data type proposals and propose the minimum word lengths to avoid overflows.
Default word length	16 (default)	Default word length to use when Fixed point type proposal mode is set to Propose fraction lengths for specified word lengths
Default fraction length	4 (default)	Default fraction length to use when Fixed point type proposal mode is set to Propose word lengths for specified fraction lengths

Advanced Type Proposal Settings	Values	Description
When proposing types	ignore simulation ranges	Propose data types based on derived ranges.
<b>Note:</b> Manually-entered static ranges always take precedence over simulation ranges.	ignore derived ranges	Propose data types based on simulation ranges.
	use all collected data (default)	Propose data types based on both simulation and derived ranges.
Propose target container types	Yes	Propose data type with the smallest word length that can represent the range and is suitable for C code generation ( 8,16,32, 64 ... ). For



Advanced Type Proposal Settings	Values	Description
		example, for a variable with range [0 . 7], propose a word length of 8 rather than 3.
	No (default)	Propose data types with the minimum word length needed to represent the value.
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.
Safety margin for sim min/max (%)	0 (default)	Specify safety factor for simulation minimum and maximum values.  The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of 55 specifies that you want a range at least 55 percent larger. A value of -15 specifies that a range up to 15 percent smaller is acceptable.

fimath Settings	Values	Description
Rounding method	Ceiling	Specify the fimath properties for the generated fixed-point data types.
	Convergent	

<b>fimath Settings</b>	<b>Values</b>	<b>Description</b>
	Floor (default)	The default fixed-point math properties use the <b>Floor</b> rounding and <b>Wrap</b> overflow because they are the default actions in C. These settings generate the most efficient code but might cause problems with overflow.
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification.
	Wrap (default)	
Product mode	FullPrecision (default)	After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification.
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification.
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Product word length	32 (default)   any positive integer	Word length, in bits, of the product data type
Sum word length	32 (default)   any positive integer	Word length, in bits, of the sum data type

<b>Generated File Settings</b>	<b>Value</b>	<b>Description</b>
Generated fixed-point file name suffix	_fixpt (default)	Specify the suffix to add to the generated fixed-point file names. For example, by default, if you generate a static library for a project named <code>test</code> , the generated files are in the subfolder <code>codegen\lib\test_fixpt</code> . The generated static library is named <code>test.lib</code> , but the generated C code files

Generated File Settings	Value	Description
		use the suffix, for example, <code>test_fixpt.c</code> .
Plotting and Reporting Settings	Values	Description
Custom plot function	Empty string	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	Specify whether to use the Simulation Data Inspector for comparison plots.
	Yes	
Highlight potential data type issues	No (default)	Specify whether to highlight potential data types in the generated html report. If this option is turned on, the report highlights single-precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.
	Yes	

## Detect Overflows

This example shows how to detect overflows using the Fixed-Point Conversion tool. 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.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer

### Create a Project

In a local writable folder, create a MATLAB Coder project named `overflow`.

At the MATLAB command line, enter

```
coder -new overflow.prj
```

By default, the project opens in the MATLAB workspace.

### Add File

- 1 In the same folder, create a function, `overflow`.

```
function y = overflow(b,x,reset)
    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
```

```

        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
end

```

- 2 On the project **Overview** tab, click the **Add files** link. Browse to the file `overflow.m` and then click **OK** to add the file to the project.

### Define Input Types

- 1 Create a test file, `overflow_test.m` to exercise the `overflow` algorithm. You use this test file to define input types for `b`, `x`, and `reset`, and, later, to verify the fixed-point version of the algorithm.

```

function overflow_test
    % The filter coefficients were computed using the FIR1 function from
    % Signal Processing Toolbox.
    % b = fir1(11,0.25);
    b = [-0.004465461051254
        -0.004324228005260
        +0.012676739550326
        +0.074351188907780
        +0.172173206073645
        +0.249588554524763
        +0.249588554524763
        +0.172173206073645
        +0.074351188907780
        +0.012676739550326
        -0.004324228005260
        -0.004465461051254]';

    % Input signal
    nx = 256;
    t = linspace(0,10*pi,nx)';

    % Impulse
    x_impulse = zeros(nx,1); x_impulse(1) = 1;

    % Max Gain

```

```
% The maximum gain of a filter will occur when the inputs line up with the
% signs of the filter's impulse response.
x_max_gain = sign(b)';
x_max_gain = repmat(x_max_gain,ceil(nx/length(b)),1);
x_max_gain = x_max_gain(1:nx);

% Sums of sines
f0=0.1; f1=2;
x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);

% Chirp
f_chirp = 1/16; % Target frequency
x_chirp = sin(pi*f_chirp*t.^2); % Linear chirp

x = [x_impulse, x_max_gain, x_sines, x_chirp];
titles = {'Impulse', 'Max gain', 'Sum of sines', 'Chirp'};
y = zeros(size(x));

for i=1:size(x,2)
    reset = true;
    y(:,i) = overflow(b,x(:,i),reset);
end

test_plot(1,titles,t,x,y)

end
function test_plot(fig,titles,t,x,y1)
    figure(fig)
    clf
    sub_plot = 1;
    font_size = 10;
    for i=1:size(x,2)
        subplot(4,1,sub_plot)
        sub_plot = sub_plot+1;
        plot(t,x(:,i),'c',t,y1(:,i),'k')
        axis('tight')
        xlabel('t','FontSize',font_size);
        title(titles{i},'FontSize',font_size);
        ax = gca;
        ax.FontSize = 10;
    end
    figure(gcf)
end
```

- 2 On the project **Overview** tab, click the **Autodefine types** link.
- 3 In the Autodefine Input Types dialog box, add `overflow_test` as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals. MATLAB Coder determines the input types from the test file and then displays them.

- 4 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the types of `b` to `double(1x12)`, `x` to `double(256x1)`, and `reset` to `logical(1x1)`.

### Fixed-Point Conversion

- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

The project indicates that you must first define the fixed-point data types.

- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion tool opens and generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information”.

- 3 In the Fixed-Point Conversion tool, click **Advanced** to view the advanced settings. Set the fimath **Product mode** and **Sum mode** to **KeepLSB**. These settings models the behavior of integer operations in the C language.

The screenshot shows the Fixed-Point Conversion tool interface for a project named 'overflow.prj'. The main window displays a MATLAB function definition for `overflow(b,x,reset)`. The function includes logic for handling overflow and resetting a filter state. The 'Function' dropdown is set to `overflow`. The 'Default word length' is set to 16. The 'Advanced' settings panel is open, showing the following configuration:

Setting	Value
Transform for-loop index variables	No
<b>fimath</b>	
Rounding method	Floor
Overflow action	Wrap
Product mode	<b>KeepLSB</b>
Sum mode	<b>KeepLSB</b>
Product word length	32
Sum word length	32

- 4 In the Fixed-Point Conversion tool, click **Run Simulation** and verify that the `overflow_test` file is selected as the test file to run and then click the Run Simulation button.

The simulation runs and the conversion tool displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
<b>Input</b>							
b	1 x 12 double	0	0.25			No	numerictype(1, 16, 17)
reset	logical	1	1			Yes	numerictype(0, 1, 0)
x	256 x 1 double	-1.1	1.09			No	numerictype(1, 16, 14)
<b>Output</b>							
y	256 x 1 double	-1	1.04			No	numerictype(1, 16, 14)
<b>Persistent</b>							
p	double		0	4		Yes	numerictype(0, 3, 0)
z	1 x 12 double		-1	1		No	numerictype(1, 16, 14)

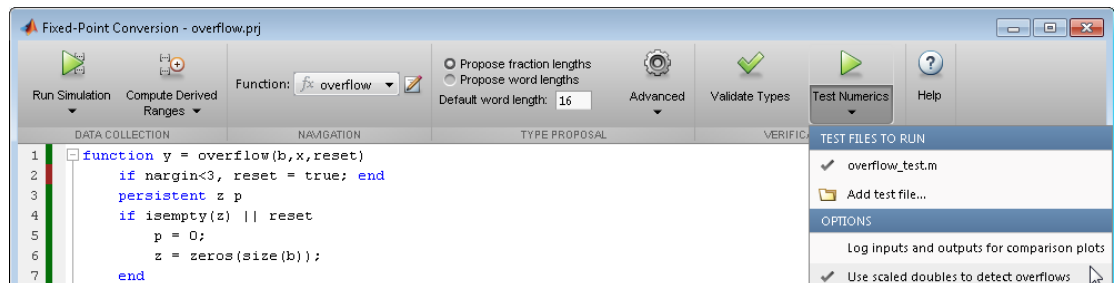
- 5 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types, displays a **Validation succeeded** message, and enables the **Test Numerics** option.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. For more information, see “Validating Types”.

### Test Numerics and Check for Overflows

- 1 Click **Test Numerics**, select **Use scaled doubles to detect overflows**, and then click the Test Numerics button.





The tool runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to detect overflows, it also runs the simulation using scaled double versions of the proposed fixed-point types. Scaled doubles store their data in double-precision floating-point, so they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type.

The simulation runs and the tool detects an overflow. The tool reports on the **Overflow** tab and highlights the expression that overflowed in the code window.

The screenshot shows the Fixed-Point Conversion tool interface. The main window displays MATLAB code for a function named `fir_filter`. The code includes several fixed-point conversion calls (`fi`) and a loop that calculates the output `y(n)`. The expression `acc + b(j)*z(k)` on line 42 is highlighted in orange, indicating an overflow error.

Below the code editor, the **Overflow** tab is selected, showing a table with the following data:

Function	Line	Description
fir_filter	42	Overflow error in expression 'acc + b(j)*z(k)'. Percentage of Current Range = 104%.

- Determine whether it was the sum or the multiplication that overflowed.

In the **Advanced** settings, set **Product mode** to **FullPrecision**, and then click **Test Numerics**.

The overflow still occurs, indicating that it is the addition in the expression that is overflowing.

## Replace the exp Function with a Lookup Table

This example shows how to replace the `exp` function with a lookup table approximation in fixed-point code generated using the Fixed-Point Conversion tool.

### Create Algorithm and Test Files

- 1 Create a MATLAB function, `my_fcn`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

- 2 Create a test file, `my_fcn_test`, that uses `my_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

### Create and Set Up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `fun_with_matlab_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new my_fcn_project.prj
```

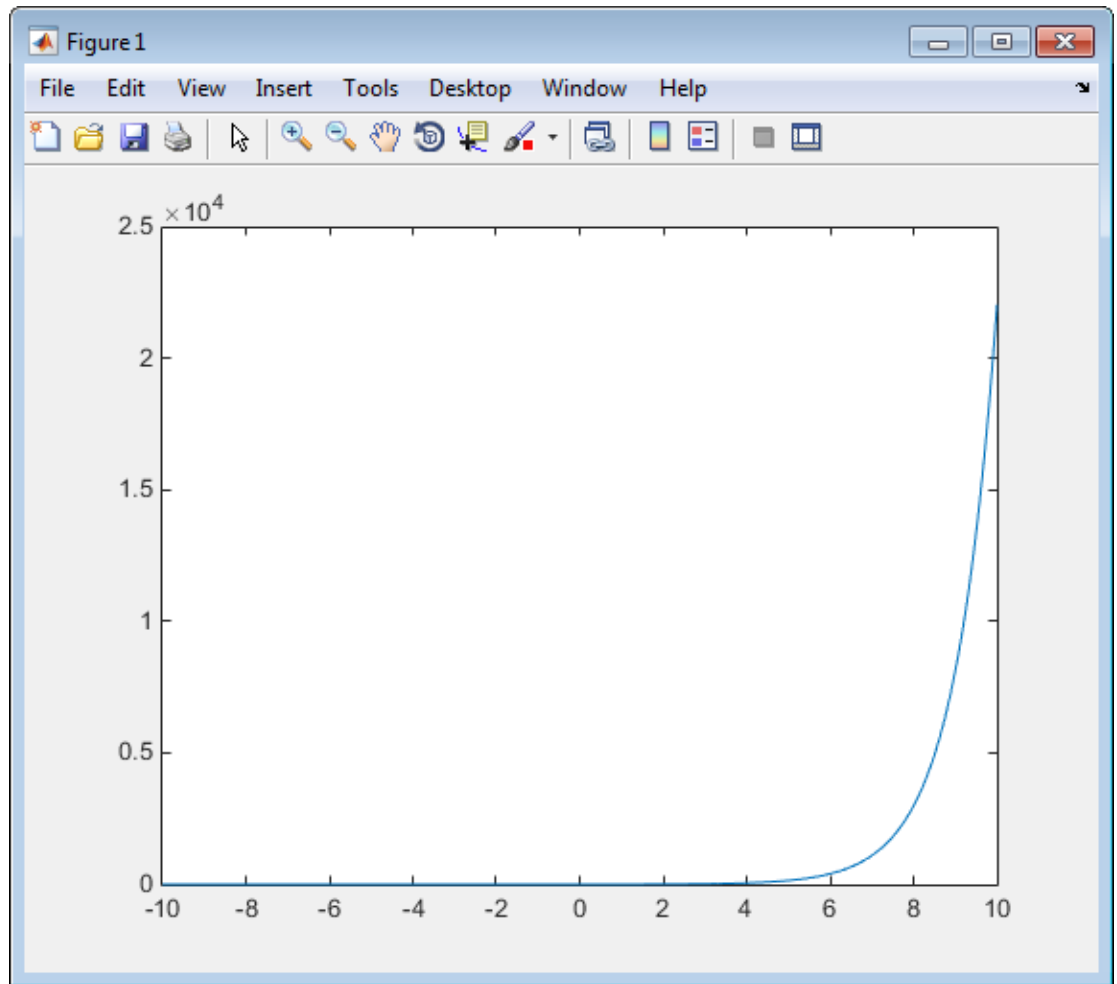
By default, the project opens in the MATLAB workspace.

- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `my_fcn.m` and then click **OK** to add the file to the project.

### Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `my_fcn_test` as a test file and then click **Run**.

The test file runs and plots the output.



- 3 MATLAB Coder determines from the test file that  $x$  is a scalar double.
- 4 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of  $x$  to `double(1x1)`.

## Fixed-Point Conversion

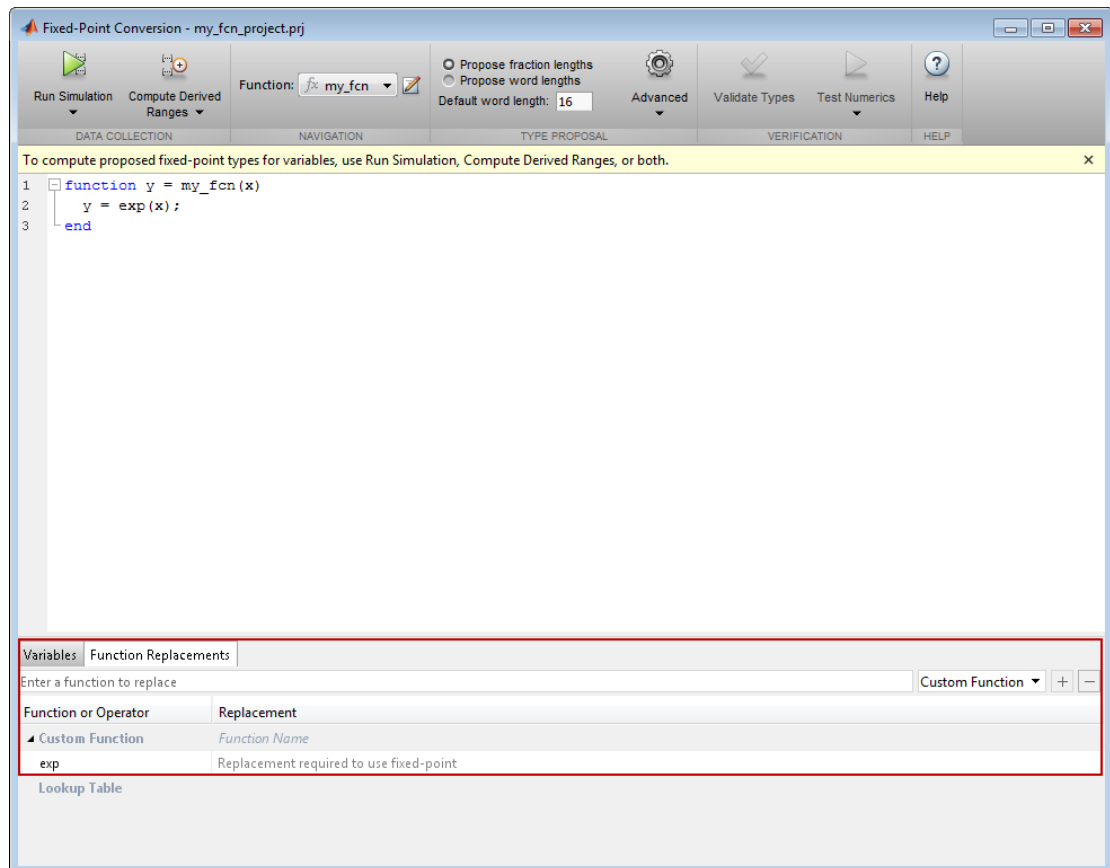
- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

The project indicates that you must first define the fixed-point data types.

- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.


The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code.

The tool displays functions in your original algorithm that are not supported for fixed-point conversion on the **Function Replacements** tab.



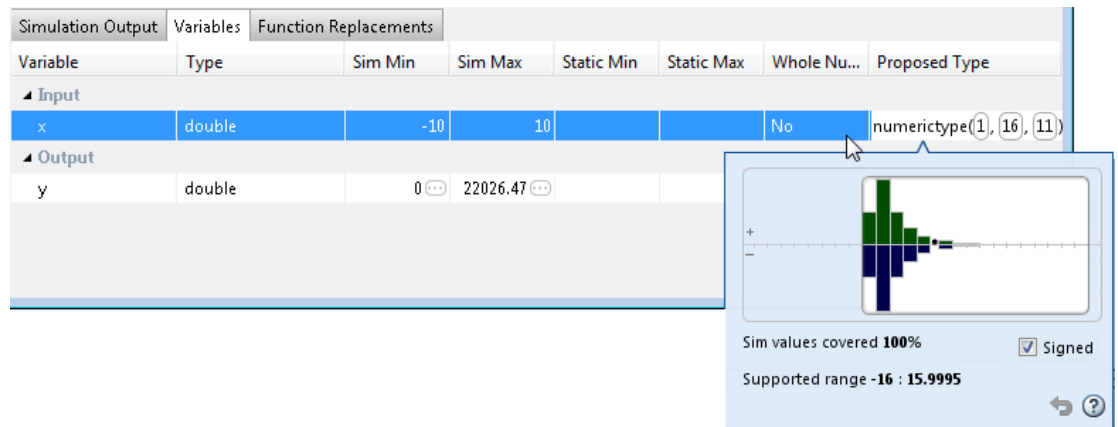
- 3 On the **Function Replacements** tab, right-click the **exp** function and select **Lookup Table**.

The tool moves the **exp** function to the list of functions that it will replace with a **Lookup Table**. By default, the lookup table uses linear interpolation, 1000 points, and the design minimum and maximum values that the tool detects by either running a simulation or computing derived ranges.

- 4 Click **Run Simulation**, select **Log data** for histogram, and verify that the **my\_fcn\_test** file is selected as a test file to run.
- 5 Click the **Run Simulation** button, 

The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

- Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field. The histogram provides range information and the percentage of simulation range covered by the proposed data type.



- To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point function, `my_fcn_fixpt`.

- On the **Type Validation Output** tab, click the `my_fcn_fixpt` link to view the generated fixed-point code.

The conversion process generates a lookup table approximation, `exp1`, for the `exp` function.

The generated fixed-point function, `my_fcn_fixpt`, calls this approximation instead of calling `exp`.

```
function y = my_fcn_fixpt(x)
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
        'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...
```

```
'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
  
y = fi(exp1(x), 0, 16, 1, fm);  
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.



## Replace a Custom Function with a Lookup Table

This example shows how to replace a custom function with a lookup table approximation function using the Fixed-Point Conversion tool.

### Create Algorithm and Test Files

In a local, writable folder:

- 1 Create a MATLAB function, `custom_fcn` which is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

- 2 Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

- 3 Create a test file, `custom_test`, that uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

### Create and Set Up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the MATLAB Coder Project dialog box, set **Name** to `custom_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new custom_project.prj
```

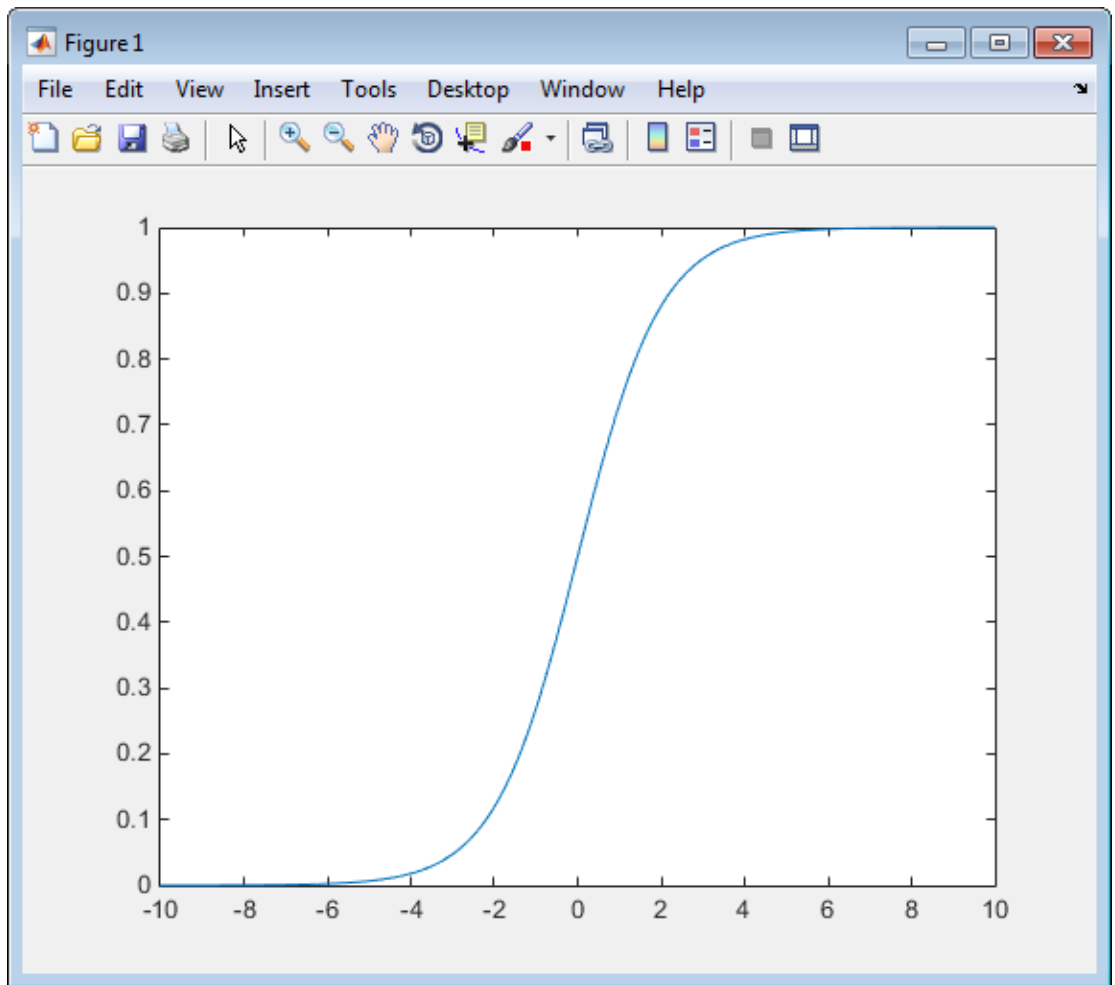
By default, the project opens in the MATLAB workspace.

- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `call_custom_fcn.m` and then click **OK** to add the file to the project.

### Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `custom_test` as a test file and then click **Run**.

The test file runs and plots the output.



- 3 MATLAB Coder determines from the test file that `x` is a scalar double.

- 4 In the Autodefine Input Types dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x1)`.

### **Fixed-Point Conversion**

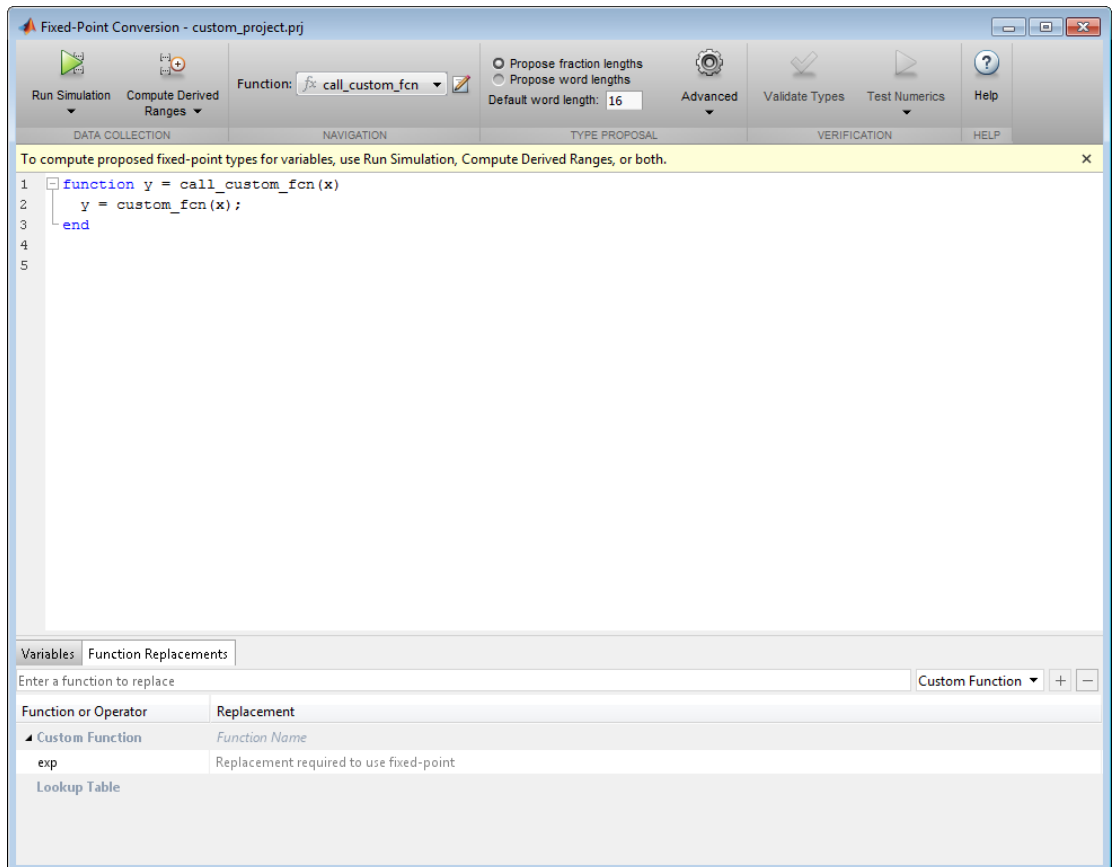
- 1 On the project **Overview** tab **Fixed-Point Conversion** pane, select **Convert to fixed-point at build time**.

The project indicates that you must first define the fixed-point data types.

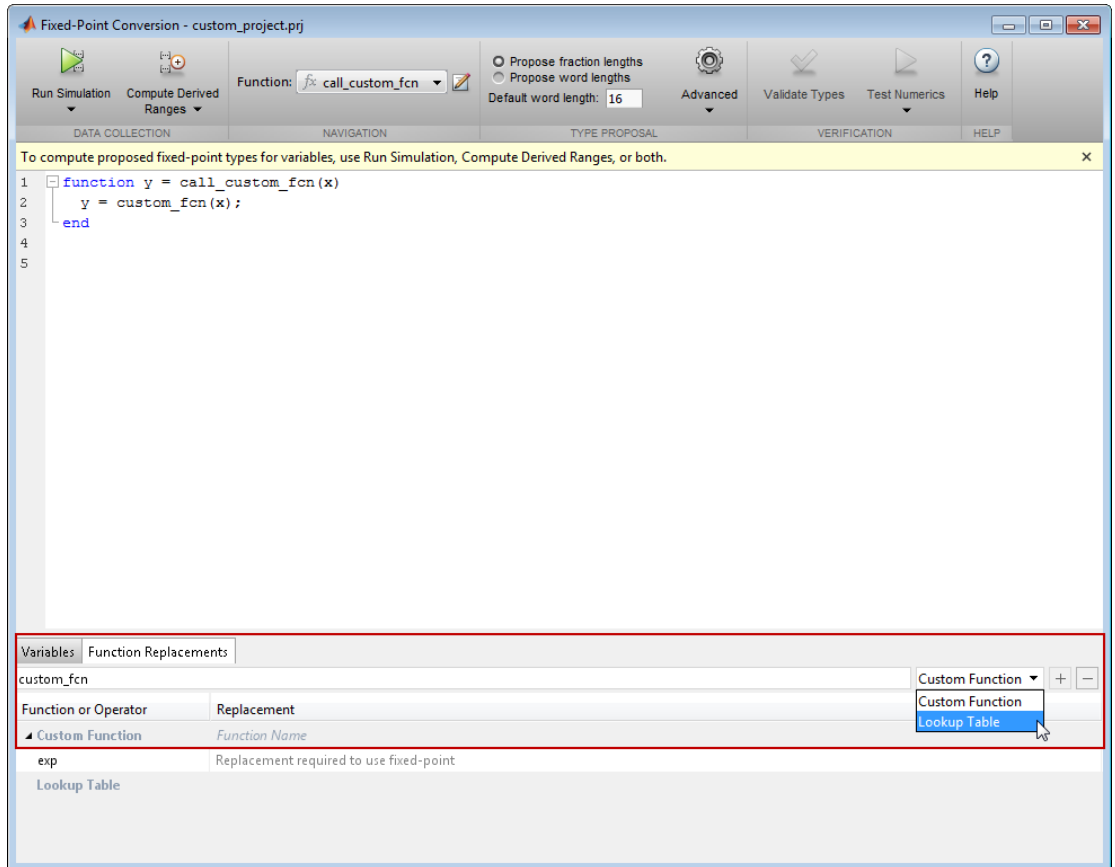
- 2 In the **Fixed-Point Conversion** pane, click **Define and validate fixed-point types**.

The Fixed-Point Conversion window opens and the tool generates an instrumented MEX function for your entry-point MATLAB function. After generating the MEX function, the tool displays compiled information — type, size, and complexity — for variables in your code.

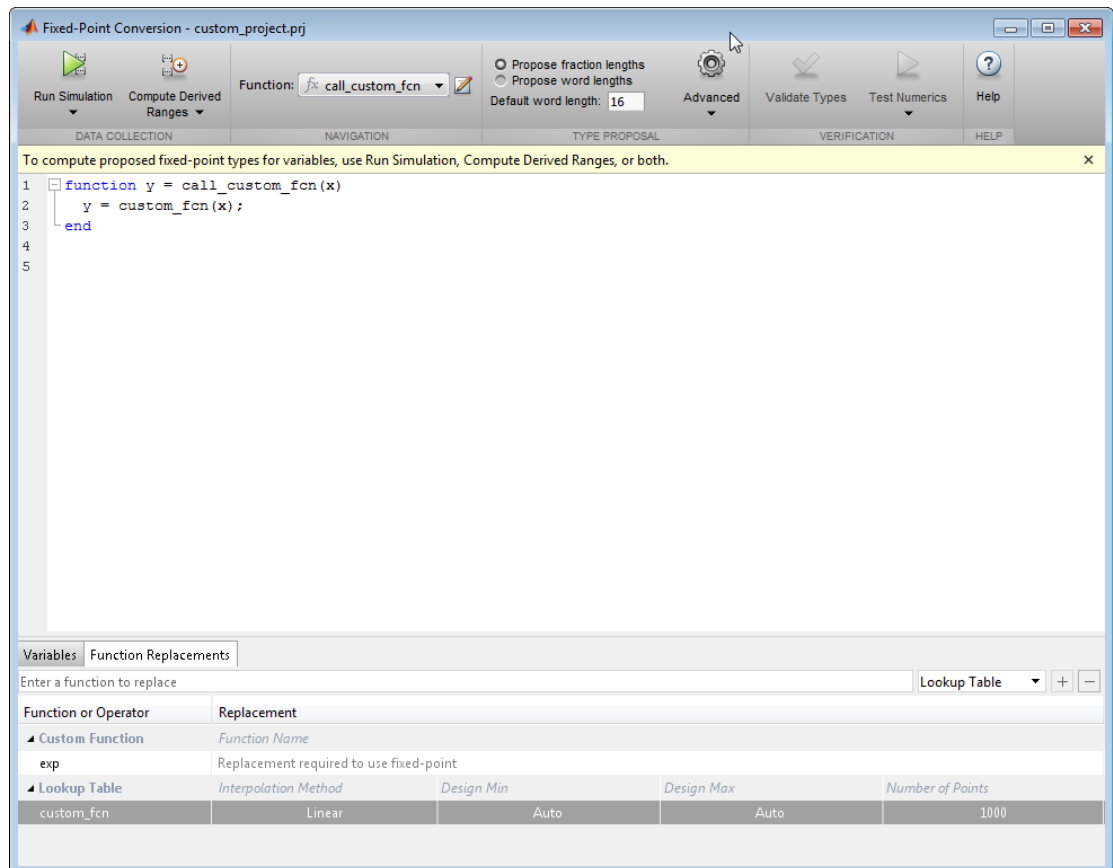
- 3 Select the **Function Replacements** tab.



- 4 Enter the name of the function to replace, `custom_fcn`, select `Lookup Table`, and then click `+`.



The tool adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation, 1000 points, and the design minimum and maximum values that the app detects by either running a simulation or computing derived ranges.

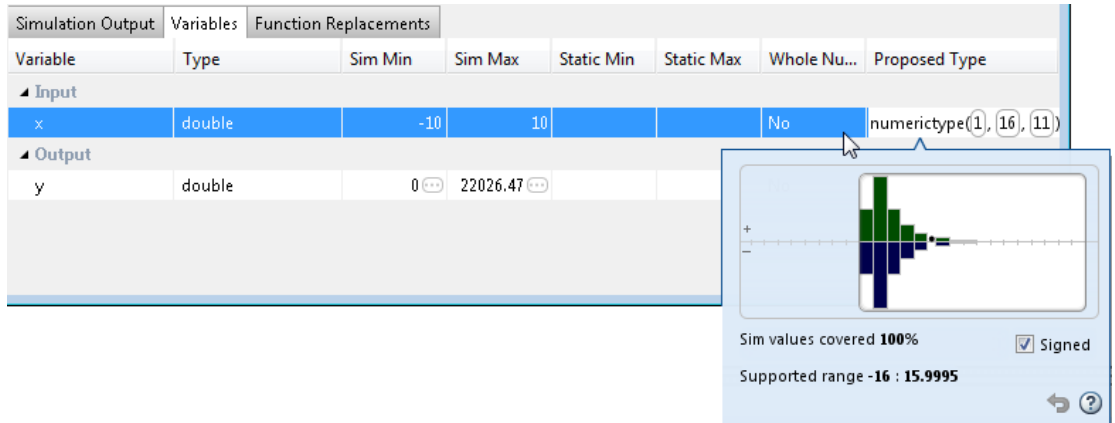


- 5 Click **Run Simulation**, select **Log data for histogram** and verify that the `custom_test` file is selected as a test file to run.
- 6 Click the **Run Simulation** button.

The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Validate Types** option is now enabled.

- 7 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field. The

histogram provides range information and the percentage of simulation range covered by the proposed data type.



- 8 To validate the build using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

- 9 On the Type Validation Output tab, click the `call_custom_fcn_fixpt` link to view the generated fixed-point code.

The conversion process generates a lookup table approximation, `custom_fcn1`, for the `custom_fcn` function.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

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

    y = fi(custom_fcn1(x), 0, 16, 16, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function.

## Enable Plotting Using the Simulation Data Inspector

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point logged input and output data. In the Fixed-Point Conversion tool:

- 1 Click **Advanced**.
- 2 In the Advanced Settings dialog box, set **Plot with Simulation Data Inspector** to **Yes**.
- 3 At the Test Numerics stage in the conversion process, click **Test Numerics**, select **Log inputs and outputs for comparison plots**, and then click



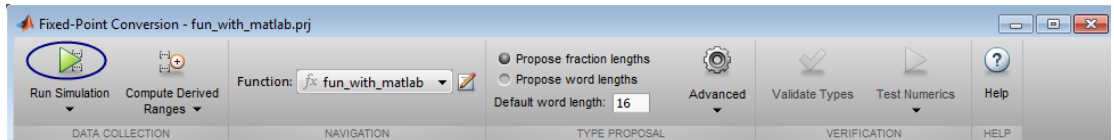
For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges”.



## Log Data for Histogram

To log data for histograms:

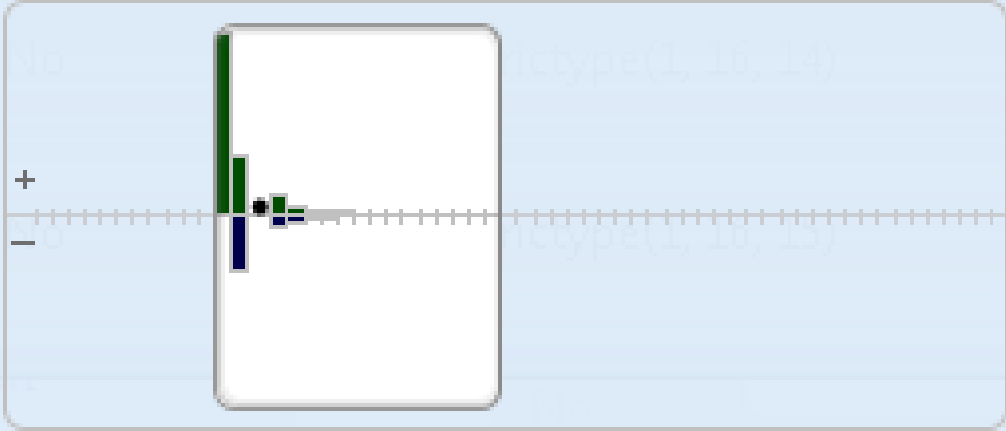
- 1 In the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the Run Simulation button.



The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

- 2 To view a histogram for a variable, click the variable's **Proposed Type** field.

Whole Number	Proposed Type
No	<code>numerictype(1, 16, 14)</code>



Sim values covered **99%**

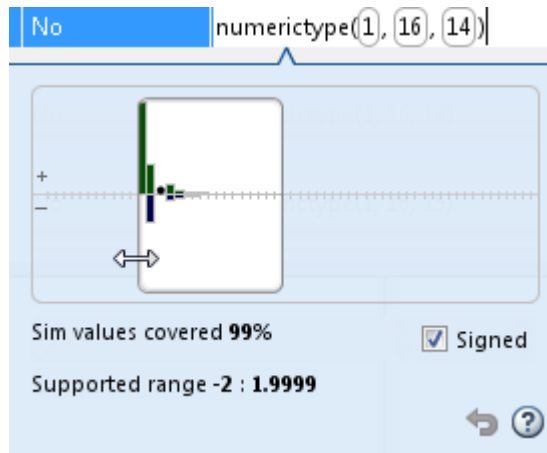
Signed

Supported range **-2 : 1.9999**


- 3 You can view the effect of changing the proposed data types by:
  - Selecting and dragging the white bounding box in the histogram window. This action does not change the word length of the proposed data type, but modifies

the position of the binary point within the word so that the fraction length of the proposed data type changes.

- Selecting and dragging the left edge of the bounding box to increase or decrease the word length. This action does not change the fraction length or the position of the binary point.



- Selecting and dragging the right edge to increase or decrease the fraction length of the proposed data type. This action does not change the position of the binary point. The word length changes to accommodate the fraction length.
- Selecting or clearing **Signed**. Clear **Signed** to ignore negative values.

Before committing changes, you can revert to the types proposed by the automatic conversion by clicking .

## View and Modify Variable Information

### View Variable Information

To view information about the variables in the MATLAB function selected in the **Navigation** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 14-94.

You can view the variable information:

- **Variable**

Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.

- **Type**

The original size, type, and complexity of each variable.

- **Sim Min**

The minimum value assigned to the variable during simulation.

- **Sim Max**

The maximum value assigned to the variable during simulation.

To search for variables in the MATLAB code pane and on the **Variables** tab, use **Ctrl+F**. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

### Modify Variable Information

If you modify variable information, the tool highlights the values in bold. You can modify the following fields:

- **Static Min**

You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 14-65.

Editing this field does not trigger static range analysis, but the tool uses the edited values in subsequent analyses.

- **Static Max**

You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 14-65.

Editing this field does not trigger static range analysis, but the tool uses the edited values in subsequent analyses.

- **Whole Number**

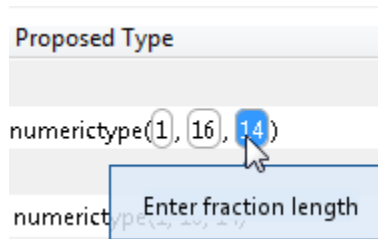
The Fixed-Point Conversion tool uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

Editing this field does not trigger static range analysis, but the tool uses the edited value in subsequent analyses.

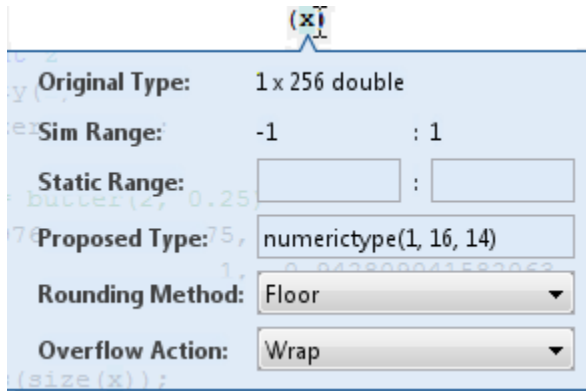
- **Proposed Type**

You can modify the signedness, word length, and fraction length settings individually by:

- On the **Variables** tab, by modifying the value in the **ProposedType** field.



- In the code window, by selecting a variable and then modifying the **ProposedType** field.



If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see “Histogram” on page 14-100.

## Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select **Reset entire table**.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
<b>Input</b>							
x	1 x 256 double	-1	1			No	numerictype(1, 16, 12)
<b>Output</b>							
y	1 x 256 double	-0.97	1.06				numerictype(1, 16, 14)
<b>Persistent</b>							
z	2 x 1 double	-0.89	0.96				numerictype(1, 16, 15)
<b>Local</b>							
a	1 x 3 double	-0.94	1				numerictype(1, 16, 14)
b	1 x 3 double	0.1	0.2			No	numerictype(0, 16, 18)
i	double	1	256			Yes	numerictype(0, 9, 0)

- To revert the type of a selected variable to the type computed by the tool, right-click the field and select **Undo changes**.
- To revert changes to variables, right-click the field and select **Undo changes for all variables**.



## Build Instrumented MEX Function

---

**Note:** This capability is not compatible with automatic fixed-point conversion. If you select `Convert to fixed point at build time`, you cannot build instrumented MEX functions.

---

- 1 In the project, click the **Build** tab.
- 2 On the **Build** tab, set the **Output type** to `Instrumented MEX Function`.
- 3 Click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates an instrumented MEX function in the current folder. It also provides a link to the report on the **Show Instrumentation Results** pane. In this report, you can view the types of variables in your MATLAB code.

After you run the instrumented MEX function, the instrumentation report provides fixed-point data type proposals based on the simulation range data. You can use this information to convert your MATLAB code to fixed point by hand. For more information, see “Propose Fixed-Point Data Types” on page 14-67



## Propose Fixed-Point Data Types

This example shows how to propose fixed-point data types using an instrumented MEX function.

This capability is not compatible with automatic fixed-point conversion. If you select **Convert to fixed point at build time**, you cannot build instrumented MEX functions.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:
 

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```
- 3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>fun_with_matlab.m</code>	Entry-point MATLAB function
Test file	<code>fun_with_matlab_test.m</code>	MATLAB script that tests <code>fun_with_matlab.m</code>

### The `fun_with_matlab` Function

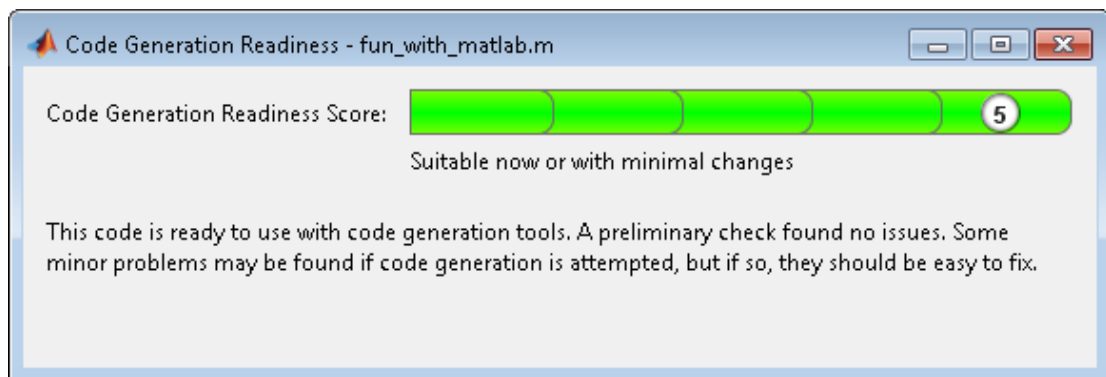
```
function y = fun_with_matlab(x) %#codegen
    persistent z
    if isempty(z)
        z = zeros(2,1);
    end
    % [b,a] = butter(2, 0.25)
    b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
    a = [1, -0.942809041582063, 0.333333333333333];

    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
end
```

### Check Code Generation Readiness

In the current working folder, right-click the `fun_with_matlab.m` function. From the context menu, select **Check Code Generation Readiness**.

The code generation readiness tool screens the code for features and functions that are not supported for code generation. The tool reports that the `fun_with_matlab.m` function is already suitable for code generation.



If your entry-point function is not suitable for code generation, the tool provides a report that lists the source files that contain unsupported features and functions. The report

also provides an indication of how much work you must do to make the MATLAB code ready for code generation. Before proposing data types, you must fix these issues. For more information, see “MATLAB Code Analysis”.

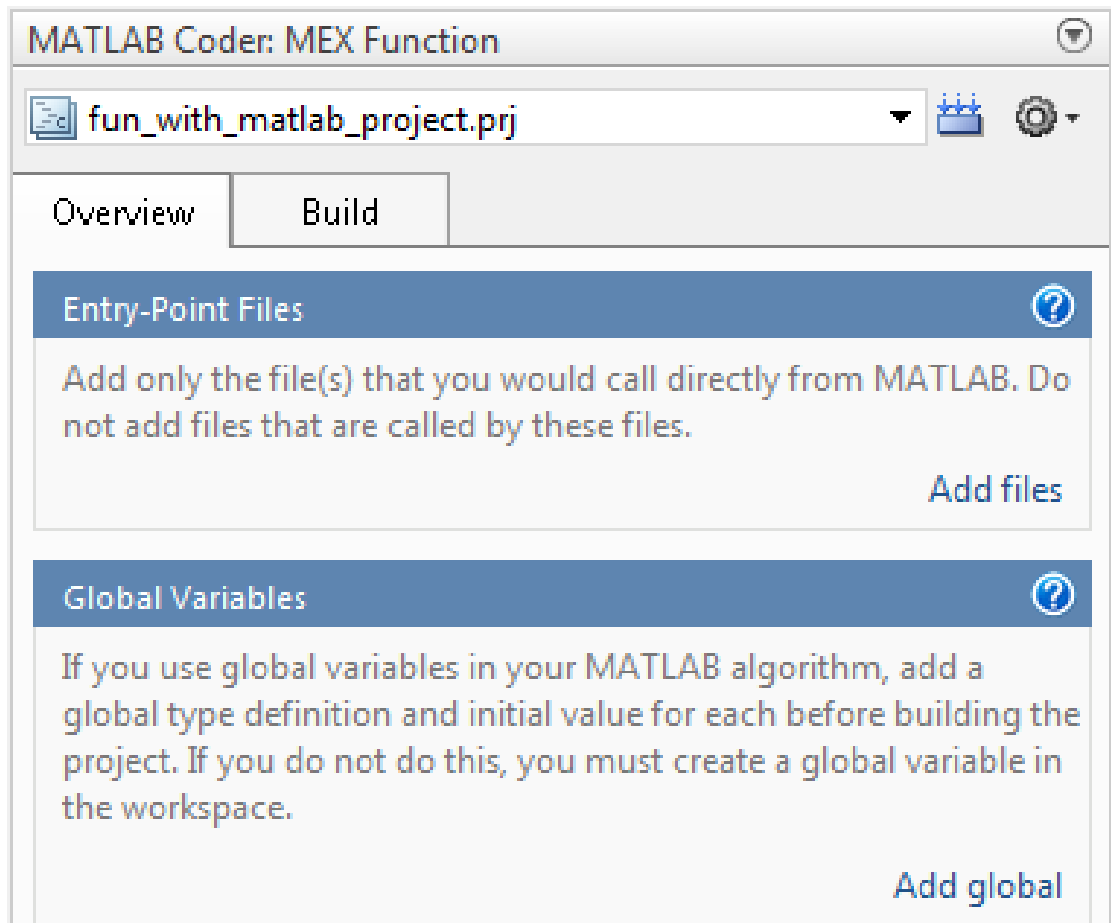
### **Create and set up a MATLAB Coder Project**

- 1** Navigate to the work folder that contains the file for this tutorial.
- 2** On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_matlab_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_matlab_project.prj
```

By default, the project opens in the MATLAB workspace.



- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_matlab.m` and then click **OK** to add the file to the project.

#### About the `fun_with_matlab_test` Script

The test script runs the `fun_with_matlab` function with three input signals: `chirp`, `step`, and `impulse`. The script then plots the results.

## Contents of fun\_with\_matlab\_test

```

% fun_with_matlab_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
    y(i,:) = fun_with_matlab(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'};
clf
for i = 1:size(x,1)
    subplot(size(x,1),1,i)
    plot(t,x(i,:),t,y(i,:))
    title(titles{i})
    legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

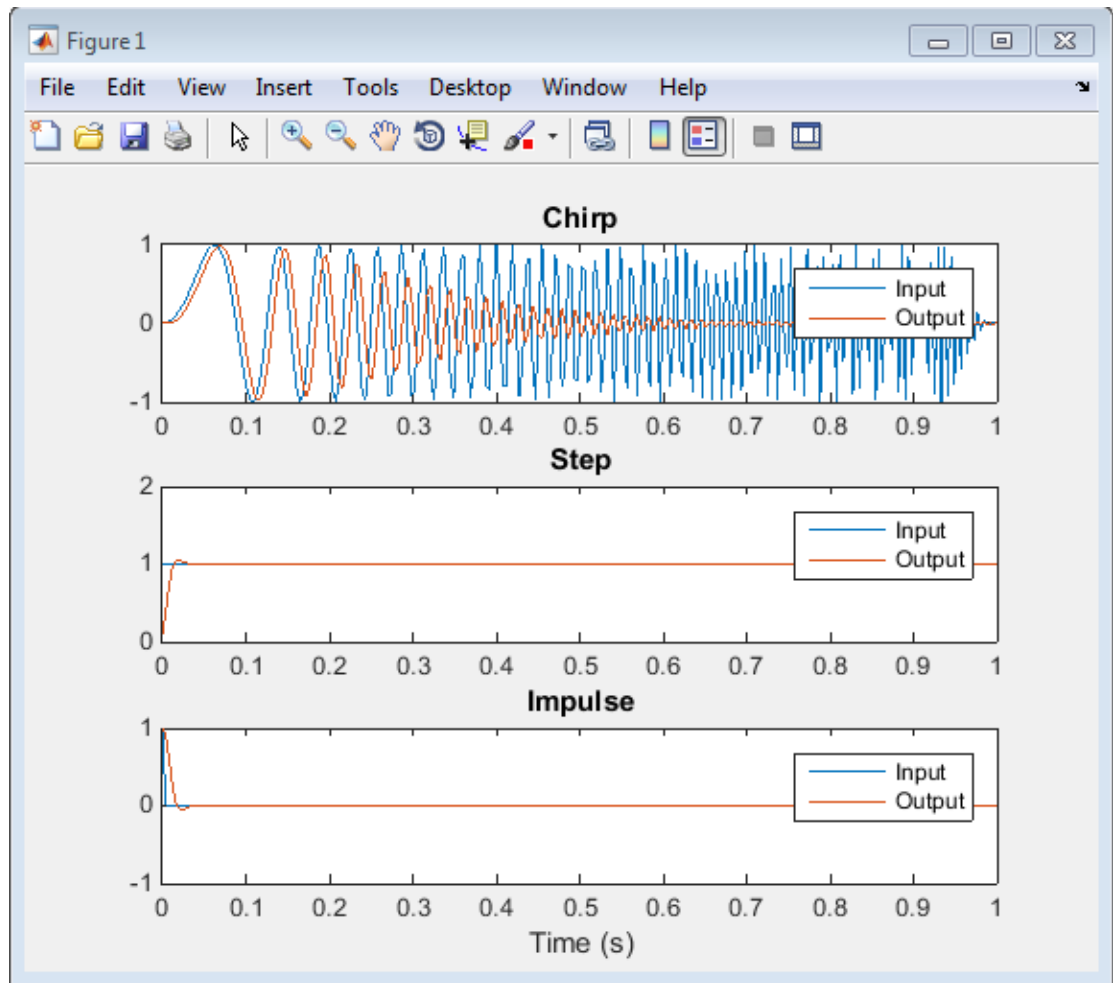
disp('Test complete.')

```

### Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add fun\_with\_matlab\_test as a test file and then click **Run**.

The test file runs and displays the outputs of the filter for each of the input signals.



MATLAB Coder determines the input types from the test file and then displays them in the Autodefine Input Types dialog box.

- 3 In this dialog box, click **Use These Types**.

MATLAB Coder sets the type of `x` to `double(1x256)`.

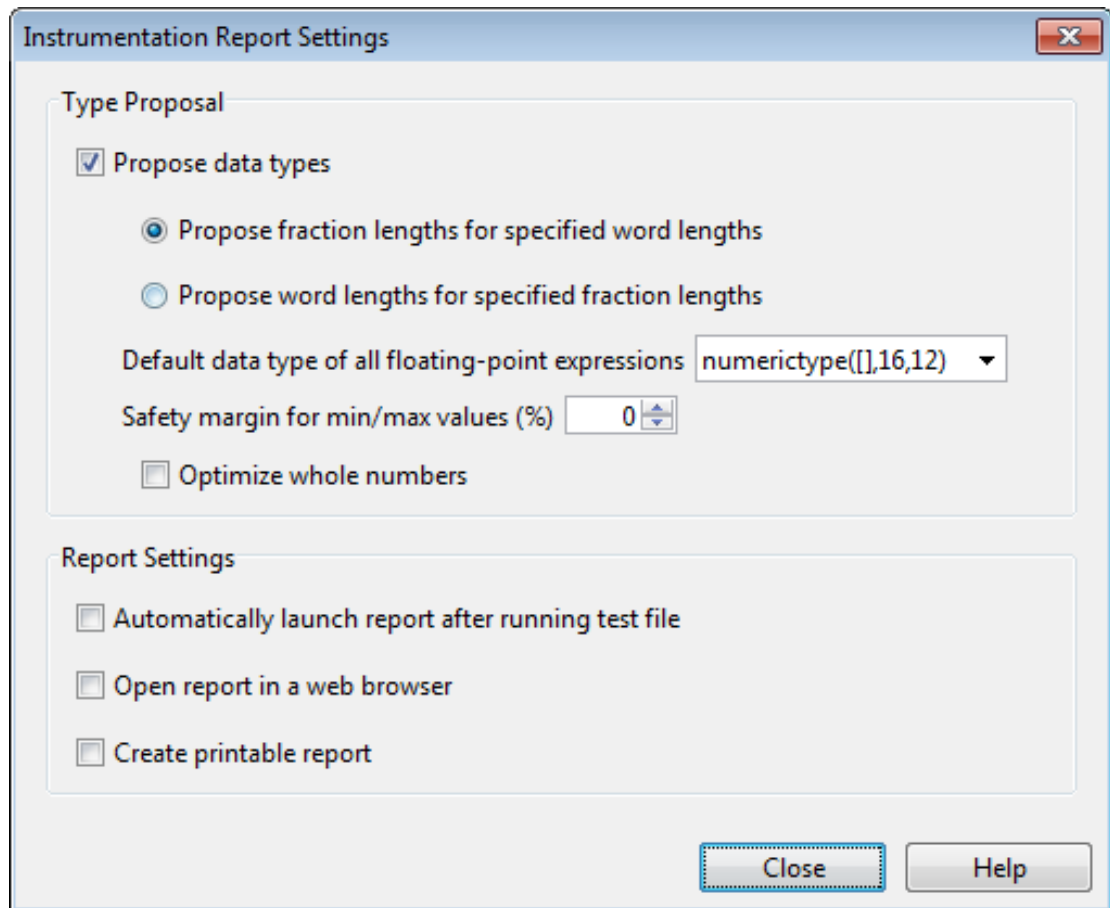
### **Build Instrumented MEX Function**

- 1** In the project, click the **Build** tab.
- 2** On the **Build** tab, set the **Output type** to Instrumented MEX Function.
- 3** Click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates an instrumented MEX function `fun_with_matlab_mex` in the current folder. It also provides a link to the report on the **Show Instrumentation Results** pane. In this report, you can view the types of variables in your MATLAB code.

### **View Data Type Proposal Settings**

- 1** On the **Show Instrumentation Results** pane, click the **Data type proposal and report settings** link.



This example uses the default data type proposal settings which propose fraction lengths for the specified word lengths. Because the MATLAB code is floating-point, the word length is specified by the **Default data type of all floating-point expressions** field. You can specify the `numerictype` signedness, word length and fraction length. Specifying `[]` for signedness instructs MATLAB Coder to choose the signedness based on simulation values. The default word length is 16. The default fraction length is 12.

For more information, see “Modify Data Type Proposal Settings”.



- 2 Close the dialog box.

## Run Simulation

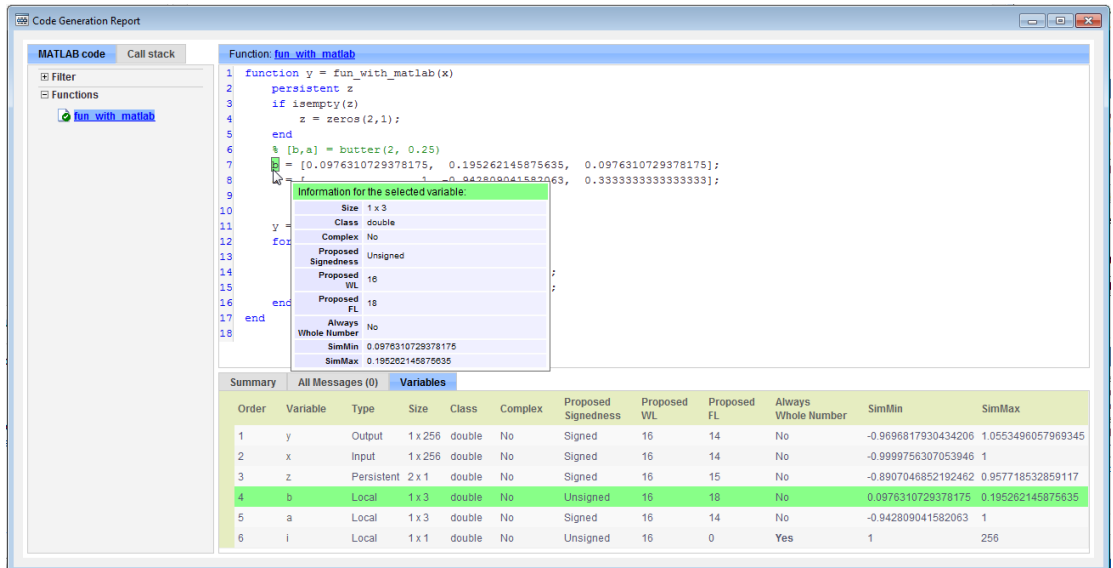
- 1 On the **Run Simulation** pane, verify that the test file is set to `fun_with_matlab_test` and that **Redirect entry-point calls to MEX function** is selected. That way, each call to `fun_with_matlab` is replaced with a call to the instrumented MEX function `fun_with_matlab_mex`.
- 2 On the **Run Simulation** pane, click **Run**.

The `fun_with_matlab_test` file runs and calls `fun_with_matlab_mex`. The outputs of the filters are displayed as before.

## View Code Generation Report

- 1 On the **Show Instrumentation Results** pane, click **View Report**.
- 2 In the **Code Generation Report**, click the **Variables** tab.

The report displays the simulation minimum and maximum values and the proposed data types.



The screenshot shows the Code Generation Report window for the function `fun_with_matlab`. The MATLAB code is displayed in the main pane, and the Variables tab is selected in the bottom pane. A tooltip provides information for the selected variable `b`.

**Information for the selected variable:**

Size	1 x 3
Class	double
Complex	No
Proposed Signedness	Unsigned
Proposed WL	16
Proposed FL	18
Always Whole Number	No
SimMin	0.0976310729378175
SimMax	0.195262145875635

**Summary Table:**

Order	Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax
1	y	Output	1 x 256	double	No	Signed	16	14	No	-0.9696817930434206	1.0553496057969345
2	x	Input	1 x 256	double	No	Signed	16	14	No	-0.9999756307053946	1
3	z	Persistent	2 x 1	double	No	Signed	16	15	No	-0.8907046852192462	0.957718532859117
4	b	Local	1 x 3	double	No	Unsigned	16	18	No	0.0976310729378175	0.195262145875635
5	a	Local	1 x 3	double	No	Signed	16	14	No	-0.942809041582063	1
6	i	Local	1 x 1	double	No	Unsigned	16	0	Yes	1	256

MATLAB Coder proposes data types with word length of 16 and fraction length optimized to avoid overflows.

### **Next Steps**

To learn how to apply the proposed data types to your entry-point MATLAB function and verify that the fixed-point version of your algorithm is functionally equivalent to your original MATLAB algorithm, see “Apply Fixed-Point Data Types”.

## Apply Fixed-Point Data Types

This example shows how to write a fixed-point version of your entry-point function using the data types proposed in “Propose Fixed-Point Data Types”.

This capability is not compatible with automatic fixed-point conversion. If you select **Convert to fixed point at build time**, you cannot build instrumented MEX functions.

You will learn how to:

- Use the proposed data types to create a fixed-point version of your entry-point function.
- Update your test file to call the fixed-point entry-point function.
- Verify that the fixed-point function is functionally equivalent to the original MATLAB algorithm.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\coder\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:  

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```
- 3 Copy the following files to your local working folder.

Type	Name	Description
Function code	<code>fun_with_matlab.m</code>	Entry-point MATLAB function
Test file	<code>fun_with_matlab_test.m</code>	MATLAB script that tests <code>fun_with_matlab.m</code>
Function code	<code>fun_with_fi.m</code>	Entry-point MATLAB function — fixed-point version of <code>fun_with_matlab</code> that uses data types proposed in “Propose Fixed-Point Data Types”
Test file	<code>fun_with_fi_test.m</code>	MATLAB script that runs both <code>fun_with_matlab</code> and <code>fun_with_fi</code> and compares the results

### The `fun_with_fi` Function

The `fun_with_fi` is a fixed-point version of the `fun_with_matlab` function that uses the data types proposed in “Propose Fixed-Point Data Types”.

Variable	Proposed Signedness	Proposed Word Length	Proposed Fraction Length
<code>y</code>	Signed	16	14
<code>x</code>	Signed	16	14
<code>z</code>	Signed	16	15
<code>a</code>	Unsigned	16	18
<code>b</code>	Signed	16	14
<code>i</code>	Unsigned	16	0

For example, in `fun_with_matlab`, variable `y` is defined as `y = zeros(size(x));`. In `fun_with_fi`, to specify that it is a signed fixed-point data type with a word length of 16 and a fraction length of 14:

```
y = fi(zeros(size(x)),1,16,14,'OverflowAction','Wrap','RoundingMethod','Floor');
```

For more information, see `fi`.

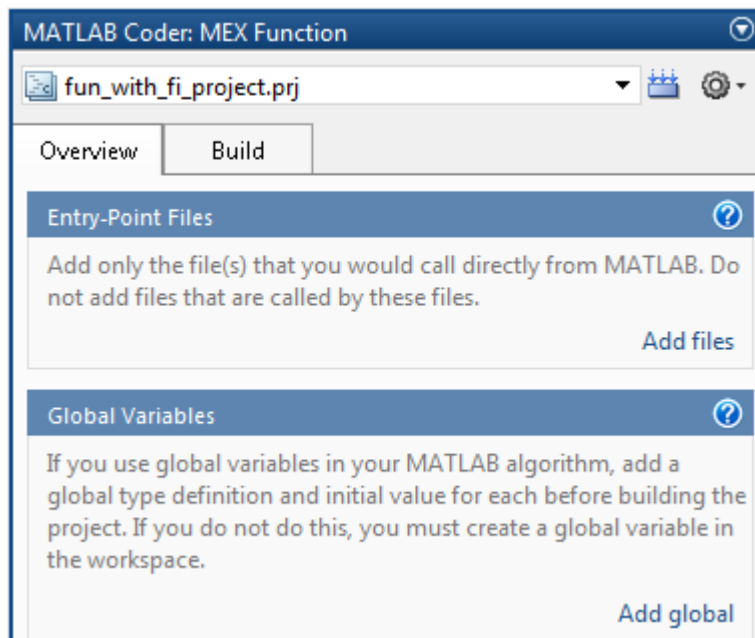
## Create and set up a MATLAB Coder Project

- 1 Navigate to the work folder that contains the file for this tutorial.
- 2 On the MATLAB **Apps** tab, select **MATLAB Coder** and then, in the **MATLAB Coder Project** dialog box, set **Name** to `fun_with_fi_project.prj`.

Alternatively, at the MATLAB command line, enter

```
coder -new fun_with_fi_project.prj
```

By default, the project opens in the MATLAB workspace.



- 3 On the project **Overview** tab, click the **Add files** link. Browse to the file `fun_with_fi.m`, and then click **OK** to add the file to the project.

## Define Input Types

- 1 On the project **Overview** tab, click the **Autodefine types** link.
- 2 In the Autodefine Input Types dialog box, add `fun_with_fi_test` as a test file, and then click **Run**.

The test file runs and plots the outputs of the filter. MATLAB Coder determines the input types from the test file and then displays them.

- 3 In the Autodefine Input Types dialog box, click **Use These Types** to accept the autodefined input type.

MATLAB Coder sets the type of `x` to `double(1x256)`.

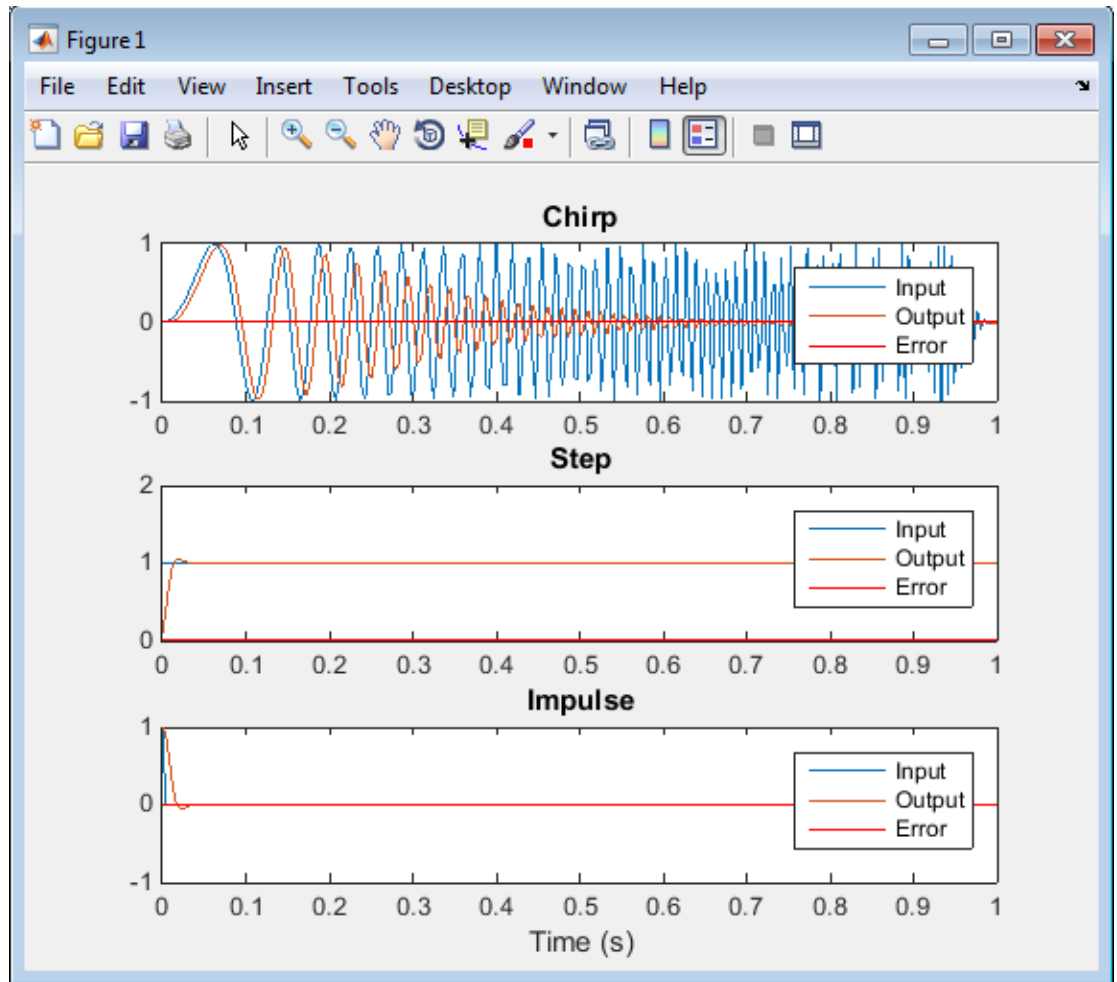
### The `fun_with_fi_test` Script

The `fun_with_fi_test` script runs the original floating-point MATLAB algorithm, `fun_with_matlab`, then runs the fixed-point version of the algorithm, `fun_with_fi`. The script then plots the outputs for the floating-point and fixed-point algorithms and the difference in results.

### Run Simulation

- 1 In the project, click the **Build** tab.
- 2 On the **Verification** pane, verify that the test file is set to `fun_with_fi_test`. Clear **Redirect entry-point calls to MEX function** so that the test file calls the MATLAB versions of the original and fixed-point algorithms.
- 3 On **Verification** pane, click **Run**.

The `fun_with_fi_test` file runs. The test file runs the original MATLAB algorithm and the fixed-point version, and plots the difference in their outputs.



- 4 Optionally, zoom in on each plot in turn to view the error (difference between the two versions of the algorithm). In this example, the errors are very small, on the order of  $10^{-3}$ . If the error is unacceptably large, refine the fixed-point data types.

## Modify Data Type Proposal Settings

When generating instrumented MEX functions, to modify data type proposal settings, on the project **Build** tab, on the **Show Instrumentation Results** pane, click **Data type proposal and report settings**.

Type Proposal Setting	Description
<b>Propose data types</b>	<p>Specify whether to propose data types based on simulation minimum and maximum values. You can view the proposed data types in the code generation report.</p> <p>Dependencies:</p> <ul style="list-style-type: none"> <li>• This parameter enables:               <ul style="list-style-type: none"> <li>• <b>Propose fraction lengths for specified word lengths</b></li> <li>• <b>Propose word lengths for specified fraction lengths</b></li> <li>• <b>Default data type of all floating-point expressions</b></li> <li>• <b>Safety margin for min/max values</b></li> <li>• <b>Optimize whole numbers</b></li> </ul> </li> </ul>
<b>Propose fraction lengths for specified word lengths</b>	<p>Select to propose fraction lengths for the word lengths specified in the code.</p> <p>Use simulation minimum and maximum information to propose fraction lengths for variables in your entry-point MATLAB function. MATLAB Coder proposes data types for variables that are scaled doubles and built-in data types only. For floating-point data types in your entry-point function, uses the word length and signedness specified in <b>Default data type of all floating-point expressions</b> to determine the optimal fraction lengths.</p> <p>Dependency:</p> <ul style="list-style-type: none"> <li>• Clearing <b>Propose data types</b> disables this parameter.</li> </ul>
<b>Propose word lengths for specified fraction lengths</b>	<p>Select to propose word lengths for the fraction lengths specified in the code.</p> <p>Use simulation minimum and maximum information to propose word lengths for variables in your entry-point MATLAB function. MATLAB Coder proposes data types for variables that are scaled doubles and built-in data types only. For floating-point data types in your entry-point function, uses the fraction</p>



Type Proposal Setting	Description										
	<p>length and signedness specified in <b>Default data type of all floating-point expressions</b> to determine the optimal word lengths.</p> <p>Dependency:</p> <ul style="list-style-type: none"> <li>• Clearing <b>Propose data types</b> disables this parameter.</li> </ul>										
<p><b>Default data type of all floating-point expressions</b></p>	<p>Specify the default data type to use for floating-point expressions in your entry-point MATLAB function.</p> <p>MATLAB Coder uses this default data type to change the floating-point data types in the code to fixed point.</p> <p>Dependency:</p> <ul style="list-style-type: none"> <li>• Clearing <b>Propose data types</b> disables this parameter.</li> </ul> <table border="1" data-bbox="359 795 1337 1364"> <tbody> <tr> <td data-bbox="359 795 698 1062"> <p><code>numerictype([ ],16,12)</code> (Default)</p> </td> <td data-bbox="704 795 1337 1062"> <p>Set the default data type for floating-point signals to the fixed-point data type specified by <code>numerictype</code>. You can modify the parameters provided to <code>numerictype</code> to specify signedness, word length, and fraction length.</p> <p>Specifying <code>[ ]</code> for signedness instructs MATLAB Coder to choose the appropriate signedness.</p> </td> </tr> <tr> <td data-bbox="359 1065 698 1138"> <p>Remain floating-point</p> </td> <td data-bbox="704 1065 1337 1138"> <p>Do not change the data type of floating-point signals.</p> </td> </tr> <tr> <td data-bbox="359 1142 698 1215"> <p><code>int8</code></p> </td> <td data-bbox="704 1142 1337 1215"> <p>Set the default data type for floating-point signals to <code>int8</code>.</p> </td> </tr> <tr> <td data-bbox="359 1218 698 1291"> <p><code>int16</code></p> </td> <td data-bbox="704 1218 1337 1291"> <p>Set the default data type for floating-point signals to <code>int16</code>.</p> </td> </tr> <tr> <td data-bbox="359 1295 698 1364"> <p><code>int32</code></p> </td> <td data-bbox="704 1295 1337 1364"> <p>Set the default data type for floating-point signals to <code>int32</code>.</p> </td> </tr> </tbody> </table>	<p><code>numerictype([ ],16,12)</code> (Default)</p>	<p>Set the default data type for floating-point signals to the fixed-point data type specified by <code>numerictype</code>. You can modify the parameters provided to <code>numerictype</code> to specify signedness, word length, and fraction length.</p> <p>Specifying <code>[ ]</code> for signedness instructs MATLAB Coder to choose the appropriate signedness.</p>	<p>Remain floating-point</p>	<p>Do not change the data type of floating-point signals.</p>	<p><code>int8</code></p>	<p>Set the default data type for floating-point signals to <code>int8</code>.</p>	<p><code>int16</code></p>	<p>Set the default data type for floating-point signals to <code>int16</code>.</p>	<p><code>int32</code></p>	<p>Set the default data type for floating-point signals to <code>int32</code>.</p>
<p><code>numerictype([ ],16,12)</code> (Default)</p>	<p>Set the default data type for floating-point signals to the fixed-point data type specified by <code>numerictype</code>. You can modify the parameters provided to <code>numerictype</code> to specify signedness, word length, and fraction length.</p> <p>Specifying <code>[ ]</code> for signedness instructs MATLAB Coder to choose the appropriate signedness.</p>										
<p>Remain floating-point</p>	<p>Do not change the data type of floating-point signals.</p>										
<p><code>int8</code></p>	<p>Set the default data type for floating-point signals to <code>int8</code>.</p>										
<p><code>int16</code></p>	<p>Set the default data type for floating-point signals to <code>int16</code>.</p>										
<p><code>int32</code></p>	<p>Set the default data type for floating-point signals to <code>int32</code>.</p>										

Type Proposal Setting	Description
<b>Safety margin for min/max values</b>	<p>Specify safety factor for simulation minimum and maximum values.</p> <p>The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of 55 specifies that you want a range at least 55 percent larger. A value of -15 specifies that a range up to 15 percent smaller is acceptable.</p> <p>Dependency:</p> <ul style="list-style-type: none"><li>• Clearing <b>Propose data types</b> disables this parameter.</li></ul>
<b>Optimize whole numbers</b>	<p>Specify to use integer scaling for variables that were whole numbers during simulation.</p> <p>Dependency:</p> <ul style="list-style-type: none"><li>• Clearing <b>Propose data types</b> disables this parameter.</li></ul>

## Modify Instrumentation Report Settings

When generating instrumented MEX functions, to modify instrumentation report settings, on the project **Build** tab, on the **Show Instrumentation Results** pane, click **Data type proposal and report settings**.

Report Setting	Description
<b>Automatically launch report after running test file</b>	Specify whether to automatically display the report after running the test file.
<b>Open report in a web browser</b>	Specify whether to open the report in a Web browser. Enabling this option allows you to open multiple reports simultaneously.
<b>Create printable report</b>	Specify whether to create a printable report.

## Automated Fixed-Point Conversion

### In this section...

“License Requirements” on page 14-86

“Automated Fixed-Point Conversion Capabilities” on page 14-86

“Code Coverage” on page 14-88

“Proposing Data Types” on page 14-91

“Locking Proposed Data Types” on page 14-93

“Viewing Functions” on page 14-93

“Viewing Variables” on page 14-94

“Histogram” on page 14-100

“Function Replacements” on page 14-102

“Validating Types” on page 14-103

“Testing Numerics” on page 14-103

“Detecting Overflows” on page 14-104

### License Requirements

Fixed-point conversion requires the following licenses:

- Fixed-Point Designer
- MATLAB Coder

### Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Conversion tool in MATLAB Coder projects. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see “Locking Proposed Data Types” on page 14-93.

For a list of supported MATLAB features and functions, see “MATLAB Language Features Supported for Automated Fixed-Point Conversion”.

The screenshot shows the 'Fixed-Point Conversion - fun\_with\_matlab.prj' window. The top toolbar includes 'Run Simulation', 'Compute Derived Ranges', 'Advanced' (with 'Propose fraction lengths' and 'Propose word lengths' options), 'Validate Types', 'Test Numerics', and 'Help'. The 'Default word length' is set to 16. Below the toolbar, a yellow banner reads: 'To compute proposed fixed-point types for variables, use Run Simulation, Compute Derived Ranges, or both.' The main area contains MATLAB code for a function named 'fun\_with\_matlab'. Below the code is a table with two tabs: 'Variables' and 'Function Replacements'. The 'Variables' tab is active, showing a table with columns: Variable, Type, Sim Min, Sim Max, Static Min, Static Max, Whole Number, and Proposed Type.

```

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

```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
x	1×256 double					No	
▲ Output							
y	1×256 double					No	
▲ Persistent							
z	2×1 double					No	
▲ Local							
a	1×3 double					No	
b	1×3 double					No	
i	double					No	

During fixed-point conversion, you can:

- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- 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.
- Detect overflows.

## Code Coverage

By default, the Fixed-Point Conversion tool shows code coverage results. Your test files should exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results helps you verify that your test file is exercising the algorithm adequately. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. If you simulate multiple test files in one run, the tool displays cumulative coverage. However, if you specify multiple test files but run them one at a time, the tool displays the coverage of the file that ran last.

Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage might speed up simulation. To turn off code coverage, in the Fixed-Point Conversion tool:

- 1 Click **Run Simulation**.
- 2 Clear **Show code coverage**.

The tool covers basic MATLAB control constructs and shows statement coverage for basic blocks of code. The tool displays a color-coded coverage bar to the left of the code.

Coverage Bar Color	How Often Code is Executed During Test File Simulation
Dark green	Always
Light green	Sometimes
Orange	Once
Red	Never

```

16 if isempty(current_state)
17     current_state = S1;
18 end
19
20 % switch to new state based on the value state register
21 switch (current_state)
22
23     case S1,
24
25         % value of output 'Z' depends both on state and inputs
26         if (A)
27             Z = true;
28             current_state = S1;
29         else
30             Z = false;
31             current_state = S2;
32         end
33
34     case S2,
35
36         if (A)
37             Z = false;
38             current_state = S1;
39         else
40             Z = true;
41             current_state = S2;
42         end
43
44     case S3,
45
46         if (A)
47             Z = false;
48             current_state = S2;
49         else
50             Z = true;
51             current_state = S3;
52         end
53

```

When you position your cursor over the coverage bar, the color highlighting extends over the code and the tool displays more information about how often the code is executed. For MATLAB constructs that affect control flow (if-elseif-else, switch-case, for-continue-break, return), it displays statement coverage as a percentage coverage for basic blocks inside these constructs.

```

16 if isempty(current_state)
17     current_state = S1;
18 end
19
20 % switch to new state based on the value state register
21 switch (current_state)
22
23     case S1,
24
25         % value of output 'Z' depends both on state and inputs
26         if (A)
27             Z = true;
28             current_state = S1;
29         else
30             Z = false;
31             current_state = S2;
32         end
33
34     case S2,
35
36         if (A)
37             Z = false;
38             current_state = S1;
39         else
40             Z = true;
41             current_state = S2;
42         end
43
44     case S3,
45
46         if (A)
47             Z = false;
48             current_state = S2;
49         else
50             Z = true;
51             current_state = S3;
52         end
53

```

Execution statistics from the screenshot:

- Line 16: Executed once
- Line 23: Reached 58% of the time
- Line 26: Reached 33% of the time
- Line 29: Reached 25% of the time
- Line 33: Reached 58% of the time
- Line 34: Reached 42% of the time
- Line 36: Reached 17% of the time
- Line 39: Reached 25% of the time
- Line 43: Reached 42% of the time
- Line 44: Not reached

To verify that your test file is testing your algorithm over the intended operating range, review the code coverage results and take action as described in the following table.

Coverage Bar Color	Action Required
Dark green	None
Light green	Review percentage coverage and verify that it is reasonable based on your algorithm. If there are areas of code that you expect to be executed more frequently, modify your test file or add more test files to increase coverage.



Coverage Bar Color	Action Required
Orange	This is expected behavior for initialization code, for example, the initialization of persistent variables. For other cases, verify that this behavior is reasonable for your algorithm. If there are areas of code that you expect to be executed more frequently, modify your test file or add more test files to increase coverage.
Red	If the code that is not executed is an error condition, this is acceptable behavior. If the code should be executed, modify the test file or add another test file to extend coverage. If the code is written conservatively and has upper and lower boundary limits and you cannot modify the test file to reach this code, add static minimum and maximum values (see “Computing Derived Ranges”).

## Proposing Data Types

The Fixed-Point Conversion tool proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data, or both. If you run a simulation and compute derived ranges, the conversion tool merges the simulation and derived ranges.

---

**Note:** You cannot propose data types based on derived ranges for MATLAB classes.

---

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see “Locking Proposed Data Types” on page 14-93.

## Running a Simulation

When you open the Fixed-Point Conversion tool, the tool generates an instrumented MEX function for your entry-point MATLAB file. If the build completes without errors, the tool displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the tool provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the

code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the tool displays them on the **Function Replacements** tab. See “Function Replacements” on page 14-102.

Before running a simulation, specify the test file or files that you want to run. When you run a simulation, the tool runs the test file, calling the instrumented MEX function. If you modify the MATLAB design code, the tool automatically generates an updated MEX function before running a test file.

If the test file runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If you manually enter static ranges for a variable, the manually-entered ranges take precedence over the simulation ranges. If you manually modify the proposed types by typing or using the histogram, the data types are locked so that the tool cannot modify them.

If the test file fails, the errors are displayed on the **Simulation Output** tab.

Test files should exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test file covers the operating range of the algorithm with the desired accuracy. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the conversion tool merges the simulation results.

Optionally, you can select to log data for histograms. After running a simulation, you can view the histogram for each variable. For more information, see “Histogram” on page 14-100.

### Computing Derived Ranges

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values or proposed data types for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can manually enter ranges or promote simulation ranges to use as static ranges. Manually-entered static ranges always take precedence over simulation ranges.

If you know what data type your hardware target uses, set the proposed data types to match this type. Manually-entered data types are locked so that the tool cannot modify

them. The tool uses these data types to calculate the input minimum and maximum values and to derive ranges for other variables. For more information, see “Locking Proposed Data Types” on page 14-93.

When you select **Compute Derived Ranges**, the tool runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces +/- Inf derived ranges, consider defining ranges for all persistent variables.

Optionally, you can select **Quick derived range analysis**. With this option, the conversion tool performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. The tool aborts the analysis when the timeout is reached.

## Locking Proposed Data Types

You can lock proposed data types against changes by the Fixed-Point Conversion tool using one of the following methods:

- Manually setting a proposed data type in the Fixed-Point Conversion tool.
- Right-clicking a type proposed by the tool and selecting **Lock computed value**.

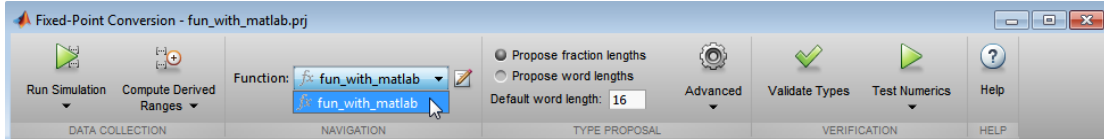
The tool displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- Manually overwriting it.
- Right-clicking it and selecting **Undo changes**. This action unlocks only the selected type.
- Right-clicking and selecting **Undo changes for all variables**. This action unlocks all locked proposed types.

## Viewing Functions

You can view a list of functions in your project on the **Navigation** pane. This list also includes function specializations and class methods. When you select a function from the

list, the MATLAB code for that function or class method is displayed in the Fixed-Point Conversion tool code window.



After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the conversion retains the original function name in the fixed-point filename and appends the fixed-point suffix. For example, the fixed-point version of `fun_with_matlab.m` is `fun_with_matlab_fixpt.m`.

## Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- **Static Min** and **Static Max** — The static minimum and maximum values.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

When you compute derived ranges, the Fixed-Point Conversion tool runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

The Fixed-Point Conversion tool determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

Because the tool does not apply data types to expressions, it does not display proposed types for them. Instead, it displays their original data types.

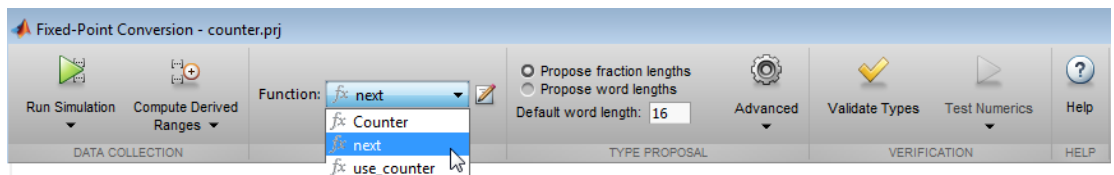
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

You can use `Ctrl+F` to search for variables in the MATLAB code and on the **Variables** tab. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

## Viewing Information for MATLAB Classes

The tool displays:

- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Function** list in the Navigation bar to select which class or class method to view.



```

1 classdef Counter < handle
2     properties
3         Value;
4     end
5
6     methods(Static)
7         function t = MAX_VALUE()
8             t = 128;
9         end
10        end
11
12    methods
13        function this = Counter()
14            this.Value = 0;
15        end
16        function out = next(this)
17            out = this.Value;
18            if this.Value == this.MAX_VALUE
19                this.Value = 0;
20            else
21                this.Value = this.Value + 1;
22            end
23        end
24    end
25 end
  
```

- Information about MATLAB classes on the **Variables** tab.

Variables		Function Replacements		Simulation Output			
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
this	Counter		Unknown	Unknown		No	
this.Value	double		0	1024		Yes	numerictype(0, 11, 0)
▲ Output							
v	double		0	1024		Yes	numerictype(0, 11, 0)

## Specializations

If a function is specialized, the tool lists each specialization and numbers them sequentially. For example, consider a function, `dut`, that calls subfunctions, `foo` and `bar`, multiple times with different input types.

```
function y = dut(u, v)
```

```
tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

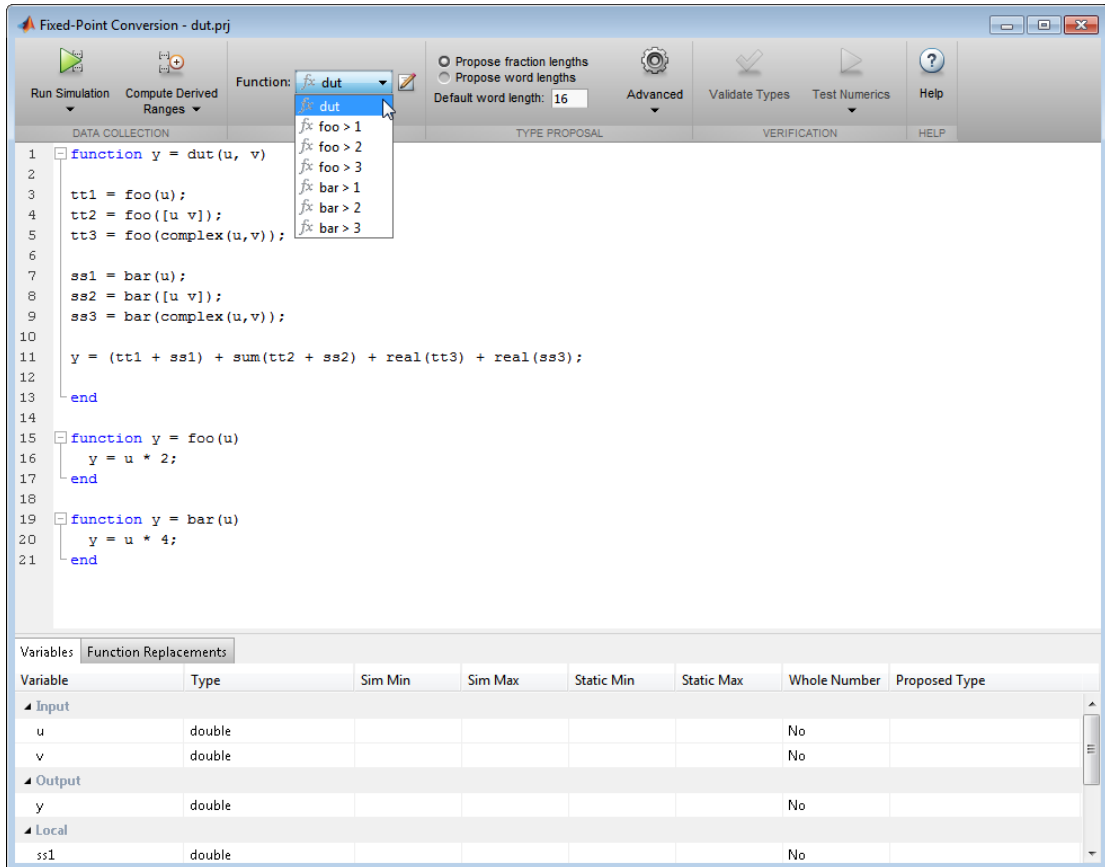
ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
    y = u * 2;
end

function y = bar(u)
    y = u * 4;
end
```



If you select a specialization, the app displays only the variables used by the specialization.



The screenshot shows the 'Fixed-Point Conversion - dut.prj' window. The top toolbar includes 'Run Simulation', 'Compute Derived Ranges', 'Function: f00 > 1', 'Propose fraction lengths', 'Propose word lengths', 'Default word length: 16', 'Advanced', 'Validate Types', 'Test Numerics', and 'Help'. The main area displays the following source code:

```

1 function y = dut(u, v)
2
3     tt1 = foo(u);
4     tt2 = foo([u v]);
5     tt3 = foo(complex(u,v));
6
7     ss1 = bar(u);
8     ss2 = bar([u v]);
9     ss3 = bar(complex(u,v));
10
11    y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13    end
14
15    function y = foo(u)
16        y = u * 2;
17    end
18
19    function y = bar(u)
20        y = u * 4;
21    end

```

Below the code is a 'Function Replacements' table:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
▲ Input							
u	double					No	
▲ Output							
y	double					No	

In the generated fixed-point code, the number of each fixed-point specialization matches the number in the Source Code 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 `f00 > 1` is named `f00_s1`.

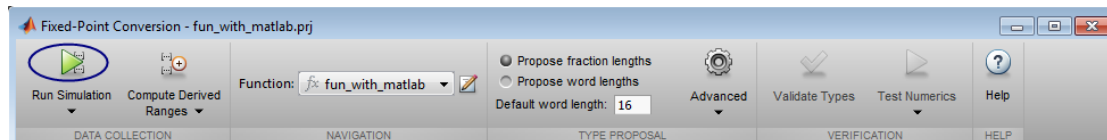
```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  %           Generated by MATLAB 8.4 and Fixed-Point Designer 4.3
4  %
5  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6  %%codegen
7  function y = dut_fixpt(u, v)
8
9
10
11  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full
12
13  tt1 = fi(foo_s1(u), 0, 5, 0, fm);
14  tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
15  tt3 = fi(foo_s3(complex(u,v)), 0, 6, 0, fm);
16
17  ss1 = fi(bar_s1(u), 0, 6, 0, fm);
18  ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
19  ss3 = fi(bar_s3(complex(u,v)), 0, 7, 0, fm);
20
21  y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);
22
23  end
24
25  function y = foo_s1(u)
26  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu
27  y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
28  end
29
30  function y = foo_s2(u)
31  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Fu
32  y = fi(u * fi(2, 0, 2, 0, fm), 0, 6, 0, fm);
33  end
34

```

## Histogram

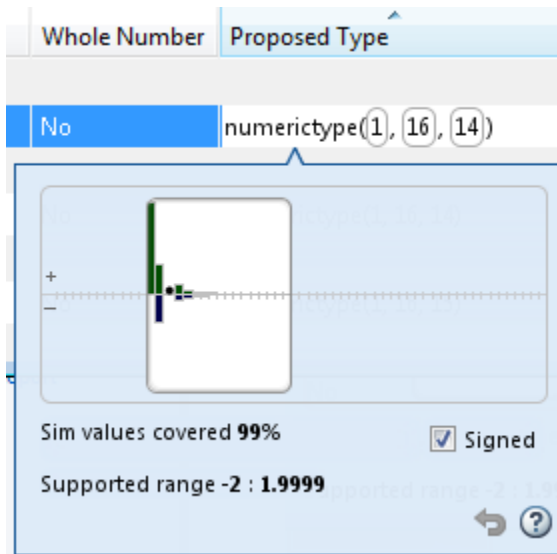
To log data for histograms, in the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the Run Simulation button.



After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along

the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1,16,14)`.




You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click .

## Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the tool lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.

Variables		Function Replacements		Simulation Output		
Enter a function to replace						
					Custom Function	+ -
Function or Operator	Replacement					
<ul style="list-style-type: none"> <li>Custom Function</li> </ul>	Function Name					
foo	foo_fixedpoint					
<ul style="list-style-type: none"> <li>Lookup Table</li> </ul>	Interpolation Method	Design Min	Design Max	Number of Points		
exp	None	Auto	Auto	1000		

You can add and remove function replacements from this list. If you enter a function replacements for a function, the replacement function is used when you build the

project. If you do not enter a replacement, the tool uses the type specified in the original MATLAB code for the function.

---

**Note:** Using this table, you can replace the names of the functions but you cannot replace argument patterns.

---

## Validating Types

Selecting **Validate Types** validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

## Testing Numerics

After validating the proposed fixed-point data types, select **Test Numerics** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test file to define inputs or run a simulation, the tool uses this test file to test numerics. Optionally, you can add test files and select to run more than one test file. The tool compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For non-scalar outputs, only the error information is shown.

By default, the Fixed-Point Conversion tool runs the test files that you added and selected for running the simulation. You can add test files and select to run more than one test file to test numerics.

If the numerical results do not meet your desired accuracy after fixed-point simulation, modify fixed-point data type settings and repeat the type validation and numerical










testing steps. You might have to iterate through these steps multiple times to achieve the desired results.

## Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion tool runs the simulation using scaled double versions of the proposed fixed-point types. Because scaled doubles store their data in double-precision floating-point, they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type. .

If the tool detects overflows, on its Overflow tab, it provides:

- A list of variables and expressions that overflowed
- Information on how much each variable overflowed
- A link to the variables or expressions in the code window

Variables	Function Replacements	Overflows	
	Function	Line	Description
	overflow_fixpt	7	Overflow error in expression 'x'.
	overflow_fixpt	7	Overflow error in expression 'y'.
	overflow_fixpt	10	Overflow error in expression 'z'.
	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
	overflow_fixpt	10	Overflow error in expression 'x'.
	overflow_fixpt	10	Overflow error in expression 'x*y'.
	overflow_fixpt	10	Overflow error in expression 'y'.
	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the tool also provides overflow information for these expressions.

### See Also

“Detect Overflows”

## Instrumented MEX Functions

### In this section...

“Generating Instrumented MEX Functions” on page 14-105

“Merging Instrumentation Results” on page 14-105

“Clearing Instrumentation Results” on page 14-106

“Redirecting Entry-Point Calls to MEX Function” on page 14-106

“Proposing Fraction Lengths” on page 14-106

“Proposing Word Lengths” on page 14-106

## Generating Instrumented MEX Functions

---

**Note:** This capability is not compatible with automatic fixed-point conversion. If you select `Convert to fixed point at build time`, you cannot build instrumented MEX functions.

---

Generating an instrumented MEX function for your MATLAB function enables instrumentation for logging minimum and maximum values of named and intermediate variables in your algorithm. It also enables instrumentation for `log2` histograms of named, intermediate and expression values.

When you run the instrumented MEX function, the instrumentation report provides fixed-point data type proposals based on the simulation range data. You can use this information to convert your MATLAB code to fixed point by hand. For more information, see “Propose Fixed-Point Data Types” on page 14-67

## Merging Instrumentation Results

When generating instrumented MEX functions, use the **Merge instrumentation results from multiple simulations** option to specify whether to merge new simulation minimum and maximum results with existing simulation results. Merging instrumentation results allows you to collect complete range information from multiple test files.

## Clearing Instrumentation Results

When generating instrumented MEX functions, click the **Clear instrumentation results** button to clear instrumentation results from previous runs.

## Redirecting Entry-Point Calls to MEX Function

By default, with the **Redirect entry-point calls to MEX function** option selected, the MATLAB Coder software automatically redirects calls to your MATLAB algorithm in the test file to calls to the generated MEX function. The generated MEX function must be in the same folder as the entry-point functions.

If your test file already calls the MEX function, or you want to run the test file to test the original MATLAB algorithm, clear this option.

## Proposing Fraction Lengths

When you simulate an instrumented MEX function, if you select to propose fraction lengths for the word lengths specified in the code, MATLAB Coder uses simulation minimum and maximum information and proposes fraction lengths for variables in your entry-point MATLAB function. For floating-point data types in your entry-point function, MATLAB Coder uses the word length and signedness specified in **Default data type of all floating-point expressions** to determine the optimal fraction lengths.

Optionally, specify a safety margin to use when proposing fraction lengths. For more information, see “Modify Data Type Proposal Settings” on page 14-82.

## Proposing Word Lengths

When you simulate an instrumented MEX function, if you select to propose word lengths for the fraction lengths specified in the code, MATLAB Coder uses simulation minimum and maximum information and proposes word lengths for variables in your entry-point MATLAB function. For floating-point data types in your entry-point function, MATLAB Coder uses the fraction length and signedness specified in **Default data type of all floating-point expressions** to determine the optimal word lengths.

Optionally, specify a safety margin to use when proposing word lengths. For more information, see “Modify Data Type Proposal Settings” on page 14-82.



## Convert Fixed-Point Conversion Project to MATLAB Scripts

This example shows how to convert a MATLAB Coder project to MATLAB scripts when the project 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.

### Prerequisites

This example uses the following files:

- Project file `fun_with_matlab.prj`
- Entry-point file `fun_with_matlab.m`
- Test bench file `fun_with_matlab_test.m`
- Generated fixed-point MATLAB file `fun_with_matlab_fixpt.m`

To obtain these files, complete the example “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-4, including these steps:

- 1 Complete the **Test Numerics** step of the fixed-point conversion process.
- 2 Configure the project to build a C/C++ static library.

### Generate the Scripts

- 1 Change to the folder that contains the project file `fun_with_matlab.prj` and the entry-point function file `fun_with_matlab.m`.
- 2 At the command line, use the `-tocode` option of the `coder` command to convert the project to the scripts. Use the `-script` option to specify the file name for the scripts.

```
coder -tocode fun_with_matlab_project -script fun_with_matlab_script.m
```

The `coder` command generates two scripts in the current folder:

`fun_with_matlab_script.m` contains the MATLAB commands to:

- Create a code configuration object that has the same settings as the project.
- Run the `codegen` command to convert the fixed-point MATLAB function `fun_with_matlab_fixpt` to a fixed-point C function.

`fun_with_matlab_script_fixpt.m` contains the MATLAB commands to:

- Create a floating-point to fixed-point conversion configuration object that has the same fixed-point conversion settings as the project.
- Run the `codegen` command to convert the MATLAB function `fun_with_matlab` to the fixed-point MATLAB function `fun_with_matlab_fixpt`.

The suffix in the script file name is the generated fixed-point file name suffix specified by the project file. In this example, the suffix is the default value `_fixpt`.

The `coder` command overwrites existing files that have the same names as the generated scripts. If you omit the `-script` option, the `coder` command writes the scripts to the Command Window.

### Run Script That Generates Fixed-Point C Code

To run the script that generates fixed-point C code from fixed-point MATLAB code, the fixed-point MATLAB function specified in the script must be available.

- 1 Make sure that the fixed-point MATLAB function `fun_with_matlab_fixpt.m` is on the search path.

```
addpath c:\coder\fun_with_matlab\codegen\fun_with_matlab\fixpt
```

- 2 Run the script:

```
fun_with_matlab_script
```

The code generation software generates a C static library with the name `fun_with_matlab_fixpt` in the folder `codegen\lib\fun_with_matlab_fixpt`. The variables `cfg` and `ARGS` appear in the base workspace.

### Run Script That Generates Fixed-Point MATLAB Code

If you do not have the fixed-point MATLAB function, or if you want to regenerate it, use the script that generates the fixed-point MATLAB function from the floating-point MATLAB function.

- 1 Make sure that the current folder contains the entry-point function `fun_with_matlab.m`, and the test bench file `fun_with_matlab_test.m`.

**2** Run the script:

```
fun_with_matlab_script_fixpt
```

The code generation software generates `fun_with_matlab_fixpt.m` in the folder `codegen\fun_with_matlab\fixpt`. The variables `cfg` and `ARGS` appear in the base workspace.

**See Also**

`coder.FixptConfig` | `codegen` | `coder`

**Related Examples**

- “Convert MATLAB Code to Fixed-Point C Code”
- “Propose Fixed-Point Data Types Based on Simulation Ranges”
- “Convert MATLAB Coder Project to MATLAB Script”

## Generated Fixed-Point Code

### In this section...

“Location of Generated Fixed-Point Files” on page 14-110

“Minimizing `fi`-casts to Improve Code Readability” on page 14-111

“Avoiding Overflows in the Generated Fixed-Point Code” on page 14-111

“Controlling Bit Growth” on page 14-112

“Avoiding Loss of Range or Precision” on page 14-112

“Handling Non-Constant `mpower` Exponents” on page 14-114

### Location of Generated Fixed-Point Files

By default, the fixed-point conversion process generates files in a folder named `codegen/fcn_name/fixpt` in your local working folder. `fcn_name` is the name of the MATLAB function that you are converting to fixed point.

Filename	Description
<code>fcn_name_fixpt.m</code>	Generated fixed-point MATLAB code.  To integrate this fixed-point code into a larger application, consider generating a MEX-function for the function and calling this MEX-function in place of the original MATLAB code.
<code>fcn_name_fixpt_exVal.mat</code>	MAT-file containing: <ul style="list-style-type: none"> <li>• A structure for the input arguments.</li> <li>• The name of the fixed-point file.</li> </ul>
<code>fcn_name_fixpt_report.html</code>	Link to the type proposal report that displays the generated fixed-point code and the proposed type information.
<code>fcn_name_report.html</code>	Link to the type proposal report that displays the original MATLAB code and the proposed type information.

Filename	Description
fcn_name_wrapper_fixpt.m	File that converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during the conversion step. These fixed-point values are fed into the converted fixed-point function, fcn_name_fixpt.

## Minimizing `fi`-casts to Improve Code Readability

The conversion process tries to reduce the number of `fi`-casts by analyzing the floating-point code. If an arithmetic operation is comprised of only compile-time constants, the conversion process does not cast the operands to fixed point individually. Instead, it casts the entire expression to fixed point.

For example, here is the fixed-point code generated for the constant expression  $x = 1/\sqrt{2}$  when the selected word length is 14.

Original MATLAB Code	Generated Fixed-Point Code
<code>x = 1/sqrt(2);</code>	<code>x = fi(1/sqrt(2), 0, 14, 14, fm);</code>  <code>fm is the local fimath.</code>

## Avoiding Overflows in the Generated Fixed-Point Code

The conversion process avoids overflows by:

- Using full-precision arithmetic unless you specify otherwise.
- Avoiding arithmetic operations that involve double and `fi` data types. Otherwise, if the word length of the `fi` data type is not able to represent the value in the double constant expression, overflows occur.
- Avoiding overflows when adding and subtracting non fixed-point variables and fixed-point variables.

The fixed-point conversion process casts non-`fi` expressions to the corresponding `fi` type.

For example, consider the following MATLAB algorithm.

```

% A = 5;
% B = ones(300, 1)
function y = fi_plus_non_fi(A, B)
    % '1024' is non-fi, cast it
    y = A + 1024;
    % 'size(B, 1)*length(A)' is a non-fi, cast it
    y = A + size(B, 1)*length(A);
end

```

The generated fixed-point code is:

```

%#codegen
% A = 5;
% B = ones(300, 1)
function y = fi_plus_non_fi_fixpt(A, B)
    % '1024' is non-fi, cast it
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
               'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
               'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(A + fi(1024, 0, 11, 0, fm), 0, 11, 0, fm);
    % 'size(B, 1)*length(A)' is a non-fi, cast it
    y(:) = A + fi(size(B, fi(1, 0, 1, 0, fm))*length(A), 0, 9, 0, fm);
end

```

## Controlling Bit Growth

The conversion process controls bit growth by using subscripted assignments, that is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. Using subscripted assignment keeps fixed-point variables fixed point rather than inadvertently turning them into doubles. Maintaining the fixed-point type reduces the number of type declarations in the generated code. Subscripted assignment also prevents bit growth which is useful when you want to maintain a particular data type for the output.

## Avoiding Loss of Range or Precision

### Avoiding 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.

For example, consider the following MATLAB algorithm.

```
% A = 1;
% B = 5
function [y,z] = unsigned_subtraction(A,B)
    y = A - B;

    C = -20;
    z = C - B;
end
```

In the original code, both *A* and *B* are unsigned and the result of *A*-*B* can be negative. In the generated fixed-point code, *A* is promoted to signed. In the original code, *C* is signed, so does not require promotion in the generated code.

```
 %#codegen
% A = 1;
% B = 5
function [y,z] = unsigned_subtraction_fixpt(A,B)

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

y = fi(fi_signed(A) - B, 1, 3, 0, fm);
C = fi(-20, 1, 6, 0, fm);
z = fi(C - B, 1, 6, 0, fm);
end

function y = fi_signed(a)
coder.inline( 'always' );
if isfi( a ) && ~(issigned( a ))
    nt = numerictype( a );
    new_nt = numerictype( 1, nt.WordLength + 1, nt.FractionLength );
    y = fi( a, new_nt, fimath( a ) );
else
    y = a;
end
end
```

### Avoiding Loss of Range When Concatenating Arrays of Fixed-Point Numbers

If you concatenate matrices using `vertcat` and `horzcat`, the conversion process 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.

For example, consider the following MATLAB algorithm.

```
% A = 1, B = 100, C = 1000
function [y, z] = lb_node(A, B, C)
    %% single rows
    y = [A B C];
    %% multiple rows
    z = [A 5; A B; A C];
end
```

In the generated fixed-point code:

- For the expression  $y = [A \ B \ C]$ , the leftmost element,  $A$ , is cast to the type of  $C$  because  $C$  has the largest type in the row.
- For the expression  $[A \ 5; \ A \ B; \ A \ C]$ :
  - In the first row,  $A$  is cast to the type of  $C$  because  $C$  has the largest type of the whole expression.
  - In the second row,  $A$  is cast to the type of  $B$  because  $B$  has the larger type in the row.
  - In the third row,  $A$  is cast to the type of  $C$  because  $C$  has the larger type in the row.

```
##codegen
% A = 1, B = 100, C = 1000
function [y, z] = lb_node_fixpt(A, B, C)
    %% single rows
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', ...
                'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...
                'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi([fi(A, 0, 10, 0, fm) B C], 0, 10, 0, fm);

    %% multiple rows
    z = fi([fi(A, 0, 10, 0, fm) 5; fi(A, 0, 7, 0, fm) B; ...
            fi(A, 0, 10, 0, fm) C], 0, 10, 0, fm);
end
```

## Handling Non-Constant mpower Exponents

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



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

For example, consider the following MATLAB algorithm.

```
% a = 1
% b = 3
function y = exp_operator(a, b)
    % exponent is a constant so no need to specify precision
    y = a^3;
    % exponent is not a constant, use 'SpecifyPrecision' for 'ProductMode'
    y = b^a;
end
```

In the generated fixed-point code, for the expression  $y = a^3$ , the exponent is a constant, so there is no need to specify precision. For the expression  $y = b^a$ , the exponent is not constant, so the ProductMode is set to SpecifyPrecision.

```
##codegen
% a = 1
% b = 3
function y = exp_operator_fixpt(a, b)
    % exponent is a constant so no need to specify precision
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
               'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
               'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    y = fi(a^3, 0, 2, 0, fm);
    % exponent is not a constant, use 'SpecifyPrecision' for 'ProductMode'
    y(:) = fi(b, 'ProductMode', 'SpecifyPrecision',...
              'ProductWordLength', 2, 'ProductFractionLength', 0)^a;
end
```

## Fixed-Point Code for MATLAB Classes

In this section...
“Automated Conversion Support for MATLAB Classes” on page 14-116
“Unsupported Constructs” on page 14-116
“Coding Style Best Practices” on page 14-117

### Automated Conversion Support for MATLAB Classes

The automated conversion process:

- Proposes fixed-point data types based on simulation ranges for MATLAB classes. It does not propose data types based on derived ranges for MATLAB classes.

After simulation, the Fixed-Point Conversion tool:

- Function list contains class constructors, methods, and specializations.
- Code window displays the objects used in each function.
- Provides code coverage for methods.

For more information, see “Viewing Information for MATLAB Classes”.

- Supports class methods, properties, and specializations. For each specialization of a class, `class_name`, the conversion generates a separate `class_name_fixpt.m` file. For every instantiation of a class, the generated fixed-point code contains a call to the constructor of the appropriate specialization.
- Supports classes that have `get` and `set` methods such as `get.PropertyName`, `set.PropertyName`. These methods are called when properties are read or assigned. The `set` methods can be specialized. Sometimes, in the generated fixed-point code, assignment statements are transformed to function calls.

### Unsupported Constructs

The automated conversion process does not support:

- Class inheritance.
- Packages.
- Constructors that use `nargin` and `varargin`.

## Coding Style Best Practices

When you write MATLAB code that uses MATLAB classes:

- Initialize properties in the class constructor.
- Replace constant properties with static methods.

For example, consider the `counter` class.

```
classdef Counter < handle
    properties
        Value = 0;
    end

    properties(Constant)
        MAX_VALUE = 128
    end

    methods
        function out = next(this)
            out = this.Count;
            if this.Value == this.MAX_VALUE
                this.Value = 0;
            else
                this.Value = this.Value + 1;
            end
        end
    end
end
```

To use the automated fixed-point conversion process, rewrite the class to have a static class that initializes the constant property `MAX_VALUE` and a constructor that initializes the property `Value`.

```
classdef Counter < handle
    properties
        Value;
    end

    methods(Static)
        function t = MAX_VALUE()
            t = 128;
        end
    end
end
```

```
methods
function this = Counter()
    this.Value = 0;
end
function out = next(this)
    out = this.Value;
    if this.Value == this.MAX_VALUE
        this.Value = 0;
    else
        this.Value = this.Value + 1;
    end
end
end
end
```

## Automated Fixed-Point Conversion Best Practices

### In this section...

“Create a Test File” on page 14-119

“Prepare Your Algorithm for Code Acceleration or Code Generation” on page 14-120

“Check for Fixed-Point Support for Functions Used in Your Algorithm” on page 14-121

“Manage Data Types and Control Bit Growth” on page 14-121

“Convert to Fixed Point” on page 14-122

“Use the Histogram to Fine-Tune Data Type Settings” on page 14-122

“Optimize Your Algorithm” on page 14-124

“Avoid Explicit Double and Single Casts” on page 14-126

### Create a Test File

A best practice for structuring your code is to separate your core algorithm from other code that you use to test and verify the results. Create a test file to call your original MATLAB algorithm and fixed-point versions of the algorithm. For example, as shown in the following table, you might set up some input data to feed into your algorithm, and then, after you process that data, create some plots to verify the results. Since you need to convert only the algorithmic portion to fixed-point, it is more efficient to structure your code so that you have a test file, in which you create your inputs, call your algorithm, and plot the results, and one (or more) algorithmic files, in which you do the core processing.

Original code	Best Practice	Modified code
<pre>% TEST INPUT x = randn(100,1);  % ALGORITHM y = zeros(size(x)); y(1) = x(1); for n=2:length(x)     y(n)=y(n-1) + x(n); end  % VERIFY RESULTS yExpected=cumsum(x);</pre>	<p><b>Issue</b></p> <p>Generation of test input and verification of results are intermingled with the algorithm code.</p> <p><b>Fix</b></p> <p>Create a test file that is separate from your algorithm.</p>	<pre>Test file  % TEST INPUT x = randn(100,1);  % ALGORITHM y = cumulative_sum(x);  % VERIFY RESULTS yExpected = cumsum(x); plot(y-yExpected) title('Error')</pre>

Original code	Best Practice	Modified code
<pre>plot(y-yExpected) title('Error')</pre>	Put the algorithm in its own function.	<pre>Algorithm in its own function  function y = cumulative_sum(x)     y = zeros(size(x));     y(1) = x(1);     for n=2:length(x)         y(n) = y(n-1) + x(n);     end end</pre>

You can use the test file to:

- Verify that your floating-point algorithm behaves as you expect before you convert it to fixed point. The floating-point algorithm behavior is the baseline against which you compare the behavior of the fixed-point versions of your algorithm.
- Propose fixed-point data types.
- Compare the behavior of the fixed-point versions of your algorithm to the floating-point baseline.
- Help you determine initial values for static ranges.

By default, the Fixed-Point Conversion tool shows code coverage results. Your test files should exercise the algorithm over its full operating range so that the simulation ranges are accurate. For example, for a filter, realistic inputs are impulses, sums of sinusoids, and chirp signals. With these inputs, using linear theory, you can verify that the outputs are correct. Signals that produce maximum output are useful for verifying that your system does not overflow. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results help you verify that your test file is exercising the algorithm adequately. Review code flagged with a red code coverage bar because this code is not executed. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. For more information see “Code Coverage”.

## Prepare Your Algorithm for Code Acceleration or Code Generation

The automated conversion process instruments your code and provides data type proposals to help you convert your algorithm to fixed point.

MATLAB algorithms that you want to convert to fixed point automatically must comply with code generation requirements and rules. To view the subset of the MATLAB

language that is supported for code generation, see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List”.

To help you identify unsupported functions or constructs in your MATLAB code, add the `%#codegen` pragma to the top of your MATLAB file. The MATLAB code analyzer flags functions and constructs that are not available in the subset of the MATLAB language supported for code generation. This advice appears in real-time as you edit your code in the MATLAB editor. For more information, see “Check Code With the Code Analyzer”. The software provides a link to a report that identifies calls to functions and the use of data types that are not supported for code generation. For more information, see “Check Code Using the Code Generation Readiness Tool”.

## Check for Fixed-Point Support for Functions Used in Your Algorithm

The Fixed-Point Conversion tool flags unsupported function calls found in your algorithm on the **Function Replacements** tab. For example, if you use the `fft`, which is not supported for fixed point. The tool adds an entry to the table on this tab and indicates that you need to specify a replacement function to use for fixed-point operations.

Variables	Function Replacements	Simulation Output
	Function or Operator	Fixed-Point Replacement
	<code>fft</code>	Replacement required to use fixed-point
	Click to add	Click to add

You can specify additional replacement functions. For example, functions like `sin`, `cos`, and `sqrt` may support fixed point, but for better efficiency, you may want to consider an alternative implementation like a lookup table or CORDIC-based algorithm.

## Manage Data Types and Control Bit Growth

The automated fixed-point conversion process automatically manages data types and controls bit growth. It controls bit growth by using subscripted assignments, that is, assignments that use the colon (`:`) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. In addition to preventing bit growth,

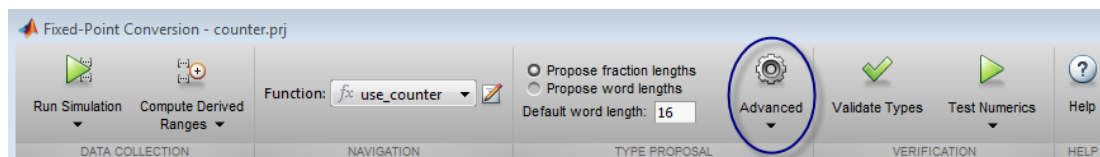
subscripted assignment reduces the number of casts in the generated fixed-point code and makes the code more readable.

## Convert to Fixed Point

### What Are Your Goals for Converting to Fixed Point?

Before you start the conversion, consider your goals for converting to fixed point. Are you implementing your algorithm in C or HDL? What are your target constraints? The answers to these questions determine many fixed-point properties such as the available word length, fraction length, and math modes, as well as available math libraries.

To set up these properties, use the **Advanced** settings.



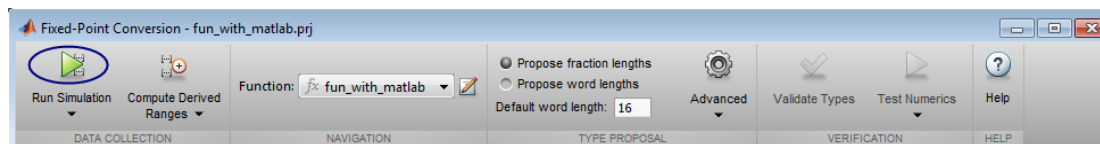
For more information, see “Type Proposal Settings”.

### Run With Fixed-Point Types and Compare Results

Create a test file to validate that the floating-point algorithm works as expected before converting it to fixed point. You can use the same test file to propose fixed-point data types, and to compare fixed-point results to the floating-point baseline after the conversion. For more information, see “Running a Simulation” on page 14-91 and “Histogram” on page 14-100 .

### Use the Histogram to Fine-Tune Data Type Settings

To fine-tune fixed-point type settings, use the histogram. To log data for histograms, in the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the Run Simulation button.





After simulation and static analysis:

- To view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The screenshot shows a software interface with two tabs: "Whole Number" and "Proposed Type". The "Proposed Type" tab is active, displaying a histogram for a variable named "No". The histogram shows a distribution of values, with a peak near zero. The "Proposed Type" field is set to "numerictype(1, 16, 14)". Below the histogram, the text "Sim values covered 99%" is displayed, along with a checked "Signed" checkbox and the "Supported range -2 : 1.9999".

Variable	Proposed Type
No	numerictype(1, 16, 14)

Sim values covered **99%**  Signed

Supported range **-2 : 1.9999**

You can view the effect of changing the proposed data types by dragging the edges of the bounding box in the histogram window to change the proposed data type and selecting or clearing the **Signed** option.

- If the values overflow and the range cannot fit the proposed type, the table shows proposed types in red.

When the tool applies data types, it generates an html report that provides overflow information and highlights overflows in red. Review the proposed data types.

## Optimize Your Algorithm

### Use `fimath` to Get Optimal Types for C or HDL

`fimath` properties define the rules for performing arithmetic operations on `fi` objects, including math, rounding, and overflow properties. You can use the `fimath ProductMode` and `SumMode` properties to retain optimal data types for C and HDL. HDL can have arbitrary word length types in the generated HDL code whereas C requires container types (`uint8`, `uint16`, `uint32`). Use the **Advanced** settings, see “Type Proposal Settings”.

#### C

The `KeepLSB` setting for `ProductMode` and `SumMode` models the behavior of integer operations in the C language, while `KeepMSB` models the behavior of many DSP devices. Different rounding methods require different amounts of overhead code. Setting the `RoundingMethod` property to `Floor`, which is equivalent to two's complement truncation, provides the most efficient rounding implementation. Similarly, the standard method for handling overflows is to wrap using modulo arithmetic. Other overflow handling methods create costly logic. Whenever possible, set `OverflowAction` to `Wrap`.

MATLAB Code	Best Practice	Generated C Code
Code being compiled  <pre>function y = adder(a,b)     y = a + b; end</pre>	<b>Issue</b>  With the default word length set to 16 and the default <code>fimath</code> settings, additional code is generated to implement saturation overflow, nearest rounding, and full-precision arithmetic.	<pre>int adder(short a, short b) {     int y;     int i0;     int i1;     int i2;     int i3;     i0 = a;     i1 = b;     if ((i0 &amp; 65536) != 0) {</pre>
<b>Note:</b> In the Fixed-Point Conversion tool, set		

MATLAB Code	Best Practice	Generated C Code												
Default word length to 16.		<pre> i2 = i0   -65536; } else { i2 = i0 &amp; 65535; }  if ((i1 &amp; 65536) != 0) { i3 = i1   -65536; } else { i3 = i1 &amp; 65535; }  i0 = i2 + i3; if ((i0 &amp; 65536) != 0) { y = i0   -65536; } else { y = i0 &amp; 65535; }  return y; } </pre>												
	<p><b>Fix</b></p> <p>To make the generated C code more efficient, choose fixed-point math settings that match your processor types.</p> <p>To customize fixed-point type proposals, use the Fixed-Point Conversion tool <b>Advanced</b> settings. Select <b>fimath</b> and then set:</p> <table border="1" data-bbox="457 1236 967 1520"> <tbody> <tr> <td>Rounding method</td> <td>Floor</td> </tr> <tr> <td>Overflow action</td> <td>Wrap</td> </tr> <tr> <td>Product mode</td> <td>KeepLSB</td> </tr> <tr> <td>Sum mode</td> <td>KeepLSB</td> </tr> <tr> <td>Product word length</td> <td>32</td> </tr> <tr> <td>Sum word length</td> <td>32</td> </tr> </tbody> </table>	Rounding method	Floor	Overflow action	Wrap	Product mode	KeepLSB	Sum mode	KeepLSB	Product word length	32	Sum word length	32	<pre> int adder(short a, short b) { return a + b; } </pre>
Rounding method	Floor													
Overflow action	Wrap													
Product mode	KeepLSB													
Sum mode	KeepLSB													
Product word length	32													
Sum word length	32													

## HDL

For HDL code generation, set:

- `ProductMode` and `SumMode` to `FullPrecision`
- `Overflow action` to `Wrap`
- `Rounding method` to `Floor`

## Replace Built-in Functions With More Efficient Fixed-Point Implementations

Some MATLAB built-in functions can be made more efficient for fixed-point implementation. For example, you can replace a built-in function with a Lookup table implementation, or a CORDIC implementation, which requires only iterative shift-add operations. For more information, see “Function Replacements” on page 14-102.

## Re-implement Division Operations Where Possible

Often, division is not fully supported by hardware and can result in slow processing. When your algorithm requires a division, consider replacing it with one of the following options:

- Use bit shifting when the denominator is a power of two. For example, `bitsra(x,3)` instead of `x/8`.
- Multiply by the inverse when the denominator is constant. For example, `x*0.2` instead of `x/5`.
- If the divisor is not constant, use a temporary variable for the division. Doing so results in a more efficient data type proposal and, if overflows occur, makes it easier to see which expression is overflowing.

## Eliminate Floating-Point Variables

For more efficient code, the automated fixed-point conversion process eliminates floating-point variables. The one exception to this is loop indices because they usually become integer types. It is good practice to inspect the fixed-point code after conversion to verify that there are no floating-point variables in the generated fixed-point code.

## Avoid Explicit Double and Single Casts

For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts.

Instead of using casts, supply a replacement function. For more information, see “Function Replacements” on page 14-102.

## Replacing Functions Using Lookup Table Approximations

The Fixed-Point Designer software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations in a MATLAB Coder project, see “Replace the exp Function with a Lookup Table” and “Replace a Custom Function with a Lookup Table”.

To use lookup table approximations in the programmatic workflow, see `coder.approximation`, “Replace the exp Function with a Lookup Table”, and “Replace a Custom Function with a Lookup Table”.

## MATLAB Language Features Supported for Automated Fixed-Point Conversion

Fixed-Point Designer supports the following MATLAB language features in automated fixed-point conversion:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “Generate Code for Variable-Size Data”). Range computation for variable-sized data is supported via simulation mode only. Variable-sized data is not supported for comparison plotting.
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation”)
- Complex numbers (see “Code Generation for Complex Data”)
- Numeric classes (see “Supported Variable Types”)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB”)
- Program control statements `if`, `switch`, `for`, `while`, and `break`
- Arithmetic, relational, and logical operators
- Local functions
- Persistent variables (see “Define and Initialize Persistent Variables”)
- Structures. Range computation for structures is supported via simulation mode only.
- Characters

The complete set of Unicode characters is not supported for code generation. Characters are restricted to 8 bits of precision in generated code. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with characters if you intend to convert your MATLAB algorithm to fixed point.

- MATLAB classes. Range computation for MATLAB classes is supported via simulation mode only.

Automated conversion supports:

- Class properties

- Constructors
- Methods
- Specializations

It does not support class inheritance or packages. For more information, see “Fixed-Point Code for MATLAB Classes”.

- Ability to call functions (see “Resolution of Function Calls for Code Generation” on page 13-2)
- Subset of MATLAB toolbox functions (see “Functions Supported for Code Acceleration or C Code Generation”).
- Subset of DSP System Toolbox System objects.

The DSP System Toolbox System objects supported for automated conversion are:

- `dsp.BiquadFilter`
- `dsp.FIRFilter`, Direct Form only
- `dsp.FIRRateConverter`
- `dsp.LowerTriangularSolver`
- `dsp.UpperTriangularSolver`
- `dsp.ArrayVectorAdder`



## Inspecting Data Using the Simulation Data Inspector

### In this section...

“What Is the Simulation Data Inspector?” on page 14-131

“Import Logged Data” on page 14-131

“Export Logged Data” on page 14-131

“Group Signals” on page 14-131

“Run Options” on page 14-132

“Create Report” on page 14-132

“Comparison Options” on page 14-132

“Enabling Plotting Using the Simulation Data Inspector” on page 14-132

“Save and Load Simulation Data Inspector Sessions” on page 14-132

### What Is the Simulation Data Inspector?

The Simulation Data Inspector allows you to view data logged during the fixed-point conversion process. You can use it to inspect and compare the inputs and outputs to the floating-point and fixed-point versions of your algorithm.

For fixed-point conversion, there is no programmatic interface for the Simulation Data Inspector.

### Import Logged Data

Before importing data into the Simulation Data Inspector, you must have previously logged data to the base workspace or to a MAT-file.

### Export Logged Data

The Simulation Data Inspector provides the capability to save data collected by the fixed-point conversion process to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

### Group Signals

You can customize the organization of your logged data in the Simulation Data Inspector **Runs** pane. By default, data is first organized by run. You can then organize your data by logged variable or no hierarchy.

## Run Options

You can configure the Simulation Data Inspector to:

- Append New Runs

In the Run Options dialog box, the default is set to add new runs to the bottom of the run list. To append new runs to the top of the list, select **Add new runs to top**.

- Specify a Run Naming Rule

To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Configuration**.

## Create Report

You can create a report of the runs or comparison plots. Specify the name and location of the report file. By default, the Simulation Data Inspector overwrites existing files. To preserve existing reports, select **If report exists, increment file name to prevent overwriting**.

## Comparison Options

To change how signals are matched when runs are compared, specify the **Align by** and **Then by** parameters and then click **OK**.

## Enabling Plotting Using the Simulation Data Inspector

To enable the Simulation Data Inspector in the Fixed-Point Conversion tool, see “Enable Plotting Using the Simulation Data Inspector”.

To enable the Simulation Data Inspector in the programmatic workflow, see “Enable Plotting Using the Simulation Data Inspector”.

## Save and Load Simulation Data Inspector Sessions

If you have data in the Simulation Data Inspector and you want to archive or share the data to view in the Simulation Data Inspector later, save the Simulation Data Inspector session. When you save a Simulation Data Inspector session, the MAT-file contains:

- All runs, data, and properties from the **Runs** and **Comparisons** panes.
- Check box selection state for data in the **Runs** pane.

### **Save a Session to a MAT-File**

- 1** On the **Visualize** tab, click **Save**.
- 2** Browse to where you want to save the MAT-file to, name the file, and click **Save**.

### **Load a Saved Simulation Data Inspector Simulation**

- 1** On the **Visualize** tab, click **Open**.
- 2** Browse, select the MAT-file saved from the Simulation Data Inspector, and click **Open**.
- 3** If data in the session is plotted on multiple subplots, on the **Format** tab, click **Subplots** and select the subplot layout.

## Custom Plot Functions

The Fixed-Point Conversion tool provides a default time series based plotting function. The conversion process uses this function at the test numerics step to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. For example, plots that show eye diagrams and bit error differences are more suitable in the communications domain and histogram difference plots are more suitable in image processing designs.

You can choose to use a custom plot function at the test numerics step. The Fixed-Point Conversion tool facilitates custom plotting by providing access to the raw logged input and output data before and after fixed-point conversion. You supply a custom plotting function to visualize the differences between the floating-point and fixed-point results. If you specify a custom plot function, the fixed-point conversion process calls the function for each input and output 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.

Use this information to:

- Customize plot headings and axes.
- Choose which variables to plot.
- Generate different error metrics for different output variables.
- A cell array to hold the logged floating-point values for the variable.

This cell array contains values observed during floating-point simulation of the algorithm during the test numerics phase. You might need to reformat this raw data.

- A cell array to hold the logged values for the variable after fixed-point conversion.

This cell array contains values observed during fixed-point simulation of the converted design.

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 programmatic workflow, set the `coder.FixptConfig` configuration object `PlotFunction` property to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results”.

## Data Type Issues in Generated Code

Within the fixed-point conversion HTML report you have the option to highlight MATLAB code that results in double, single, or expensive fixed-point operations. Consider enabling these checks when trying to achieve a strict single, or fixed-point design.

These checks are disabled by default.

### Enable the Highlight Option in a MATLAB Coder Project

- 1 Open the **Settings** menu.
- 2 Under **Plotting and Reporting**, set **Highlight potential data type issues** to **Yes**.

### Enable the Highlight Option at the Command Line

- 1 Create a fixed-point code configuration object:

```
cfg = coder.config('fixpt');
```
- 2 Set the `HighlightPotentialDataTypeIssues` property of the configuration object to `true`.

```
cfg.HighlightPotentialDataTypeIssues = true;
```

### Stowaway Doubles

When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone. This check highlights all expressions that result in a double operation.

### Stowaway Singles

This check highlights all expressions that result in a single operation.

### Expensive Fixed-Point Operations

The expensive fixed-point operations check identifies optimization opportunities by highlighting expressions in the MATLAB code which result in cumbersome

multiplication or division, or expensive rounding in generated code. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient”.

### **Cumbersome Operations**

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that have word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

### **Expensive Rounding**

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

### **Expensive Comparison Operations**

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.





# Automated Fixed-Point Conversion Using Programmatic Workflow

---

- “Convert MATLAB Code to Fixed-Point C Code” on page 15-2
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 15-5
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-11
- “Detect Overflows” on page 15-19
- “Replace the exp Function with a Lookup Table” on page 15-23
- “Replace a Custom Function with a Lookup Table” on page 15-25
- “Enable Plotting Using the Simulation Data Inspector” on page 15-28
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 15-29

## Convert MATLAB Code to Fixed-Point C Code

This example shows how to generate fixed-point C code from floating-point MATLAB code using the programmatic workflow.

### Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name. For example:

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'fun_with_matlab_test';
```

### Configure the Fixed-Point Configuration Object for Type Proposal

The fixed-point conversion software can propose types based on simulation ranges, derived ranges, or both.

- For type proposal using only simulation ranges, enable the collection and reporting of simulation range data. By default, derived range analysis is disabled.

```
fixptcfg.ComputeSimulationRanges = true;
```

- For type proposal using only derived ranges:

- 1 Specify the design range for input parameters. For example:

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
```

- 2 Enable derived range analysis. Disable collection and reporting of simulation range data.

```
fixptcfg.ComputeDerivedRanges = true;  
fixptcfg.ComputeSimulationRanges = false;
```

### Enable Numerics Testing

Select to run the test file to verify the generated fixed-point MATLAB code.

```
fixptcfg.TestNumerics = true;
```

### Enable Plotting

Log inputs and outputs for comparison plotting. Select to plot using a custom function or Simulation Data Inspector. For example, to plot using Simulation Data Inspector:

```
fixptcfg.LogIOForComparisonPlotting = true;  
fixptcfg.PlotWithSimulationDataInspector = true;
```

### Configure Additional Fixed-Point Configuration Object Properties

Configure additional fixed-point configuration object properties as necessary. For example, define the default fixed-point word length:

```
fixptcfg.DefaultWordLength = 16;
```

### Set Up the C Code Generation Configuration Object

Create a code configuration object for generation of a C static library, dynamic library, or executable. Enable the code generation report. For example:

```
cfg = coder.config('lib');  
cfg.GenerateReport = true;
```

### Generate Fixed-Point C Code

Use the `codegen` function to convert the floating-point MATLAB function to fixed-point C code. For example:

```
codegen -float2fixed fixptcfg -config cfg fun_with_matlab
```

### View the Type Proposal Report

Click the link to the type proposal report for the entry-point function.

### View the Comparison Plots

If you selected to log inputs and outputs for comparison plots, the conversion process generates comparison plots.

- If you selected to use Simulation Data Inspector for these plots, the Simulation Data Inspector opens. Use Simulation Data Inspector to view and compare the floating-point and fixed-point run information.
- If you selected to use a custom plotting function for these plots, the conversion process uses the custom function to generate the plots.

### View the Generated Fixed-Point MATLAB and Fixed-Point C Code

Click the [View Report](#) link that follows the type proposal report. To view the fixed-point MATLAB code, click the **MATLAB code** tab. To view the fixed-point C code, click the **C code** tab.

### See Also

`coder.FixptConfig`

### **Related Examples**

- “Propose Fixed-Point Data Types Based on Simulation Ranges”
- “Propose Fixed-Point Data Types Based on Derived Ranges”
- “Enable Plotting Using the Simulation Data Inspector”

### **More About**

- “Automated Fixed-Point Conversion”

## Propose Fixed-Point Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/).

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\fun_with_matlab`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:
 

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```
- 3 Copy the `fun_with_matlab.m` and `fun_with_matlab_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>fun_with_matlab.m</code>	Entry-point MATLAB function
Test file	<code>fun_with_matlab_test.m</code>	MATLAB script that tests <code>fun_with_matlab.m</code>

## The `fun_with_matlab` Function

```
function y = fun_with_matlab(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
end
```

## The fun\_with\_matlab\_test Script

The test script runs the fun\_with\_matlab function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```
% fun_with_matlab_test
%
% Define representative inputs
N = 256; % Number of points
t = linspace(0,1,N); % Time vector from 0 to 1 second
f1 = N/2; % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N); % Step
x_impulse = zeros(1,N); % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
    y(i,:) = fun_with_matlab(x(i,:));
end

% Plot the results
titles = {'Chirp', 'Step', 'Impulse'}
```

```

clf
for i = 1:size(x,1)
    subplot(size(x,1),1,i)
    plot(t,x(i,:),t,y(i,:))
    title(titles{i})
    legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')

```

### Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name.

```

fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'fun_with_matlab_test';

```

### Set Up the C Code Generation Configuration Object

Create a code configuration object to generate a C static library. Enable the code generation report.

```

cfg = coder.config('lib');
cfg.GenerateReport = true;

```

### Collect Simulation Ranges and Generate Fixed-Point Code

Use the `codegen` function to convert the floating-point MATLAB function, `fun_with_matlab`, to fixed-point C code. Set the default word length for the fixed-point data types to 16.

```

fixptcfg.ComputeSimulationRanges = true;
fixptcfg.DefaultWordLength = 16;

% Derive ranges and generate fixed-point code
codegen -float2fixed fixptcfg -config cfg fun_with_matlab

```

`codegen` analyzes the floating-point code. Because you did not specify the input types for the `fun_with_matlab` function, the conversion process infers types by simulating the test file. The conversion process then derives ranges for variables in the algorithm. It uses these derived ranges to propose fixed-point types for these variables. When the conversion is complete, it generates a type proposal report.

### View Range Information

Click the link to the type proposal report for the `fun_with_matlab` function, `fun_with_matlab_report.html`.

The report opens in a web browser.

### Fixed-Point Report `fun_with_matlab`

```
function y = fun_with_matlab(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, 0.333333333333333 ];
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
end
```

Variable Name	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	ProposedType (Best For WL = 16)
a	double 1 x 3	-0.942809041582063	0.195262145875635	1		No	numericType(1, 16, 14)
b	double 1 x 3	0.0976310729378175	0.195262145875635			No	numericType(0, 16, 18)
i	double	1	256			Yes	numericType(0, 9, 0)
x	double 1 x 256	-0.9999756307053946	1			No	numericType(1, 16, 14)
y	double 1 x 256	-0.9696817930434206	1.0553496057969345			No	numericType(1, 16, 14)
z	double 2 x 1	-0.8907046852192462	0.957718532859117			No	numericType(1, 16, 15)

### View Generated Fixed-Point MATLAB Code

`codegen` generates a fixed-point version of the `fun_with_matlab.m` function, `fun_with_matlab_fixpt.m`, and a wrapper function that calls `fun_with_matlab_fixpt`. These files are generated in the `codegen \fun_with_matlab\fixpt` folder in your local working folder.

```
function y = fun_with_matlab_fixpt(x)
fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode',...
'FullPrecision', 'SumMode', 'FullPrecision');
%#codegen

persistent z
if isempty( z )
    z = fi(zeros( 2, 1 ), 1, 16, 15, fm);
end
```



```

% [b,a] = butter(2, 0.25)
b = fi([ 0.0976310729378175, 0.195262145875635, 0.0976310729378175 ], 0, 16, 18, fm);
a = fi([ 1, -0.942809041582063, 0.3333333333333333 ], 1, 16, 14, fm);
y = fi(zeros( size( x ) ), 1, 16, 14, fm);
for i = 1:length( x )
    y( i ) = b( 1 )*x( i ) + z( 1 );
    z( 1 ) = fi_signed(b( 2 )*x( i ) + z( 2 )) - a( 2 )*y( i );
    z( 2 ) = fi_signed(b( 3 )*x( i )) - a( 3 )*y( i );
end
end

function y = fi_signed(a)
coder.inline( 'always' );
if isfi( a ) && ~(issigned( a ))
    nt = numericitytype( a );
    new_nt = numericitytype( 1, nt.WordLength + 1, nt.FractionLength );
    y = fi( a, new_nt, fimath( a ) );
else
    y = a;
end
end

```

### View Generated Fixed-Point C Code

To view the code generation report for the C code generation, click the [View Report](#) link that follows the type proposal report.

```

===== Step3: Generate Fixed Point Code =====

### Generating Fixed Point MATLAB Code fun\_with\_matlab\_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper fun\_with\_matlab\_wrapper\_fixpt
### Generating Mex file for ' fun\_with\_matlab\_wrapper\_fixpt '
Code generation successful: View report
### Generating Type Proposal Report for 'fun_with_matlab' fun\_with\_matlab\_report.html
Code generation successful: View report
>>

```

The code generation report opens and displays the generated code for `fun_with_matlab_fixpt.c`.

### See Also

`coder.FixptConfig` | `codegen`

### Related Examples

- “Convert MATLAB Code to Fixed-Point C Code”
- “Propose Fixed-Point Data Types Based on Derived Ranges”

## Propose Fixed-Point Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges that you specify. The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time so you can save time by deriving ranges instead.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\dti`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

- 3 Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>dti.m</code>	Entry-point MATLAB function
Test file	<code>dti_test.m</code>	MATLAB script that tests <code>dti.m</code>

## The dti Function

The `dti` function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;
```

```
function b = subFunction(a)
b = a*a;
```

## The dti\_test Function

The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10)

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;

    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);
end

figure('Name', [mfilename, '_plot'])
subplot(2,1,1)
plot(1:len,x_in)
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

subplot(2,1,2)
plot(1:len,y_out)
```

```
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (DTI)')

disp('Test complete.')
```

### Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
```

### Specify Design Ranges

Specify design range information for the `dti` function input parameter `u_in`.

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
```

### Enable Plotting Using the Simulation Data Inspector

Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;
fixptcfg.LogIOForComparisonPlotting = true;
fixptcfg.PlotWithSimulationDataInspector = true;
```

### Set Up the C Code Generation Configuration Object

Create a code configuration object to generate a C static library. Enable the code generation report.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
```

### Derive Ranges and Generate Fixed-Point Code

Use the `codegen` function to convert the floating-point MATLAB function, `dti`, to fixed-point C code. Set the default word length for the fixed-point data types to 16.

```
fixptcfg.ComputeDerivedRanges = true;
fixptcfg.ComputeSimulationRanges = false;
fixptcfg.DefaultWordLength = 16;
```

```
% Derive ranges and generate fixed-point code
codegen -float2fixed fixptcfg -config cfg dti
```

codegen analyzes the floating-point code. Because you did not specify the input types for the dti function, the conversion process infers types by simulating the test file. The conversion process then derives ranges for variables in the algorithm. It uses these derived ranges to propose fixed-point types for these variables. When the conversion is complete, it generates a type proposal report.

### View Derived Range Information

Click the link to the type proposal report for the dti function, [dti\\_report.html](#).

The report opens in a web browser.

### Fixed Point Report dti

```
function [y,clip_status] = dti(u_in) %#codegen
    % Discrete Time Integrator in MATLAB
    %
    % Forward Euler method, also known as Forward Rectangular, or left-hand
    % approximation. The resulting expression for the output of the block at
    % step 'n' is y(n) = y(n-1) + K * u(n-1)
    %
    init_val = 1;
    gain_val = 1;
    limit_upper = 500;
    limit_lower = -500;
    % variable to hold state between consecutive calls to this block
    persistent u_state
    if isempty( u_state )
        u_state = init_val + 1;
    end
    % Compute Output
    if (u_state>limit_upper)
        y = limit_upper;
        clip_status = -2;
    elseif (u_state<=limit_upper)
        y = limit_upper;
        clip_status = -1;
    elseif (u_state
```

Variable Name	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	ProposedType (Best For WL = 16)
clip_status	double			-2	2	No	numerictype(1, 16, 13)
gain_val	double			1	1	Yes	numerictype(0, 1, 0)
init_val	double			1	1	Yes	numerictype(0, 1, 0)
limit_lower	double			-500	-500	Yes	numerictype(1, 10, 0)
limit_upper	double			500	500	Yes	numerictype(0, 9, 0)
tprod	double			-1	1	No	numerictype(1, 16, 14)
u_in	double			-1	1	No	numerictype(1, 16, 14)
u_state	double			-501	501	No	numerictype(1, 16, 6)
y	double			-500	500	No	numerictype(1, 16, 6)

## View Generated Fixed-Point MATLAB Code

codegen generates a fixed-point version of the dti function, dti\_fxpt.m, and a wrapper function that calls dti\_fxpt. These files are generated in the codegen\dti\fixpt folder in your local working folder.

```
function [y,clip_status] = dti_fxpt(u_in)
fm = fimath( 'RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode',...
            'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision',
            'MaxSumWordLength', 128);
%#codegen

% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation. The resulting expression for the output of the block at
% step 'n' is  $y(n) = y(n-1) + K * u(n-1)$ 
%
init_val = fi(1, 0, 1, 0, fm);
gain_val = fi(1, 0, 1, 0, fm);
limit_upper = fi(500, 0, 9, 0, fm);
limit_lower = fi(-500, 1, 10, 0, fm);
% variable to hold state between consecutive calls to this block
persistent u_state
if isempty( u_state )
    u_state = fi(init_val + fi(1, 0, 1, 0, fm), 1, 16, 6, fm);
end
% Compute Output
if (u_state>limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-2, 1, 16, 13, fm);
elseif (u_state>=limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-1, 1, 16, 13, fm);
elseif (u_state
```

## Compare Floating-Point and Fixed-Point Runs

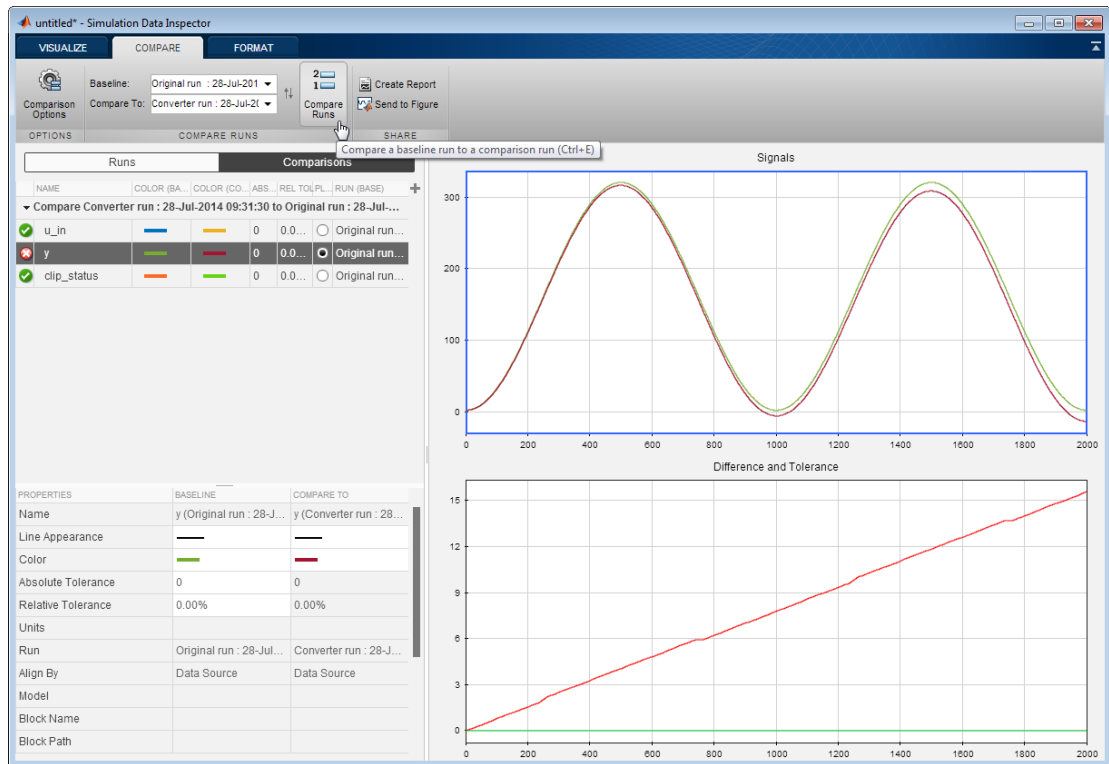
Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.

You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-



point values for the output  $y$ , on the **Compare** tab, select  $y$ , and then click **Compare Runs**.

The Simulation Data Inspector displays a plot of the baseline floating-point run against the fixed-point run and the difference between them.



## View Generated Fixed-Point C Code

To view the code generation report for the C code generation, click the **View Report** link that follows the type proposal report.

```
===== Step4: Verify Fixed Point Code =====

### Analyzing the design 'dti'
### Analyzing the test bench(es) 'dti_test'
### Begin Floating Point Simulation
Test complete.
### Floating Point Simulation Completed in 11.8506 sec(s)
### Begin Fixed Point Simulation : dti_test
Test complete.
----- Input variable : u_in -----
Plotting : u_in

----- Output variable : y -----
Plotting : y

----- Output variable : clip_status -----
Plotting : clip_status

### Fixed Point Simulation Completed in 33.5425 sec(s)
### Generating Type Proposal Report for 'dti_fixpt' dti\_fixpt\_report.html
### Elapsed Time: 46.2006 sec(s)
Code generation successful: View report
>>
```

The code generation report opens and displays the generated code for `dti_fixpt.c`.

## See Also

`coder.FixptConfig` | `codegen`

## Related Examples

- “Convert MATLAB Code to Fixed-Point C Code”
- “Propose Fixed-Point Data Types Based on Simulation Ranges”

## Detect Overflows

This example shows how to detect overflows at the command line. 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.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer

In a local, writable folder, create a function, `overflow`.

```
function y = overflow(b,x,reset)
    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
end
```

Create a test file, `overflow_test.m` to exercise the overflow algorithm.

```
function overflow_test
    % The filter coefficients were computed using the FIR1 function from
    % Signal Processing Toolbox.
    % b = fir1(11,0.25);
    b = [-0.004465461051254
         -0.004324228005260
          +0.012676739550326
          +0.074351188907780
          +0.172173206073645
          +0.249588554524763
          +0.249588554524763
          +0.172173206073645
          +0.074351188907780
          +0.012676739550326
          -0.004324228005260
          -0.004465461051254]';

    % Input signal
    nx = 256;
    t = linspace(0,10*pi,nx)';

    % Impulse
    x_impulse = zeros(nx,1); x_impulse(1) = 1;

    % Max Gain
    % The maximum gain of a filter will occur when the inputs line up with the
    % signs of the filter's impulse response.
    x_max_gain = sign(b)';
    x_max_gain = repmat(x_max_gain,ceil(nx/length(b)),1);
    x_max_gain = x_max_gain(1:nx);

    % Sums of sines
    f0=0.1; f1=2;
    x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);

    % Chirp
    f_chirp = 1/16; % Target frequency
    x_chirp = sin(pi*f_chirp*t.^2); % Linear chirp

    x = [x_impulse, x_max_gain, x_sines, x_chirp];
    titles = {'Impulse', 'Max gain', 'Sum of sines', 'Chirp'};
```

```

y = zeros(size(x));

for i=1:size(x,2)
    reset = true;
    y(:,i) = overflow(b,x(:,i),reset);
end

test_plot(1,titles,t,x,y)

end
function test_plot(fig,titles,t,x,y1)
    figure(fig)
    clf
    sub_plot = 1;
    font_size = 10;
    for i=1:size(x,2)
        subplot(4,1,sub_plot)
        sub_plot = sub_plot+1;
        plot(t,x(:,i),'c',t,y1(:,i),'k')
        axis('tight')
        xlabel('t','FontSize',font_size);
        title(titles{i},'FontSize',font_size);
        ax = gca;
        ax.FontSize = 10;
    end
    figure(gcf)
end

```

Create a coder.FixptConfig object, fixptcfg, with default settings.

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

Set the test bench name. In this example, the test bench function name is overflow\_test.

```
fixptcfg.TestBenchName = 'overflow_test';
```

Set the default word length to 16.

```
fixptcfg.DefaultWordLength = 16;
```

Enable overflow detection.

```
fixptcfg.TestNumerics = true;
fixptcfg.DetectFixptOverflows = true;
```

Set the `fimath` `Product` mode and `Sum` mode to `KeepLSB`. These settings models the behavior of integer operations in the C language.

```
fixptcfg.fimath = ...  
[ 'fimath(''RoundingMethod'', 'Floor'', 'OverflowAction'', ' ...  
  ''Wrap'', 'ProductMode'', 'KeepLSB'', 'SumMode'', 'KeepLSB'')'];
```

Create a code generation configuration object to generate a standalone C static library.

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

Convert the floating-point MATLAB function, `overflow`, to fixed-point C code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -float2fixed fixptcfg -config cfg overflow
```

The numerics testing phase reports an overflow.

```
Overflow error in expression 'acc + b( j )*( k )'. Percentage of Current Range = 104%.
```

Determine if the addition or the multiplication in this expression overflowed. Set the `fimath` `ProductMode` to `FullPrecision` so that the multiplication will not overflow, and then run the `codegen` command again.

```
fixptcfg.fimath = [ 'fimath(''RoundingMethod'', 'Floor'', 'OverflowAction'', ' ...  
  ''Wrap'', 'ProductMode'', 'FullPrecision'', 'SumMode'', 'KeepLSB'')'];  
codegen -float2fixed fixptcfg -config cfg overflow
```

The numerics testing phase still reports an overflow, indicating that it is the addition in the expression that is overflowing.

## Replace the exp Function with a Lookup Table

This example shows how to replace the `exp` function with a lookup table approximation in the generated fixed-point code using the programmatic workflow.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/).

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

### Create Algorithm and Test Files

- 1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

- 2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

### Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

## Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'my_fcn_test';  
fixptcfg.TestNumerics = true;  
fixptcfg.DefaultWordLength = 16;  
fixptcfg.addApproximation(q);
```

## Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

## View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `exp1`, for the `exp` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `my_fcn_fixpt`, calls this approximation instead of calling `exp`.

```
function y = my_fcn_fixpt(x)  
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', ...  
        'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...  
        'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
  
    y = fi(exp1(x), 0, 16,1, fm);  
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.



## Replace a Custom Function with a Lookup Table

This example shows how to replace a custom function with a lookup table approximation function using the programmatic workflow.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create a function replacement configuration object to approximate `custom_fcn`. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...  
                      'CandidateFunction',@custom_fcn, 'NumberOfPoints',50);
```

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'custom_test';  
fixptcfg.TestNumerics = true;  
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg call_custom_fcn
```

`codegen` generates fixed-point MATLAB code in `call_custom_fcn_fixpt.m`.

To view the generated fixed-point code, click the link to `call_custom_fcn_fixpt`.

The generated code contains a lookup table approximation, `custom_fcn1`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. The lookup table uses 50 points as specified. By default, it uses linear interpolation and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)  
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...  
              'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...  
              'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);  
  
    y = fi(custom_fcn1(x), 0, 14, 14, fm);  
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not

match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

## Enable Plotting Using the Simulation Data Inspector

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point logged input and output data. At the MATLAB command line:

- 1 Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'dti_test';
```

- 2 Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;  
fixptcfg.LogIOForComparisonPlotting = true;  
fixptcfg.PlotWithSimulationDataInspector = true;
```

- 3 Generate fixed-point MATLAB code using `codegen`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges”.

# Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the `codegen` function to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the `LogIOForComparisonPlotting` option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

## Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB)  
For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/)

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

## Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\custom_plot`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:  

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```
- 3 Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	myFilter.m	Entry-point MATLAB function
Test file	myFilterTest.m	MATLAB script that tests myFilter.m
Plotting function	plotDiff.m	Custom plot function
MAT-file	filterData.mat	Data to filter.

## The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
    b = complex(zeros(1,16));
    h = complex(zeros(1,16));
    h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

## The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end
```

```
end
```

## The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%         i) name
%         ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after Fixed-
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexp(varName, '_', '\\_');
    escapedFcnName = regexp(fcnName, '_', '\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

            % plot fixed point values
```

```
        y_vec = flatFixedVals;
        subplot(1, 2, 2);
        plotScatter(x_vec, y_vec, 100, fixedTitle);
    otherwise
        % Plot only output 'y' for this example, skip the rest
    end
end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');

    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

### Set Up Configuration Object

- 1 Create a `coder.FixptConfig` object.

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

- 2 Specify the test file name and custom plot function name. Enable logging and numerics testing.

```
fxptcfg.TestBenchName = 'myFilterTest';
fxptcfg.PlotFunction = 'plotDiff';
fxptcfg.TestNumerics = true;
fxptcfg.LogIOForComparisonPlotting = true;
fxptcfg.DefaultWordLength = 16;
```

### Convert to Fixed Point

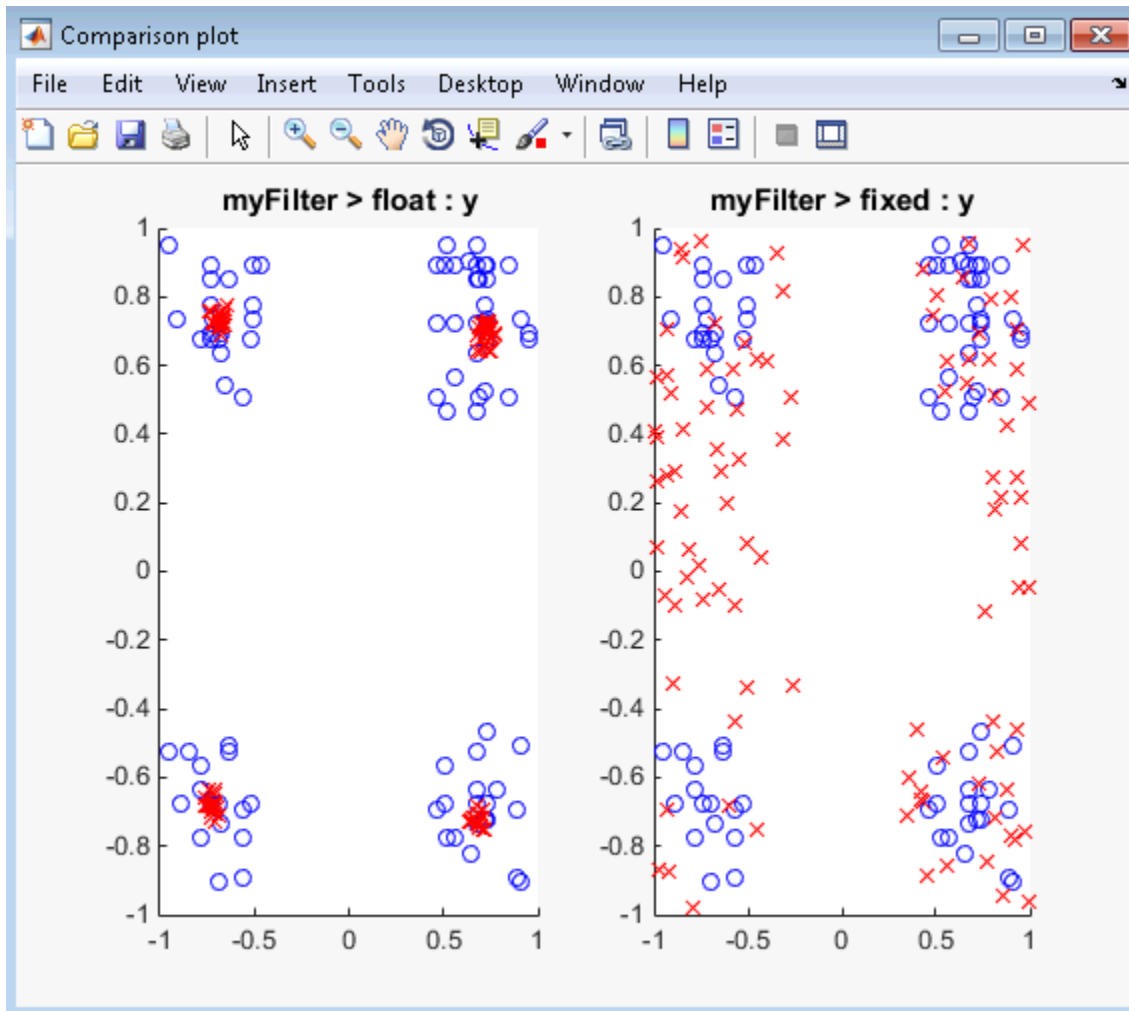
Convert the floating-point MATLAB function, `myFilter`, to floating-point MATLAB code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```



The conversion process generates fixed-point code using a default word length of 16 and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.

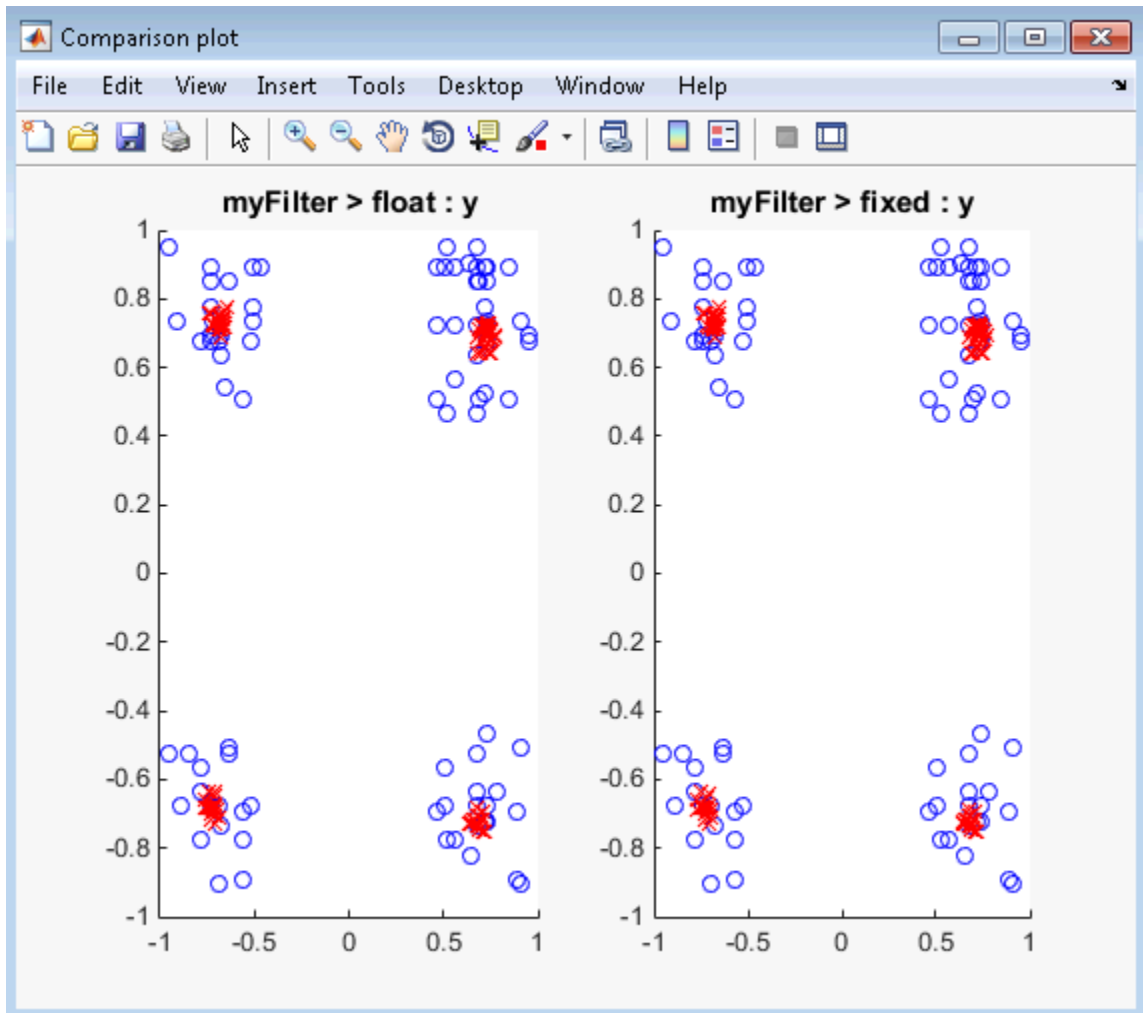


The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to 24 and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;  
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.



# Setting Up a MATLAB Coder Project

---

- “MATLAB Coder Project Set Up Workflow” on page 16-2
- “Creating a New Project” on page 16-3
- “Opening an Existing Project” on page 16-5
- “Adding Files to the Project” on page 16-6
- “Specifying Properties of Primary Function Inputs in a Project” on page 16-7
- “Autodefine Input Types” on page 16-8
- “Define Input Parameters by Example in a Project” on page 16-12
- “Define or Edit Input Parameter Type in a Project” on page 16-19
- “Define Constant Input Parameters in a Project” on page 16-29
- “Define Inputs Programmatically in the MATLAB File” on page 16-30
- “Adding Global Variables in a Project” on page 16-31
- “Specifying Global Variable Type and Initial Value in a Project” on page 16-32
- “Specify Output File Name” on page 16-40
- “Specify Output File Locations” on page 16-41
- “Selecting Output Type” on page 16-42

## **MATLAB Coder Project Set Up Workflow**

- 1** Create a new project or open an existing one.
- 2** Add the files from which you want to generate code.
- 3** Specify class, size, and complexity of all input parameters.
- 4** Optionally, add global variables.
- 5** Optionally, specify the output file name and output file locations.
- 6** Optionally, select the output type: MEX function (default), Instrumented MEX function, C/C++ static library, C/C++ dynamic library or C/C++ executable.

## Creating a New Project

### From the MATLAB APPS Tab

- 1 Select the **MATLAB Apps** tab.
- 2 In the **Code Generation** group, click **MATLAB Coder**.
- 3 In the MATLAB Coder Project dialog box, on the **New** tab, enter the name of your project in the **Name** field.
- 4 Enter the location of the project in the **Location** field.

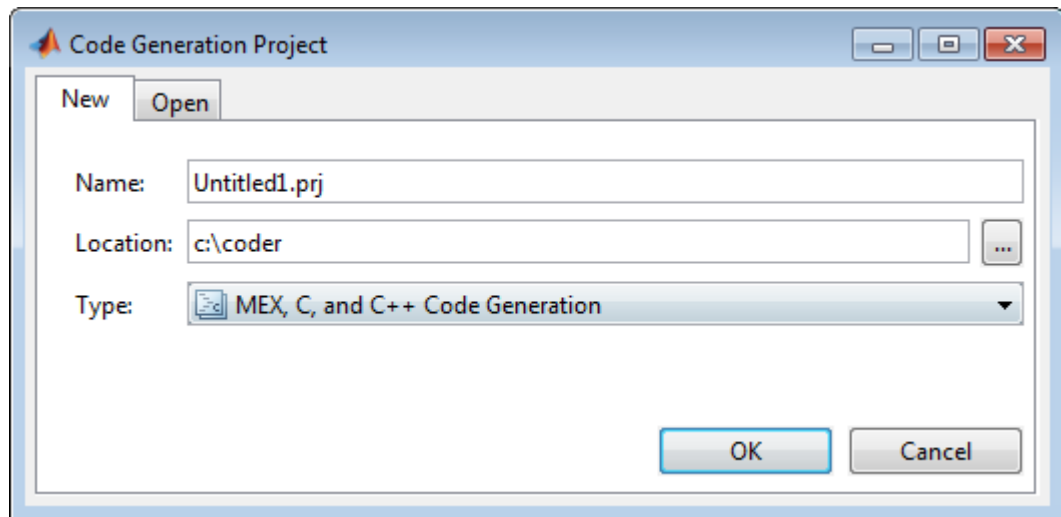
Alternatively, use the ... (browse) button to navigate to the location.

- 5 Click **OK**.

### At the Command Line

- 1 At the MATLAB command line, enter:

```
coder
```



- 2 In the **Name** field, enter the *project\_name*.
- 3 In the **Location** field, enter the location of the project.


Alternatively, use the ... (browse) button to navigate to the location.

---

**Note:** The path should not contain spaces, as this can lead to code generation failures in certain operating system configurations. If the path contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not be able to find files on this path.

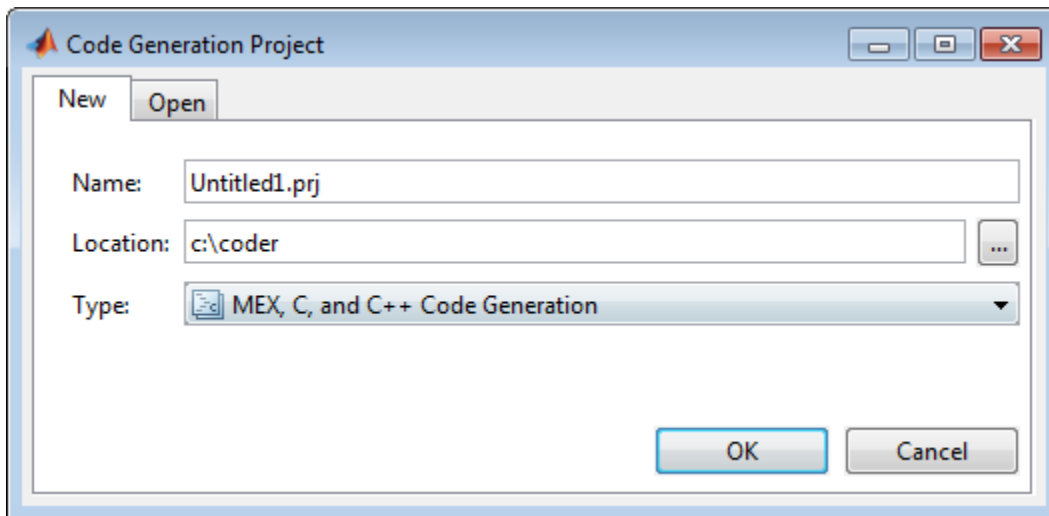
---

4 Click **OK**.

MATLAB Coder creates a project named *project\_name*.prj in the specified location and marks it with the project icon: .

## From a MATLAB Coder Project

If you already have a MATLAB Coder project open, in the upper-right corner of the project, click the **Actions** icon (⚙) and select **New Project**.



## Opening an Existing Project

### In this section...

“From the MATLAB APPS Tab” on page 16-5

“At the Command Line” on page 16-5

“From a MATLAB Coder Project” on page 16-5


### From the MATLAB APPS Tab

- 1 Select the MATLAB **Apps** tab.
- 2 In the **Code Generation** group, click **MATLAB Coder**.
- 3 In the MATLAB Coder Project dialog box, click the **Open** tab.
- 4 From the drop-down list, select a previously opened project or use the **Browse** button to find a project.
- 5 Click **OK**.

### At the Command Line

- 1 At the MATLAB command line, enter `coder`.
- 2 In the **Code Generation Project** dialog box, click the **Open** tab.
- 3 From the drop-down list, select a previously opened project or click the **Browse** button to find a project.
- 4 Click **OK**.

### From a MATLAB Coder Project

If you already have a MATLAB Coder project open, in the upper-right corner of the project, click the **Actions** icon () and select **Open Project**.

## Adding Files to the Project

First, you must add the MATLAB files from which you want to generate code to the project.

- Add only the main (entry-point) files that you call from MATLAB.
- Do not add files that are called by these files.
- Do not add files that have spaces in their names. The path should not contain spaces, as this can lead to code generation failures in certain operating system configurations.
- If the path contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not be able to find files on this path.

To add a file, do one of the following:

- In the project, in the **Entry-Point Files** pane on the **Overview** tab, click the **Add files** link and browse to the file.
- Drag a file from the current folder and drop it in the **Entry-Point Files** pane on the **Overview** tab.

If you add more than one entry-point file, MATLAB Coder lists them alphabetically.

If the functions that you added have inputs, you must define these inputs. See “Specifying Properties of Primary Function Inputs in a Project” on page 16-7.



# Specifying Properties of Primary Function Inputs in a Project

## Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB Coder must determine the properties of all variables in the MATLAB files at code generation time. To infer variable properties in MATLAB files, MATLAB Coder must be able to identify the properties of the inputs to the entry-point function. Therefore, if your entry-point function has inputs, you must specify the properties of these inputs. If your primary function has no input parameters, MATLAB Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the entry-point function.

You must specify the same number and order of inputs as the MATLAB function unless you use the tilde (~) character to specify unused function inputs. If you use the tilde character, the inputs default to real, scalar doubles.

### See Also

- “Properties to Specify”

## How to Specify an Input Definition in a Project

Specify an input definition in your MATLAB Coder project using one of the following methods:

- Autodefine Input Types
- Define Type
- Define by Example
- Define Constant
- Define Programmatically in the MATLAB File

Alternatively, specify input definitions at the command line and then use the `codegen` function to generate code. For more information, see “Primary Function Input Specification” on page 19-39.

## Autodefine Input Types

In this section...
“How MATLAB Coder Autodefines Input Types” on page 16-8
“Prerequisites for Autodefining Input Types” on page 16-8
“How to Autodefine Input Types” on page 16-8

### How MATLAB Coder Autodefines Input Types

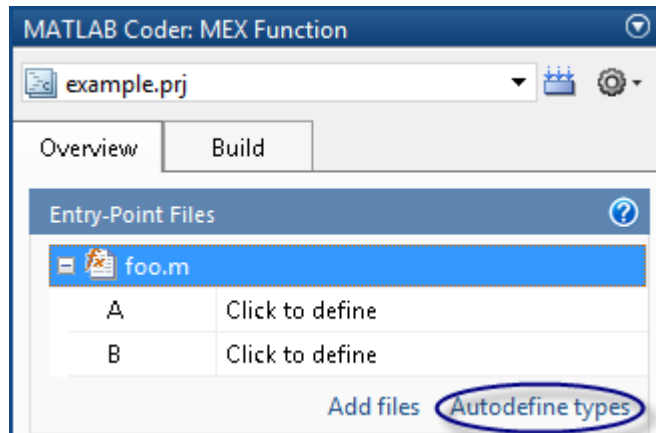
If you specify a test file that calls the project entry-point functions, the MATLAB Coder software can infer the input parameter types by running the test file. If a test file calls an entry-point function multiple times with different sized inputs, the MATLAB Coder software takes the union of the inputs and infers that the inputs are variable size, with an upper bound equal to the size of the largest input.

### Prerequisites for Autodefining Input Types

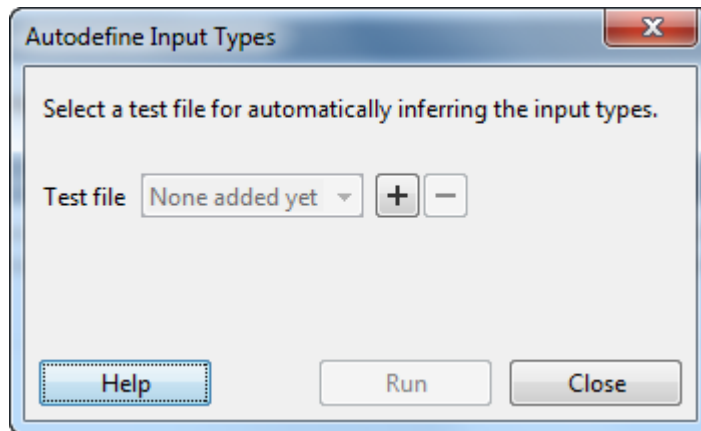
Before using MATLAB Coder to autodefine function input parameter types, you must add at least one entry-point file to your project. You must also specify a test file that calls your entry-point functions with the expected input types. The test file can be either a MATLAB function or a script. It should call the entry-point function at least once.

### How to Autodefine Input Types

- 1 On the MATLAB Coder project **Overview** tab, click the **Autodefine types** link.



- 2 In the **Autodefine Input Types** dialog box, click the **+** button to add a test file to the project.

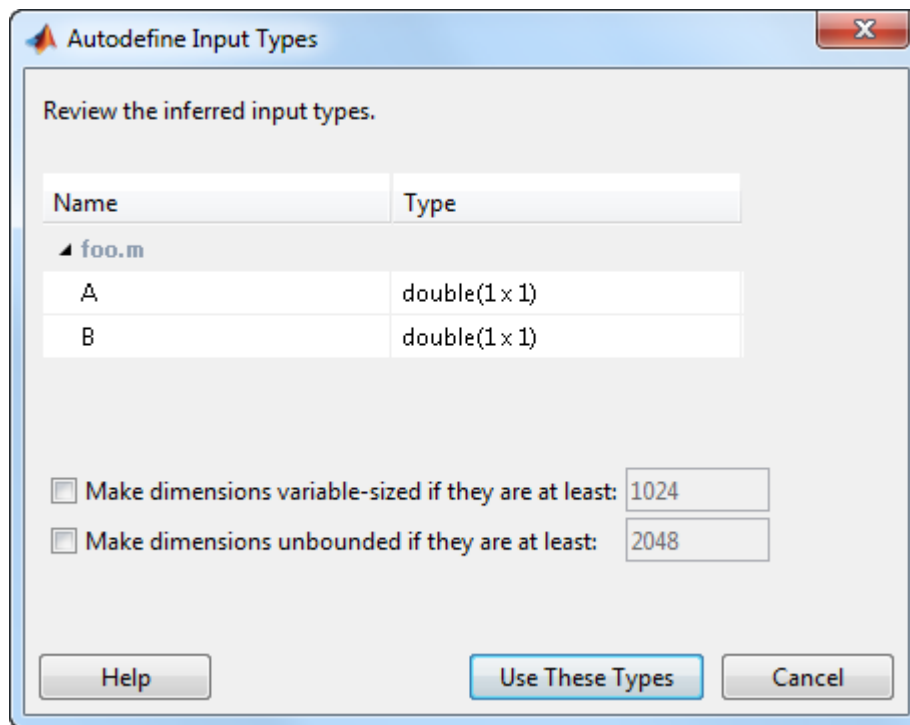


- 3 Browse to the folder that contains the test file and select the file.

Alternatively, if you have already added test files to the project, select one from the list.

- 4 Click the **Run** button.

The software runs the test file and, if the file calls entry-point functions, infers input types for these functions.



The dialog box displays a summary of the inferred types and provides the following options:

- **Make dimensions variable-sized if they are at least**

If you want inputs above a specified size to be variable size with an upper bound, select this option and specify the threshold. If the size,  $S$ , of any dimension of an input is equal to or greater than this threshold, the software makes this dimension variable size with an upper bound of  $S$ .

- **Make dimensions unbounded if they are at least**

If you want inputs above a specified size to be variable size with no upper bounds (unbounded), select this option and specify the threshold. If the size of any dimension of an input is equal to or greater than this threshold, the software makes this dimension unbounded.

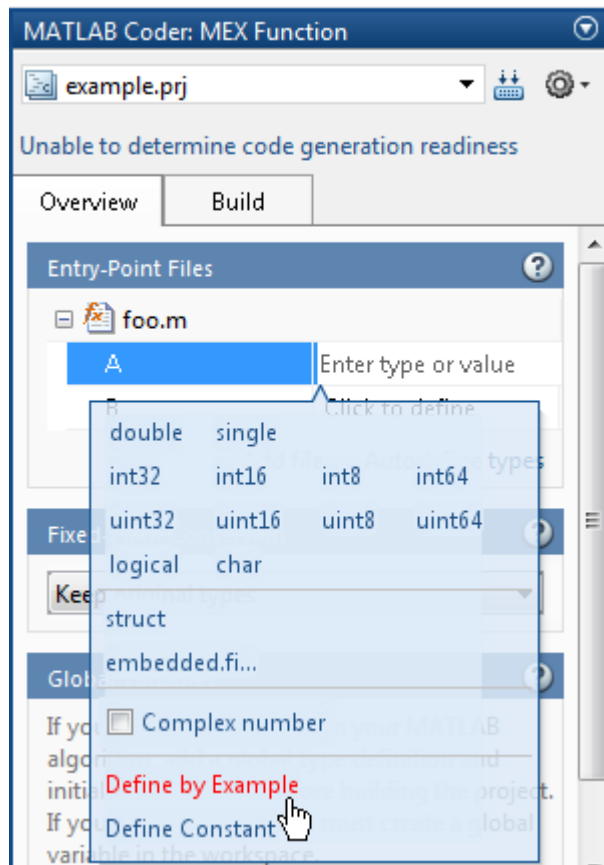
- 5 Review the inferred types. If the types are acceptable, click **Use These Types**. Otherwise, modify your test file, use a different test file to autodefine the types or define them using an alternate method. For more information, see “How to Specify an Input Definition in a Project” on page 16-7.

## Define Input Parameters by Example in a Project

In this section...
“How to Define an Input Parameter by Example” on page 16-12
“Specifying Input Parameters by Example” on page 16-13
“Specifying an Enumerated Type Input Parameter by Example” on page 16-15
“Specifying a Fixed-Point Input Parameter by Example” on page 16-16

### How to Define an Input Parameter by Example

- 1 On the MATLAB Coder project **Overview** tab, click the input parameter that you want to define.

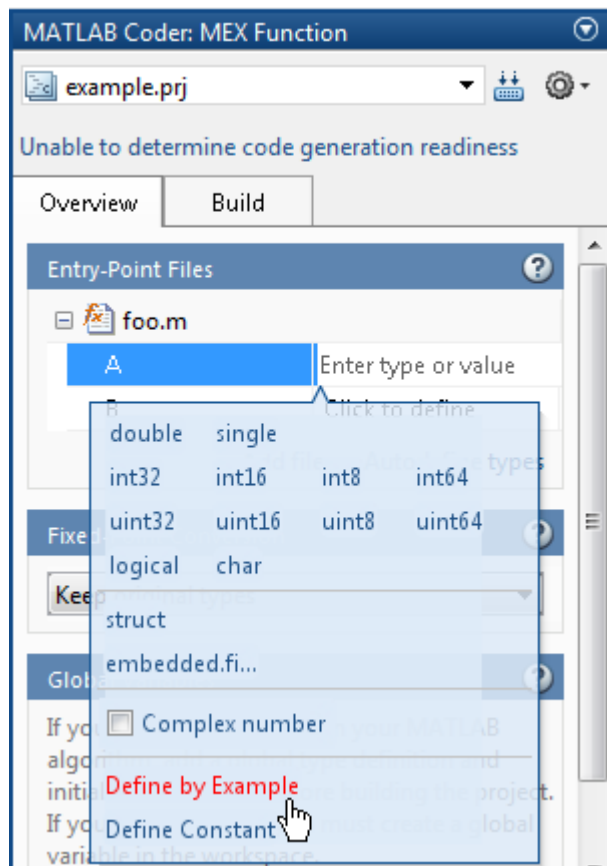


- 2 From the list of input options, select **Define by Example**.
- 3 In the field to the right of the parameter, enter a MATLAB expression. MATLAB Coder software uses the class, size, and complexity of the value of the specified variable or MATLAB expression when compiling the code.

## Specifying Input Parameters by Example

This example shows how to specify a 1-by-4 vector of unsigned 16-bit integers.

- 1 On the MATLAB Coder project **Overview** tab, click the input parameter that you want to define.



2 From the list of input options, select **Define by Example**.

3 In the field to the right of the parameter, enter:

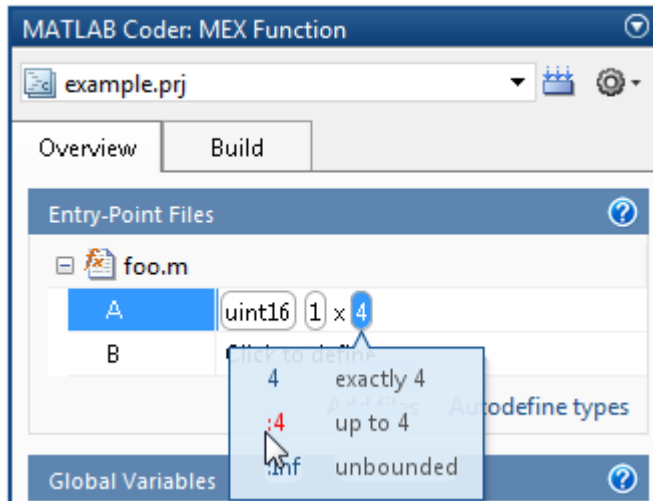
```
zeros(1,4,'uint16')
```

The input type is `uint16(1x4)`.

4 Optionally, after specifying the input type, you can specify that the input is variable size.

Select the second dimension.





- From the list of size options, select `:4` to specify that the second dimension is variable size with an upper bound of 4. Alternatively, select `:Inf` to specify that the second dimension is unbounded.

Alternatively, you can specify that the input is variable size by using the `coder.newtype` function. Enter the following MATLAB expression:

```
coder.newtype('uint16',[1 4],[0 1])
```

---

**Note:** To specify that an input is a double-precision scalar, simply enter `0`.

---

## Specifying an Enumerated Type Input Parameter by Example

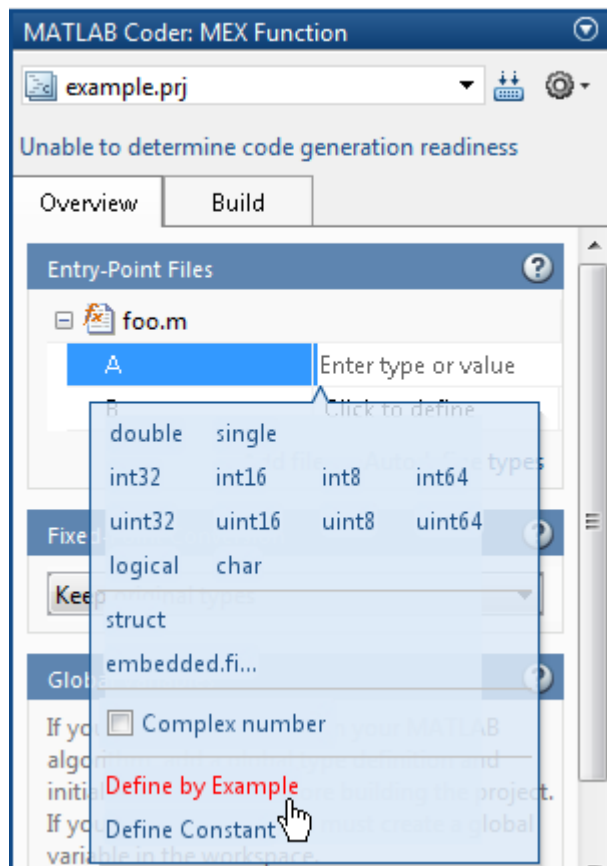
This example shows how to specify that an input uses the enumerated type `MyColors`.

- Define an enumeration `MyColors`. On the MATLAB path, create a file named `'MyColors'` containing:

```
classdef(Enumeration) MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

end

- 2 On the MATLAB Coder project **Overview** tab, click to the input parameter that you want to define.



- 3 From the list of input options, select **Define by Example**.
- 4 In the field to the right of the parameter, enter the following MATLAB expression:

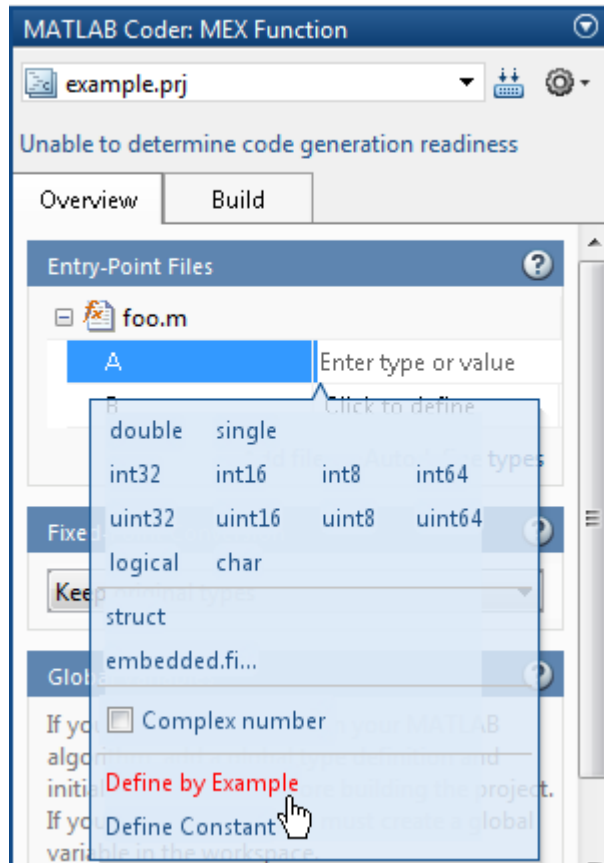
```
MyColors.red
```

## Specifying a Fixed-Point Input Parameter by Example

To specify fixed-point inputs, you must install Fixed-Point Designer software.

This example shows how to specify a signed fixed-point type with a word length of 8 bits, and a fraction length of 3 bits.

- 1 On the MATLAB Coder project **Overview** tab, click the input parameter that you want to define.



- 2 From the list of input options, select **Define by Example**.
- 3 In the field to the right of the parameter, enter:

```
fi(10, 1, 8, 3)
```

MATLAB Coder sets the type of input `u` to `embedded.fi(1x1)`. By default, if you have not specified a local `fimath`, MATLAB Coder uses the default `fimath`. For more information, see “`fimath` for Sharing Arithmetic Rules”.

Optionally, modify the fixed-point properties of the input, see “Specifying a Fixed-Point Input Parameter by Type” on page 16-22 or the size of the input, see “Define or Edit Input Parameter Type in a Project” on page 16-19.

## Define or Edit Input Parameter Type in a Project

### In this section...

- “How to Define or Edit an Input Parameter Type” on page 16-19
- “Specifying an Enumerated Type Input Parameter by Type” on page 16-21
- “Specifying a Fixed-Point Input Parameter by Type” on page 16-22
- “Specifying Structures” on page 16-23

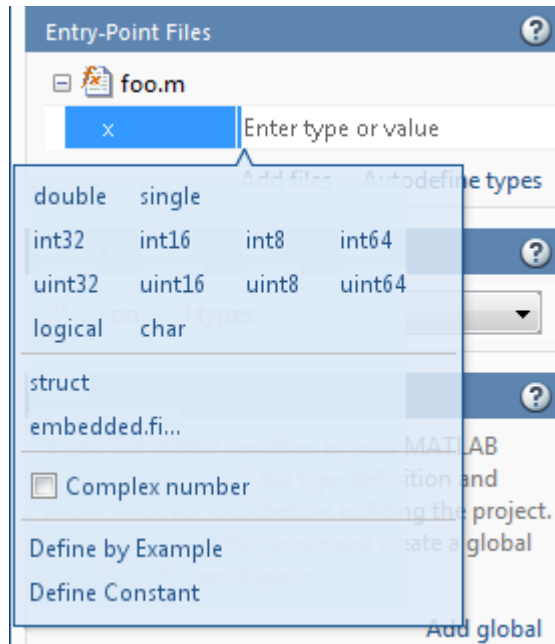
### How to Define or Edit an Input Parameter Type

The following procedure is for input types `double`, `single`, `int64`, `int32`, `int16`, `int8`, `uint64`, `uint32`, `uint16`, `uint8`, `logical`, and `char`.

For more information about defining other types, see the following table.

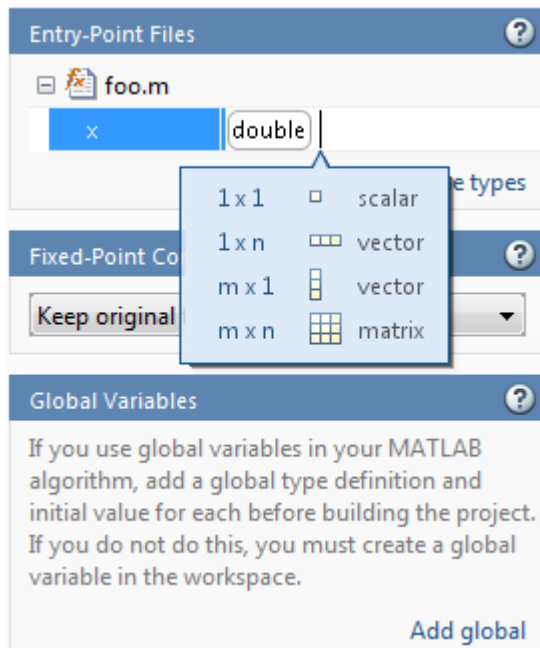
Input Type	Link
A structure ( <code>struct</code> )	“Specifying Structures” on page 16-23
A fixed-point data type (embedded <code>.fi</code> )	“Specifying a Fixed-Point Input Parameter by Type” on page 16-22
An input by example (Define by Example)	“Define Input Parameters by Example in a Project” on page 16-12
A constant (Define Constant)	“Define Constant Input Parameters in a Project” on page 16-29

- 1 On the **Overview** tab **Entry-Point Files** pane, click the field to the right of the input parameter name to view the input options.



- 2 Optionally, for numeric types, select **Complex number** to make the parameter a complex type. By default, inputs are real.
- 3 Select the input type.

The selected type is displayed for the input parameter together with size options.



- 4 From the list, select whether your input is a scalar, a  $1 \times n$  vector, a  $m \times 1$  vector or a  $m \times n$  matrix. By default, if you do not select a size option, MATLAB Coder defines inputs as scalars.
- 5 Optionally, if your input is not scalar, enter sizes  $m$  and  $n$ . You can specify:
  - Fixed size, for example, `10`.
  - Variable size, up to a specified limit, by using the `:` prefix. For example, to specify that your input can vary in size up to 10, enter `:10`.
  - Unbounded variable size by entering `:Inf`.

You can edit the size of each dimension after specifying it.

## Specifying an Enumerated Type Input Parameter by Type

To specify that an input uses the enumerated type `MyColors`:

- 1 Define an enumeration `MyColors`. On the MATLAB path, create a file named `'MyColors'` containing:

```

classdef(Enumeration) MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
end

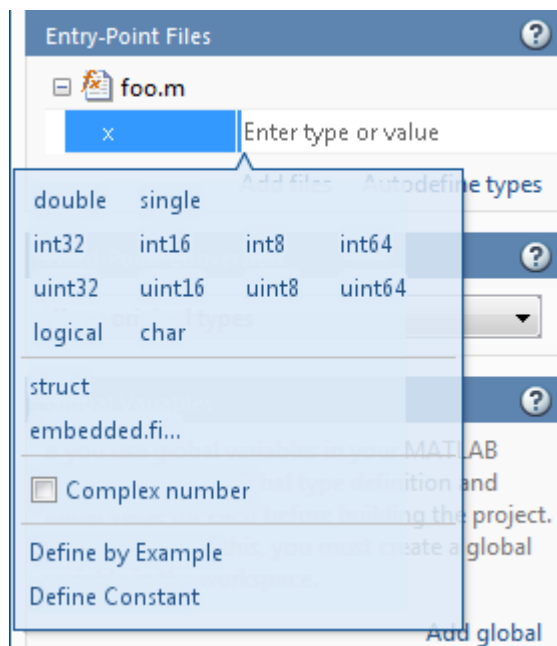
```

- 2 In the field to the right of the input parameter, enter MyColors.

## Specifying a Fixed-Point Input Parameter by Type

To specify fixed-point inputs, you must install Fixed-Point Designer software.

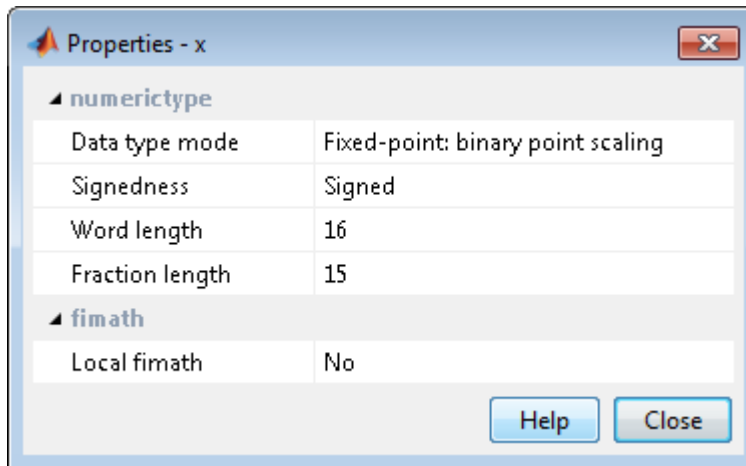
- 1 On the **Overview** tab **Entry-Point Files** pane, click the box to the right of the input parameter name to view the input options.



- 2 Select `embedded.fi.`

The **Properties** dialog box opens.





- 3 In this dialog box, set up the input parameter `numerictype` and `fimath` properties and then close the dialog box.

If you do not specify a local `fimath`, MATLAB Coder uses the default `fimath`. For more information, see “Default `fimath` Usage to Share Arithmetic Rules”.

- 4 The size of the input defaults to  $1 \times 1$ . Optionally, modify the size by selecting the dimension that you want to change and entering a new size.

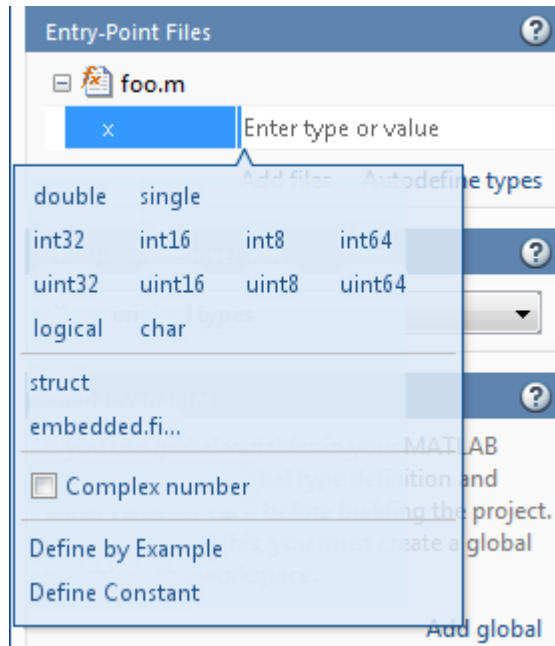
## Specifying Structures

When a primary input is a structure, MATLAB Coder treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition, as follows:

- For each field of input structures, specify class, size, and complexity.
- For each field that is fixed-point class, also specify `numerictype`, and `fimath`.

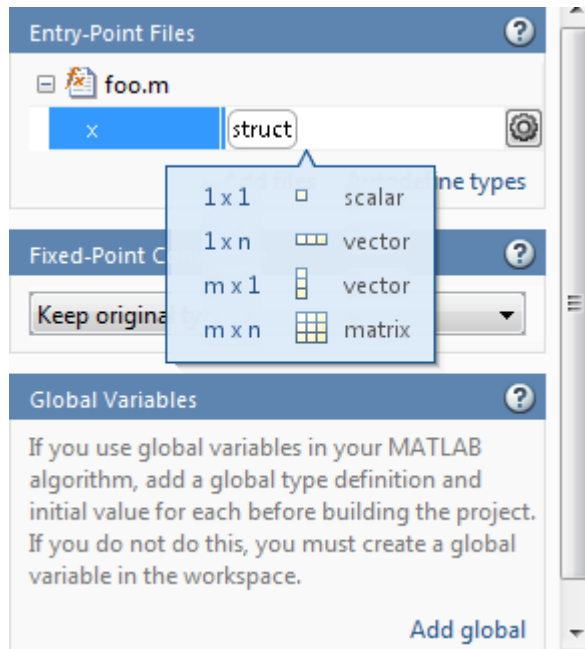
### Specifying Structures by Type

- 1 On the **Overview** tab **Entry-Point Files** pane, click the field to the right of the input parameter name to view the input options.



- 2 From the list of input options, select **struct**.

The selected type, **struct**, is displayed for the input parameter together with size options.

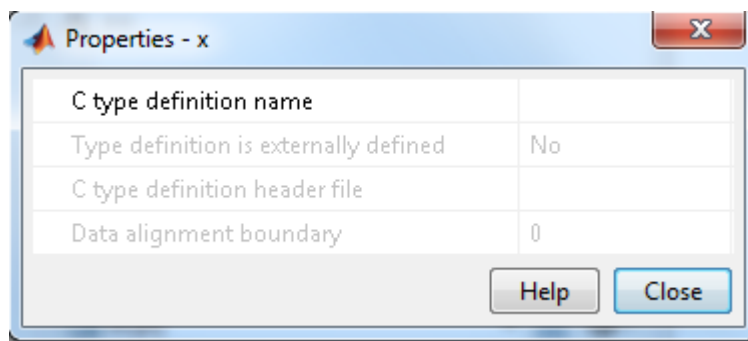


- 3 From the list, select whether your structure is a scalar,  $1 \times n$  vector,  $m \times 1$  vector or  $m \times n$  matrix. By default, if you do not select a size option, MATLAB Coder defines inputs as scalars.
- 4 Optionally, if your input is not scalar, enter sizes  $m$  and  $n$ . You can specify:
  - Fixed size, for example, 10.
  - Variable size, up to a specified limit, by using the `:` prefix. For example, to specify that your input can vary in size up to 10, enter `:10`.
  - Unbounded variable size by entering `:Inf`.
- 5 Optionally, add fields to the structure as described in “How to Add a Field to a Structure” on page 16-28 and then set their size and complexity.
- 6 Optionally, specify properties for the structure in the generated code as described in “How to Set Structure Properties” on page 16-25.

### How to Set Structure Properties

- 1 To the right of the structure definition, click the **Actions** icon, (⚙).

The structure properties dialog box opens.



- 2 Specify properties for the structure in the generated code.


Property	Description
C type definition name	Name to use for the structure variable in the generated code.
Type definition is externally defined	<p>Default: No — type definition is not externally defined</p> <p>If you select 'Yes' to declare an externally defined structure, MATLAB Coder does not generate the definition of the structure type; you must provide it in a custom include file.</p> <p>Dependency: This option is enabled by <b>C type definition name</b>.</p>
C type definition header file	Name of the header file that contains the external definition of the structure, for example, "mystruct.h". Specify the path to the file using the <b>Additional include directories</b> parameter on the <b>Project Settings</b> dialog box <b>Custom Code</b> tab.

Property	Description
	<p>By default, the generated code contains <code>#include</code> statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the C type definition header file option, MATLAB Coder includes that header file exactly at the point where it is required.</p> <p>Must be a non-empty string.</p> <p>Dependency: This option is enabled when <b>Type definition is externally defined</b> is set to <b>Yes</b>.</p>
Data alignment boundary	<p>The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. This capability allows you to take advantage of target-specific function implementations that require data to be aligned. By default, the structure is not aligned on any specific boundary so it will not be matched by CRL functions that require alignment.</p> <p>Alignment must be either -1 or a power of 2 that is no more than 128.</p> <p>Default: 0</p> <p>Dependency: This option is enabled when <b>Type definition is externally defined</b> is set to <b>Yes</b>.</p>

### How to Rename a Field in a Structure

On the project **Overview** tab, select the name field of the structure that you want to rename and enter the new name.


### How to Add a Field to a Structure

- 1 On the project **Overview** tab, select the structure to which you want to add a field.
- 2 To the right of the structure, click the **Actions** icon () to open the context menu.
- 3 From the menu, select **Add Field**.

If the structure already contains other fields, MATLAB Coder adds the field after the existing fields.

- 4 Enter the field name and define its type.


### How to Insert a Field into a Structure

- 1 On the project **Overview** tab, select the field under which you want to add another field.
- 2 To the right of the structure, click the **Actions** icon () to open the context menu.
- 3 From the menu, select **Insert Field**.

MATLAB Coder adds the field after the selected field.

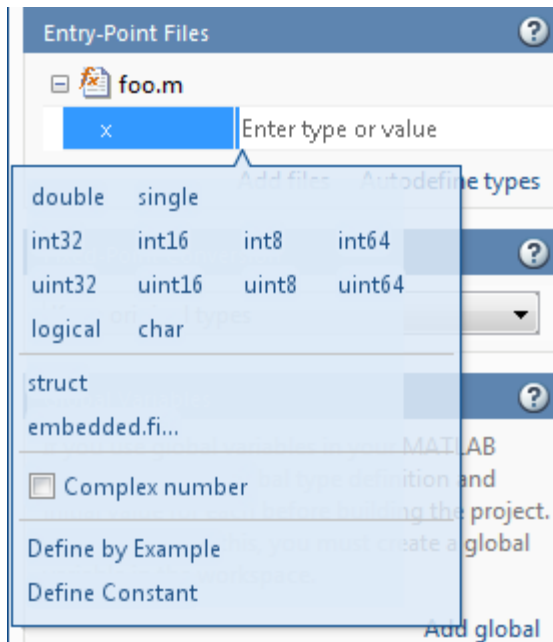
- 4 Enter the field name and define its type.

### How to Remove a Field from a Structure

- 1 In the project **Overview** tab, select the field that you want to remove.
- 2 To the right of the structure, click the **Actions** icon () to open the context menu.
- 3 From the menu, select **Remove Field**.

## Define Constant Input Parameters in a Project

- 1 On the **Overview** tab **Entry-Point Files** pane, click the field to the right of the input parameter name to view the input options.



- 2 Select **Define Constant**.
- 3 In the field to the right of the parameter name, enter the value of the constant or a MATLAB expression that represents the constant.

MATLAB Coder software uses the value of the specified MATLAB expression as a compile-time constant.

## Define Inputs Programmatically in the MATLAB File

You can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB entry-point files.

To enable this option, on the **Project Settings** dialog box **All Settings** tab, under **Advanced**, select **Determine input types from source code preconditions**. If you enable this option:

- MATLAB Coder labels all entry-point function inputs as **Deferred** and determines the input types at compile time.
- You cannot specify input types in this project using any other input specification method.

For more information, see “Define Input Properties Programmatically in the MATLAB File”.



## Adding Global Variables in a Project

To add global variables to the project:

- 1** On the project **Overview** tab, click **Add global**.

By default, MATLAB Coder names the first global variable in a project **g**, and subsequent global variables **g1**, **g2**, etc.

- 2** Enter the name of the global variable.
- 3** After adding a global variable, before building the project, specify its type and initial value. If you do not do this, you must create a variable with the same name in the global workspace. See “Specifying Global Variable Type and Initial Value in a Project” on page 16-32.

## Specifying Global Variable Type and Initial Value in a Project

### In this section...

“Why Specify a Type Definition for Global Variables?” on page 16-32

“How to Specify a Global Variable Type” on page 16-32

“Defining a Global Variable by Example” on page 16-33

“Defining or Editing Global Variable Type” on page 16-34

“Defining Global Variable Initial Value” on page 16-36

“Defining Global Variable Constant Value” on page 16-38

“Removing Global Variables” on page 16-39

### Why Specify a Type Definition for Global Variables?

If you use global variables in your MATLAB algorithm, before building the project, you must add a global type definition and initial value for each. If you do not initialize the global data, MATLAB Coder looks for the variable in the MATLAB global workspace. If the variable does not exist, MATLAB Coder generates an error.

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable. At code generation time, MATLAB Coder needs to have an initial value to determine the type of a global variable. Otherwise, the global variable might be used before it is defined and then MATLAB Coder cannot determine the type to use in the generated code.

For MEX functions, if you use global data, you must also specify whether to synchronize this data between MATLAB and the MEX function. For more information, see “Synchronizing Global Data with MATLAB”.

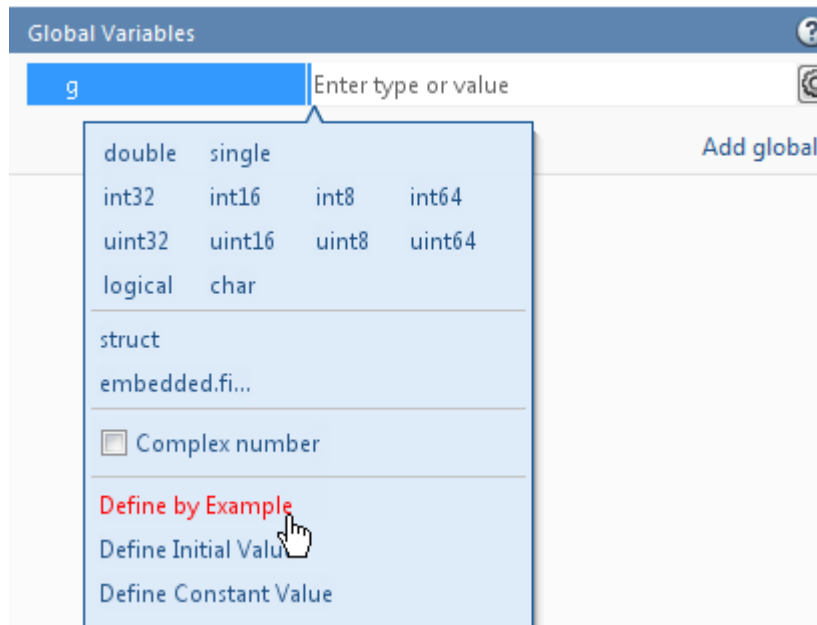
### How to Specify a Global Variable Type

- 1 Specify the type of each global variable using one of the following methods:
  - Define by example
  - Define type
- 2 Define an initial value for each global variable.

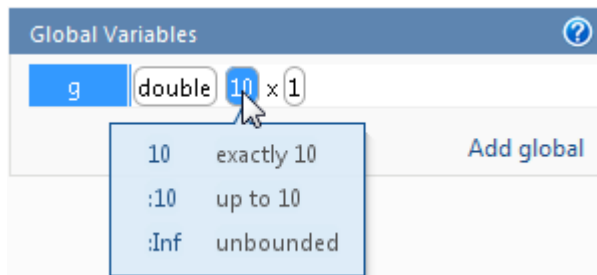
If you do not provide a type definition and initial value for a global variable, you **must** create a variable with the same name and suitable class, size, complexity, and value in the MATLAB workspace.

## Defining a Global Variable by Example

- 1 On the project **Overview** tab, click the field to the right of the global variable that you want to define.



- 2 Select **Define by Example**.
- 3 In the field to the right of the global name, enter a MATLAB expression that has the required class, size, and complexity. MATLAB Coder software uses the class, size, and complexity of the value of this expression as the type for the global variable.
- 4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, 10.



You can specify:

- Fixed size. In this example, select 10.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to 10, select `:10`.
- Unbounded variable size by selecting `:Inf`.

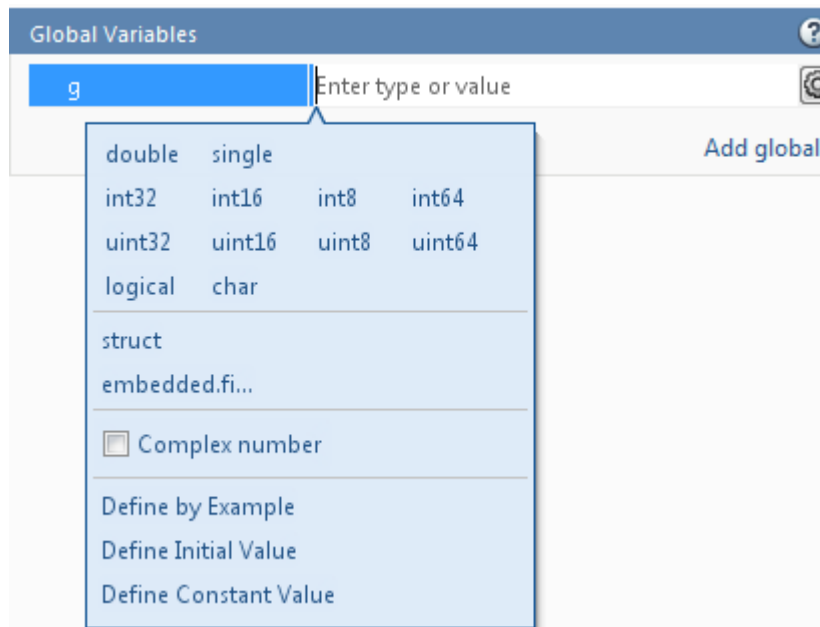
---

**Note:** You define global variables in the same way that you define input parameters. For more information, see “Define Input Parameters by Example in a Project” on page 16-12

---

## Defining or Editing Global Variable Type

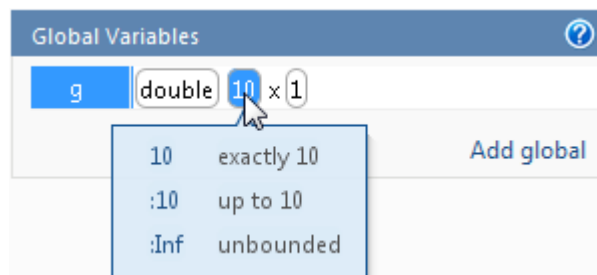
- 1 On the project **Overview** tab, click the field to the right of the global variable that you want to define.



- 2 Optionally, for numeric types, select **Complex** to make the parameter a complex type. By default, inputs are real.
- 3 Select the type for the global variable. For example, double.

By default, the global variable is a scalar.

- 4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, 10.



You can specify:

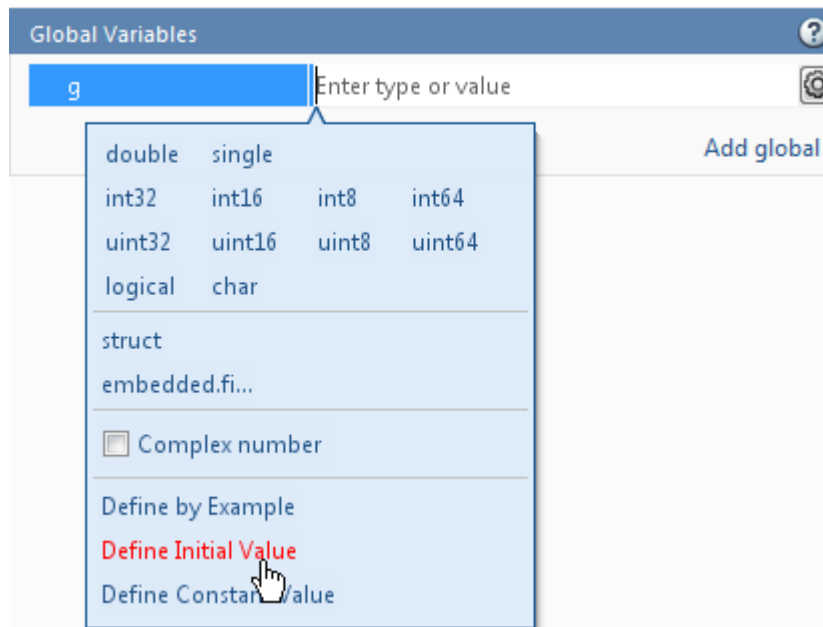
- Fixed size. In this example, select 10.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to 10, select `:10`.
- Unbounded variable size by selecting `:Inf`.

## Defining Global Variable Initial Value

- “Define Initial Value Before Defining Type” on page 16-36
- “Define Initial Value After Defining Type” on page 16-37

### Define Initial Value Before Defining Type

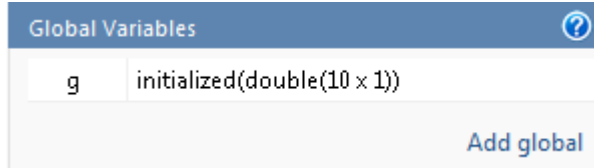
- 1 On the project **Overview** tab, click the field to the right of the global variable.



- 2 Select **Define Initial Value**.
- 3 Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable. Because you

did not define the type of the global variable before you defined its initial value, MATLAB Coder uses the type of the initial value as the global variable type.

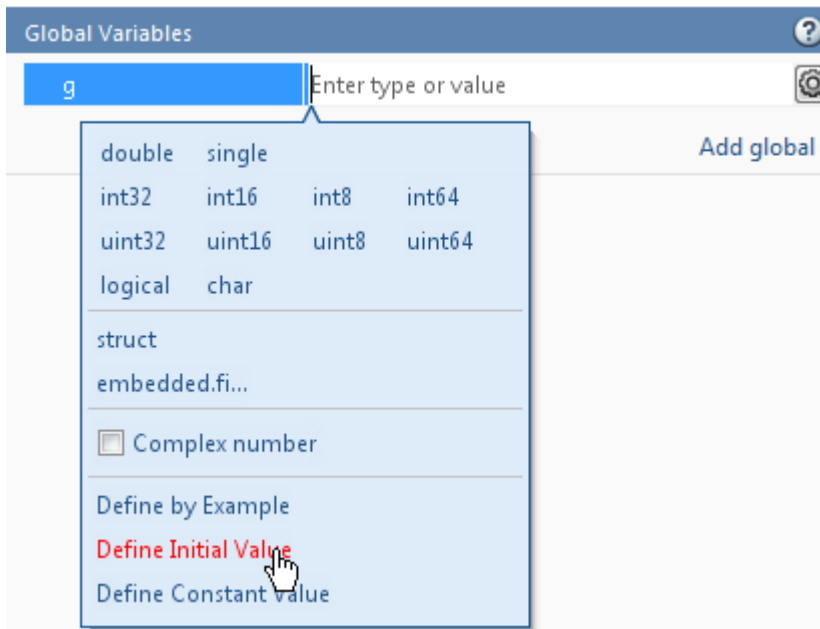
The project displays that the global variable is initialized.



If you change the type of a global variable after defining its initial value, you must redefine the initial value.

### Define Initial Value After Defining Type

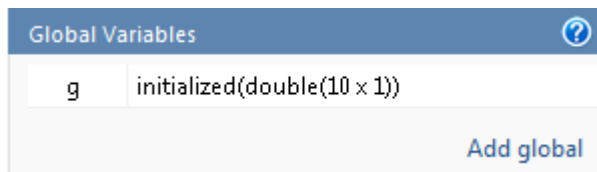
- 1 On the project **Overview** tab, click the type field of the global variable.



- 2 Select **Define Initial Value**.

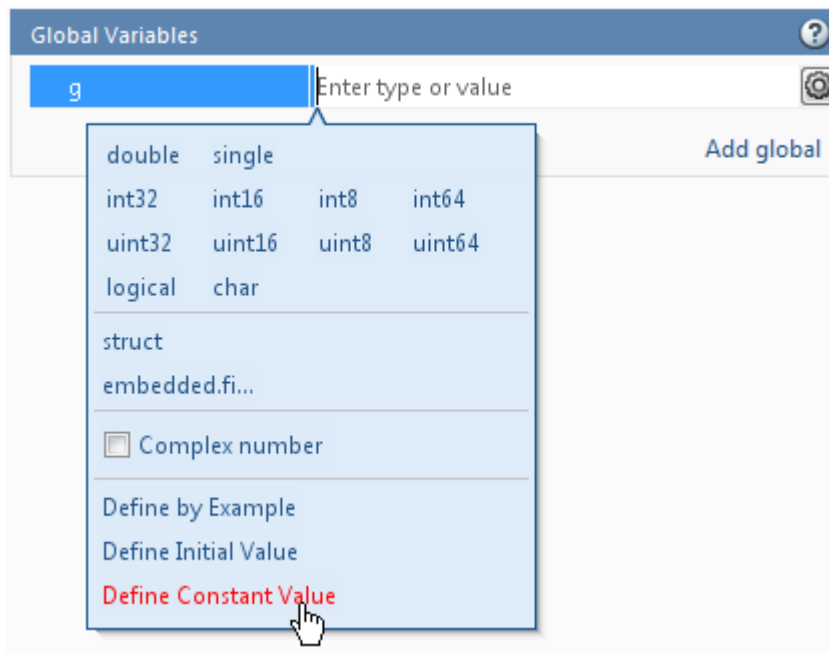
- 3 Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

The project displays that the global variable is initialized.



## Defining Global Variable Constant Value

- 1 On the project **Overview** tab, click the field to the right of the global variable.



- 2 Select **Define Constant Value**.
- 3 In the field to the right of the global variable, enter a MATLAB expression.



## Removing Global Variables

- 1 On the project **Overview** tab, select the global variable that you want to remove.
- 2 To the right of the variable, click the **Actions** icon (⚙) to open the context menu.
- 3 From this menu, select **Remove Global**.

MATLAB Coder removes the global variable.

## Specify Output File Name

On the project **Build** tab, in the **Output file** field, enter the file name. The file name can include an existing path.

---

**Note:** Do not put any spaces in the file name.

---

By default, if the name of the first entry-point MATLAB file is *fcn1*, the output file name is:

- *fcn1* for C/C++ libraries and executables.
- *fcn1\_mex* for MEX functions.

By default, MATLAB Coder generates files in the folder *project\_folder/codegen/target/fcn1*:

- *project\_folder* is your current project folder
- *target* is:
  - *mex* for MEX functions
  - *lib* for static C/C++ libraries
  - *dll* for dynamic C/C++ libraries
  - *exe* for C/C++ executables

To learn how to change the default output folder, see “Specify Output File Locations” on page 16-41.

### Command Line Alternative

Use the `codegen` function `-o` option.

## Specify Output File Locations

The path should not contain:

- Spaces, as this can lead to code generation failures in certain operating system configurations.
- Non 7-bit ASCII characters, such as Japanese characters.

- 1** On the project **Build** tab, click **More settings**.
- 2** In the Project Settings dialog box, click the **Paths** tab.

The default setting for the **Build Folder** field is A subfolder of the project folder. By default, MATLAB Coder generates files in the folder *project\_folder/codegen/target/fcn1*:

- `fcn1` is the name of the first entry-point file
- `target` is:
  - `mex` for MEX functions
  - `dll` for dynamic C/C++ libraries
  - `lib` for static C/C++ libraries
  - `exe` for C/C++ executables

- 3** To change the output location, you can either:

- Set **Build Folder** to A subfolder of the current MATLAB working folder

MATLAB Coder generates files in the *MATLAB\_working\_folder/codegen/target/fcn1* folder

- Set **Build Folder** to Specified folder. In the **Build folder name** field, provide the path to the folder.

### Command Line Alternative

Use the `codegen` function `-d` option.

## Selecting Output Type

On the project **Build** tab, from the **Output type** drop-down list, select one of the available output types:

- MEX Function (default)
- Instrumented MEX Function

Building an instrumented MEX function requires a Fixed-Point Designer license and clears prior instrumentation results.

- C/C++ Static Library
- C/C++ Dynamic Library
- C/C++ Executable

### Command Line Alternative

Use the `codegen` function `-config` option.

### Changing Output Type

MEX functions use a different set of configuration parameters than C/C++ libraries and executables use. When you switch the output type between MEX Function or Instrumented MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, you should verify these settings.

If you enable any of the following parameters when the output type is MEX Function or Instrumented MEX Function, and you want to use the same setting for C/C++ code generation as well, you must enable it again for C/C++ Static Library, C/C++ Dynamic Library, and C/C++ Executable.

#### Check These MATLAB Coder Project Parameters When Changing Output Type

Project Settings Dialog Box Tab	Parameter Name
Paths	Working folder
	Build folder
	Search paths
Speed	Saturate on integer overflow
Memory	Enable variable-sizing

Project Settings Dialog Box Tab	Parameter Name
	Dynamic memory allocation
	Stack usage max
Code Appearance	Generated file partitioning method
	Include comments
	MATLAB source code as comments
	Reserved names
Debugging	Always create a code generation report
	Automatically launch a report if one is generated
Custom Code	Source file
	Header file
	Initialize function
	Terminate function
	Additional include directories
	Additional source files
	Additional libraries
	Post-code-generation command
Advanced	Constant folding timeout
	Language
	Inline threshold
	Inline threshold max
	Inline stack limit
	Use memcpy for vector assignment
	Memcpy threshold (bytes)
	Use memset to initialize floats and doubles to 0.0

### Check These Command-Line Parameters When Changing Output Type

When you switch between MEX and C output types, check these `coder.MexCodeConfig`, `coder.CodeConfig` or `coder.EmbeddedCodeConfig` configuration object parameters, as applicable.

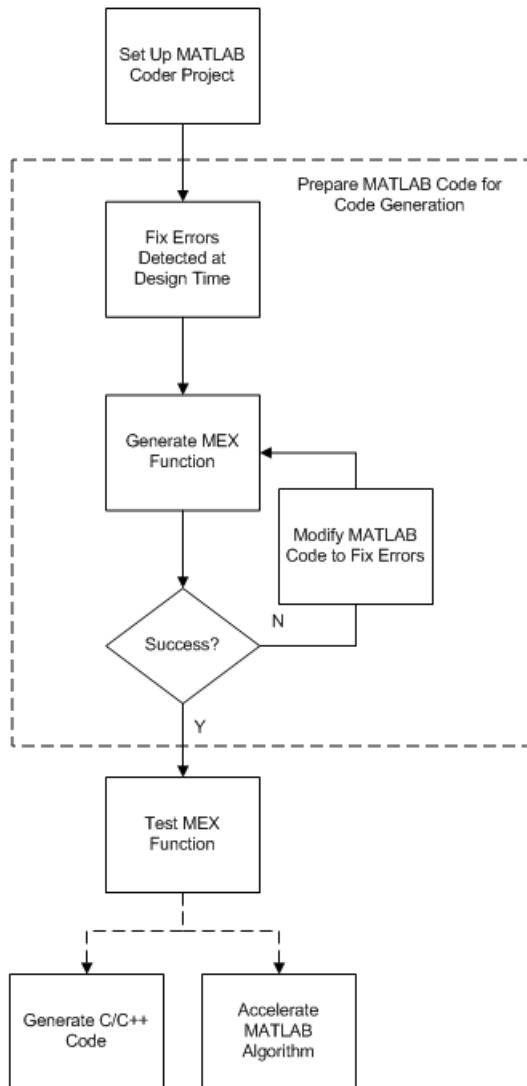
- ConstantFoldingTimeout
- CustomHeaderCode
- CustomInclude
- CustomInitializer
- CustomLibrary
- CustomSource
- CustomSourceCode
- CustomTerminator
- DynamicMemoryAllocation
- EnableMemcpy
- EnableVariableSizing
- FilePartitionMethod
- GenCodeOnly
- GenerateComments
- GenerateReport
- InitFltsAndDblsToZero
- InlineStackLimit
- InlineThreshold
- InlineThresholdMax
- LaunchReport
- MATLABSourceComments
- MemcpyThreshold
- PostCodeGenCommand
- ReservedNameArray
- SaturateOnIntegerOverflow
- StackUsageMax
- TargetLang

# Preparing MATLAB Code for C/C++ Code Generation

---

- “Workflow for Preparing MATLAB Code for Code Generation” on page 17-2
- “Fixing Errors Detected at Design Time” on page 17-4
- “Using the Code Analyzer” on page 17-5
- “Check Code With the Code Analyzer” on page 17-6
- “Check Code Using the Code Generation Readiness Tool” on page 17-8
- “Code Generation Readiness Tool” on page 17-10
- “Unable to Determine Code Generation Readiness” on page 17-17
- “Generate MEX Functions Using the MATLAB Coder Project Interface” on page 17-18
- “Generate MEX Functions at the Command Line” on page 17-26
- “Fix Errors Detected at Code Generation Time” on page 17-28
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 17-29
- “Running MEX Functions” on page 17-31
- “Debugging Strategies” on page 17-32

## Workflow for Preparing MATLAB Code for Code Generation





## See Also

- “MATLAB Coder Project Set Up Workflow”
- “Fixing Errors Detected at Design Time” on page 17-4
- “Generate MEX Functions Using the MATLAB Coder Project Interface”
- “Fix Errors Detected at Code Generation Time” on page 17-28
- “Workflow for Testing MEX Functions in MATLAB”
- “C/C++ Code Generation”
- “Accelerate MATLAB Algorithms”

## Fixing Errors Detected at Design Time

Use the code analyzer and the code generation readiness tool to detect issues at design time. Before generating code, you must fix these issues.

### See Also

- “Check Code With the Code Analyzer” on page 17-6
- “Check Code Using the Code Generation Readiness Tool” on page 17-8
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 17-29
- “Debugging Strategies” on page 17-32

## Using the Code Analyzer

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing compilation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications.

To use the code analyzer to identify warnings and errors specific to MATLAB for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of code generation analyzer messages is available in the MATLAB Code Analyzer preferences. For more information, see “Running the Code Analyzer Report”.

---

**Note:** The code analyzer might not detect all MATLAB for code generation issues. After eliminating the errors or warnings that the code analyzer detects, compile your code with MATLAB Coder to determine if the code has other compliance issues.

---

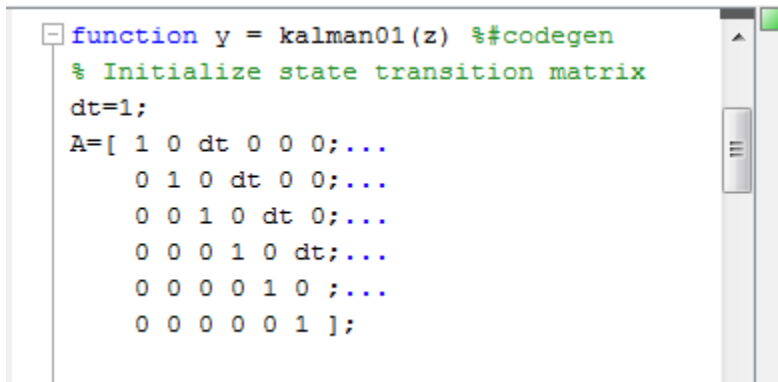
## Check Code With the Code Analyzer

The code analyzer checks your code for problems and recommends modifications. You can use the code analyzer to check your code interactively in the MATLAB Editor while you work.

To verify that continuous code checking is enabled:

- 1 In MATLAB, select the **Home** tab and then click **Preferences**.
- 2 In the **Preferences** dialog box, select **Code Analyzer**.
- 3 In the **Code Analyzer Preferences** pane, verify that **Enable integrated warning and error messages** is selected.

The code analyzer provides an indicator in the top right of the editor window. If the indicator is green, the analyzer did not detect code generation issues.

A screenshot of the MATLAB Code Analyzer interface. The main window displays MATLAB code for a function named 'kalman01'. The code includes a comment '%#codegen' and another comment '% Initialize state transition matrix'. Below the comments, the variable 'dt' is set to 1, and a matrix 'A' is defined with several rows of values. To the right of the code editor, there is a vertical toolbar with a green square indicator at the top, indicating that no code generation issues were detected.

```
function y = kalman01(z) %#codegen
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

If the indicator is red, the analyzer has detected errors in your code. If it is orange, it has detected warning. When the indicator is red or orange, a red or orange marker appears to the right of the code where the error occurs. Place your pointer over the marker for information about the error. Click the underlined text in the error message for a more detailed explanation and suggested actions to fix the error.

```
p_prd = A * p_est * A' + Q;

% Estimation
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';

% Estimated state and covariance
x_est = x_prd + klm_gain * (z(1:2,i) - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;

% Compute the estimated measurements
v(:,i) = H * x_est;
end
end
```

Line 46: Code generation requires variable 'y' to be fully defined before subscripting it.

Line 46: Code generation does not support variable 'y' size growth through indexing.

Before generating code from your MATLAB code, you must fix the errors detected by the code analyzer.

## Check Code Using the Code Generation Readiness Tool

### In this section...

“Run Code Generation Readiness Tool at the Command Line” on page 17-8

“Run Code Generation Readiness Tool from the Current Folder Browser” on page 17-8

“Run the Code Generation Readiness Tool in a Project” on page 17-8

“See Also” on page 17-9

### Run Code Generation Readiness Tool at the Command Line

- 1 Navigate to the folder that contains the file that you want to check for code generation readiness.
- 2 At the MATLAB command prompt, enter:

```
coder.screener('filename')
```

The **Code Generation Readiness** tool opens for the file named `filename`, provides a code generation readiness score, and lists issues that must be fixed prior to code generation.

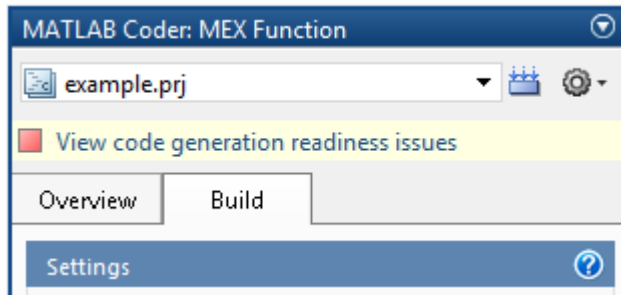
### Run Code Generation Readiness Tool from the Current Folder Browser

- 1 In the current folder browser, right-click the file that you want to check for code generation readiness.
- 2 From the context menu, select **Check Code Generation Readiness**.

The **Code Generation Readiness** tool opens for the selected file. It provides a code generation readiness score and lists issues that must be fixed prior to code generation.

### Run the Code Generation Readiness Tool in a Project

- 1 After you have added entry-point files to the project, if MATLAB Coder detects code generation issues, it displays a link at the top of the project window.



- 2 Click the link to open the **Code Generation Readiness** tool.

The tool opens and provides a code generation readiness score and lists issues that must be fixed prior to code generation.

## See Also

- “Code Generation Readiness Tool” on page 17-10

## Code Generation Readiness Tool

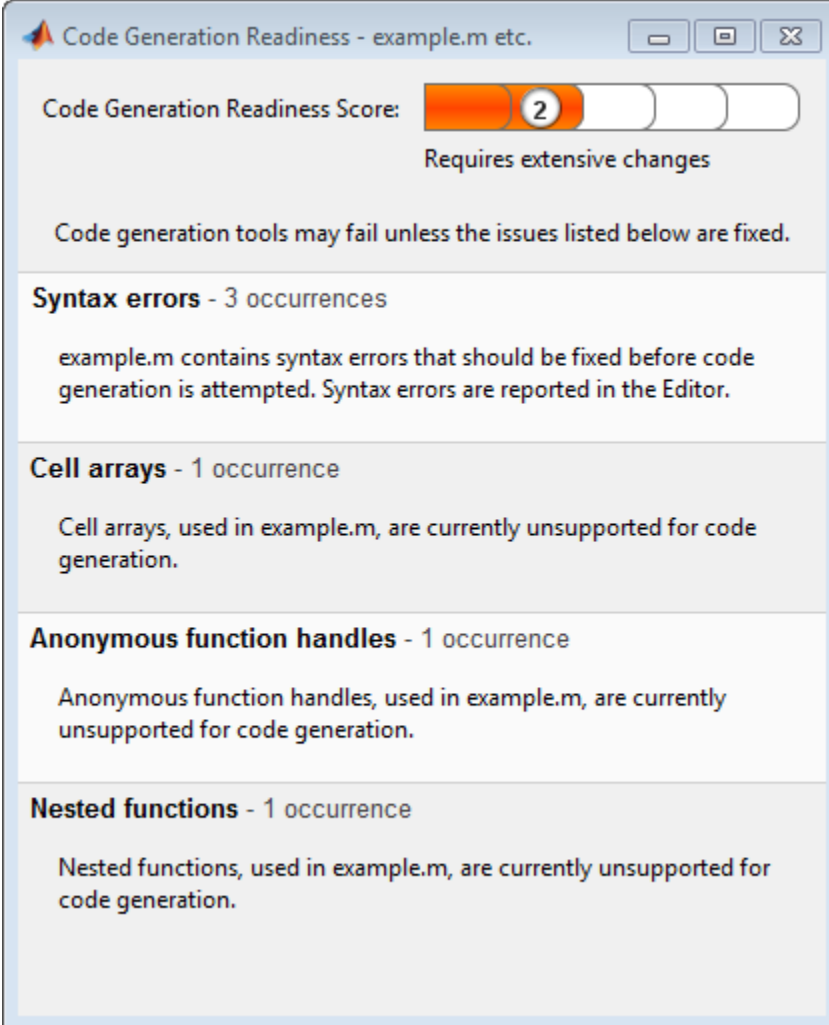
In this section...
“What Information Does the Code Generation Readiness Tool Provide?” on page 17-10
“Summary Tab” on page 17-11
“Code Structure Tab” on page 17-13
“See Also” on page 17-16

### What Information Does the Code Generation Readiness Tool Provide?

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. The report also provides an indication of how much work you must do to make the MATLAB code suitable for code generation. The tool might not detect all code generation issues. Under certain circumstances, it might report false errors. Because the tool might not detect all issues, or might report false errors, generate a MEX function to verify that your code is suitable for code generation before generating C code.



## Summary Tab



The screenshot shows a window titled "Code Generation Readiness - example.m etc." with standard window controls. The main content area displays a "Code Generation Readiness Score" of 2, represented by a progress bar with 5 segments, the first two of which are filled with orange. Below the score, it states "Requires extensive changes". A warning message reads: "Code generation tools may fail unless the issues listed below are fixed." The issues are listed in a scrollable area:

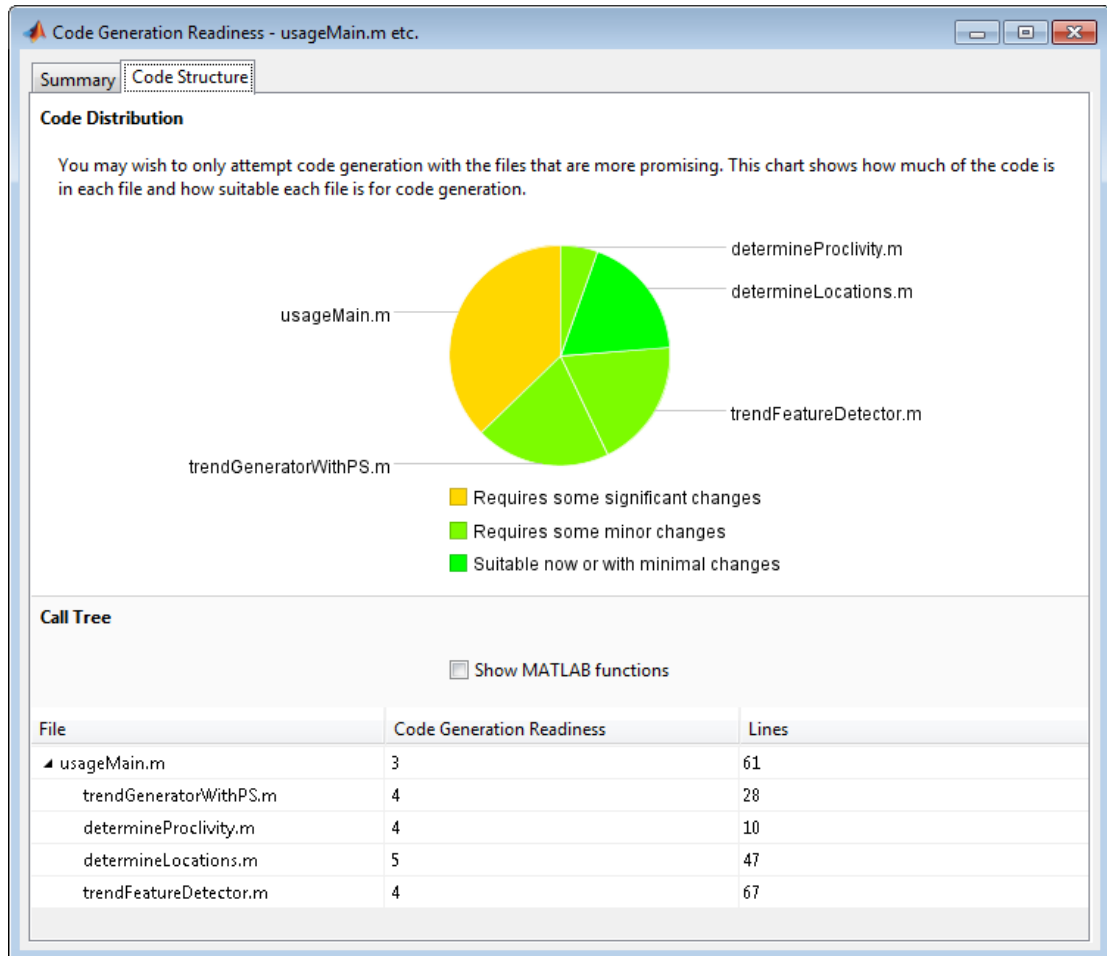
- Syntax errors** - 3 occurrences  
example.m contains syntax errors that should be fixed before code generation is attempted. Syntax errors are reported in the Editor.
- Cell arrays** - 1 occurrence  
Cell arrays, used in example.m, are currently unsupported for code generation.
- Anonymous function handles** - 1 occurrence  
Anonymous function handles, used in example.m, are currently unsupported for code generation.
- Nested functions** - 1 occurrence  
Nested functions, used in example.m, are currently unsupported for code generation.

The **Summary** tab provides a **Code Generation Readiness Score** which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes.

On this tab, the tool also provides information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. Use the code analyzer to learn more about the issues and how to fix them.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features, such as recursion, cell arrays, and nested functions.
- Unsupported data types.

## Code Structure Tab



If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab provides information about the relative size of each file and how suitable each file is for code generation.

### Code Distribution

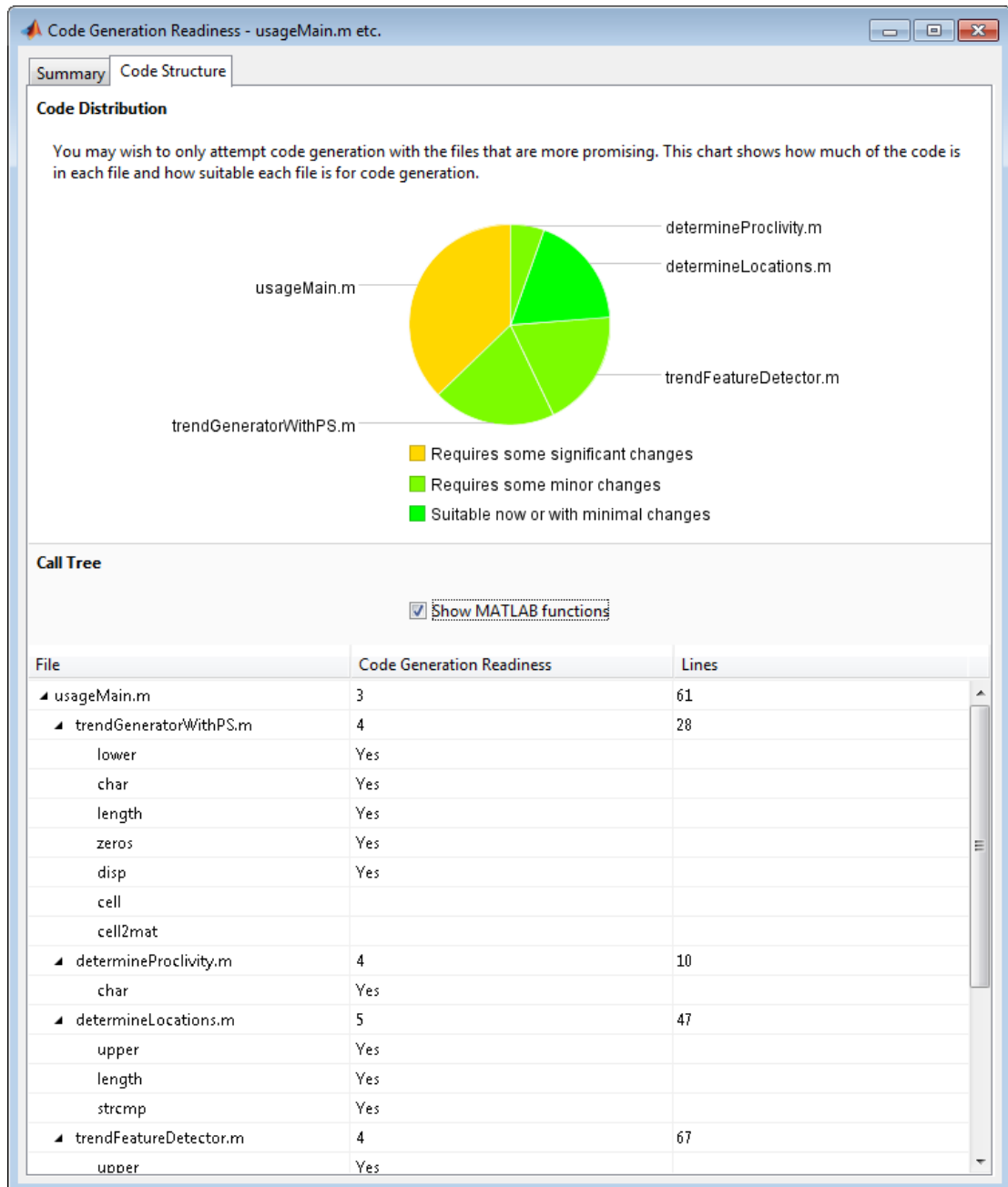
The **Code Distribution** pane provides a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. This information is useful during the planning phase of a project for estimation and scheduling purposes. If the report indicates that there are multiple files not yet suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

### Call Tree

The **Call Tree** pane provides information on the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score which ranges from 1 to 5. A score of 1 indicates that the tool has detected issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool has not detected code generation issues; the code is ready to use with no or minimal changes. The report also lists the number of lines of code in each file.

### Show MATLAB Functions

If you select **Show MATLAB Functions**, the report also lists the MATLAB functions called by your function code. For each of these MATLAB functions, if the function is supported for code generation, the report sets **Code Generation Readiness** to **Yes**.



### See Also

- “Check Code Using the Code Generation Readiness Tool”

## Unable to Determine Code Generation Readiness

Sometimes the code generation readiness tool cannot determine whether the entry-point functions in your project are suitable for code generation. The most likely reason is that the tool is unable to find the entry-point files. Verify that your current working folder is set to the folder that contains your entry-point files. If it is not, either make this folder your current working folder or add the folder containing these files to the MATLAB path.

## Generate MEX Functions Using the MATLAB Coder Project Interface

### In this section...

“Project Workflow for Generating MEX Functions” on page 17-18

“Generate MEX Functions Using the Project Interface” on page 17-18

“Configure Project Settings” on page 17-23

“Build a MATLAB Coder Project” on page 17-24

“See Also” on page 17-25

### Project Workflow for Generating MEX Functions

Step	Action	Details
1	Set up your MATLAB Coder project.	“Creating a New Project”
2	Fix errors detected by the code analyzer.	“Fixing Errors Detected at Design Time” on page 17-4
3	Specify build configuration parameters.	“Configure Project Settings” on page 17-23
4	Build the project.	“Build a MATLAB Coder Project” on page 17-24

### Generate MEX Functions Using the Project Interface

In this example, you create a MATLAB function that adds two numbers, then create a MATLAB Coder project for this file. Using the project user interface, you specify types for the function input parameters, and then generate a MEX function for the MATLAB code.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

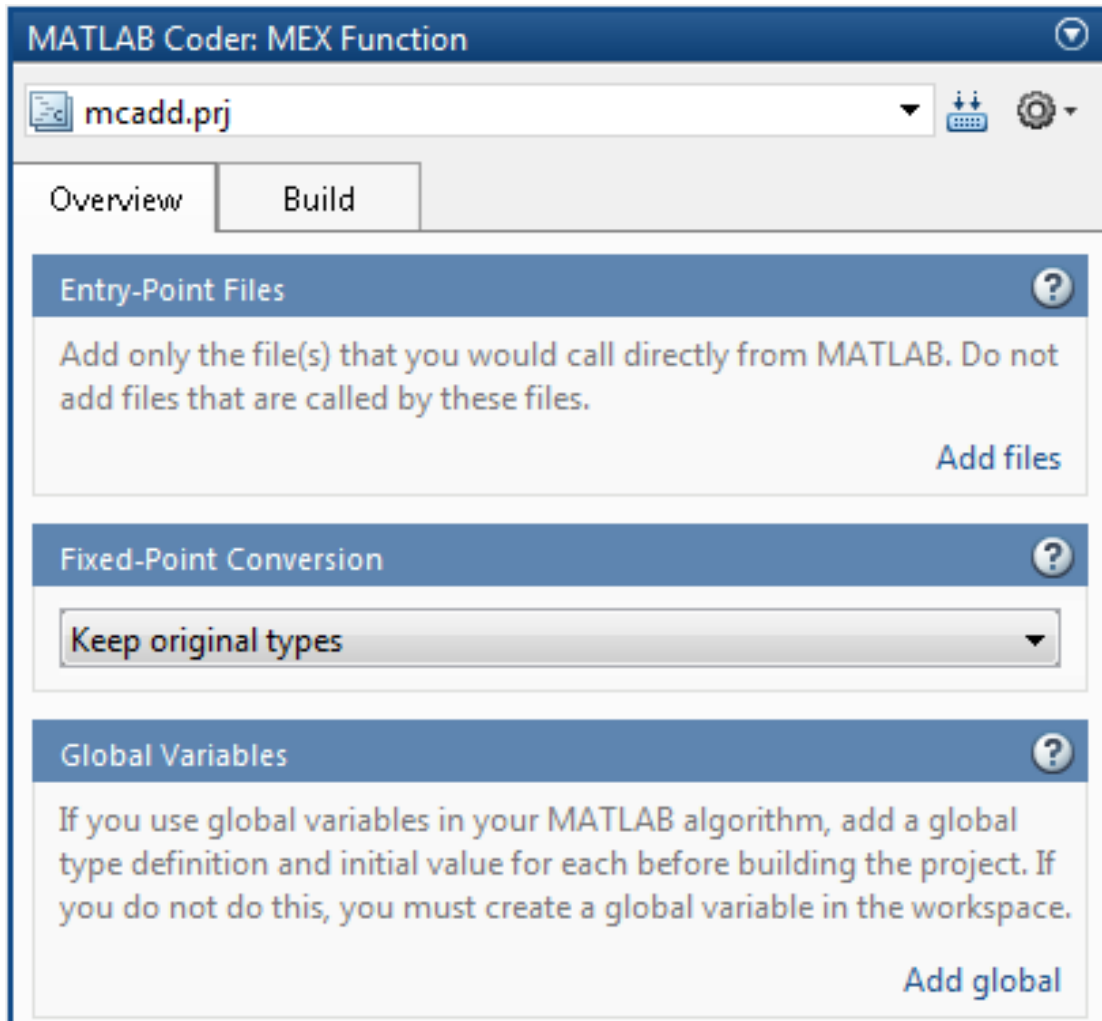
- 2 In the same folder, set up a MATLAB Coder project.

- a At the MATLAB command line, enter

```
coder -new mcadd.prj
```

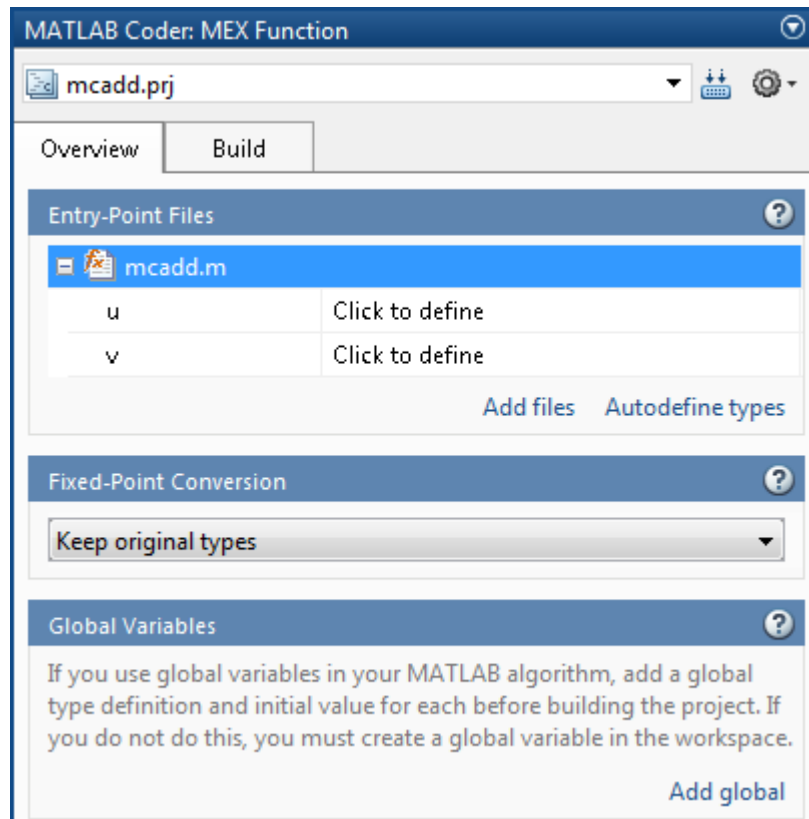


By default, the project opens in the MATLAB workspace on the right side.

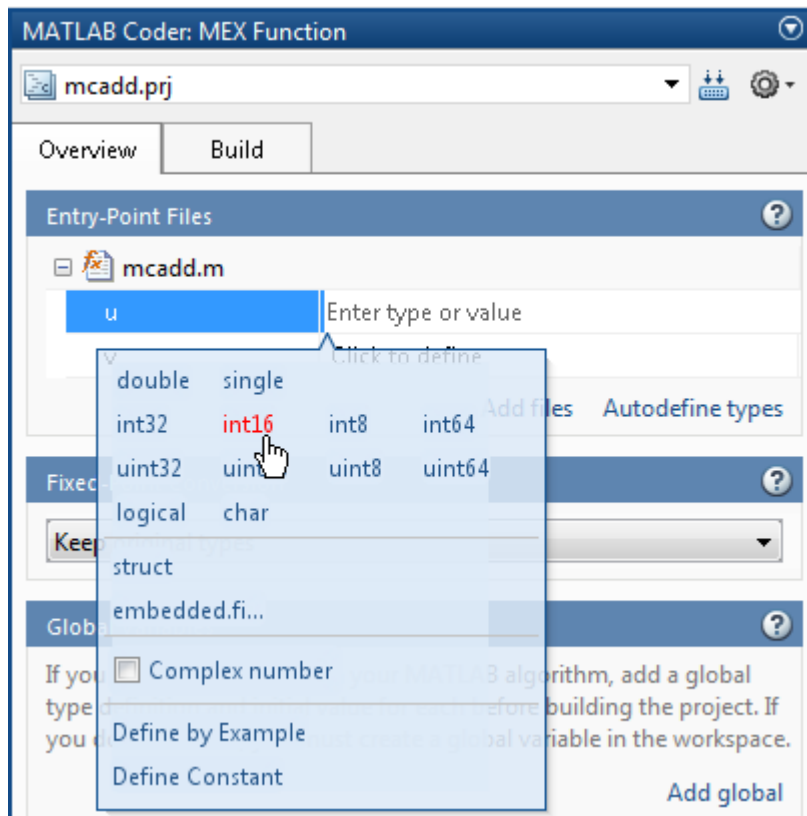


- b** On the project **Overview** tab, click the **Add files** link, browse to the file `mcadd.m`, and click **Open** to add the file to the project.

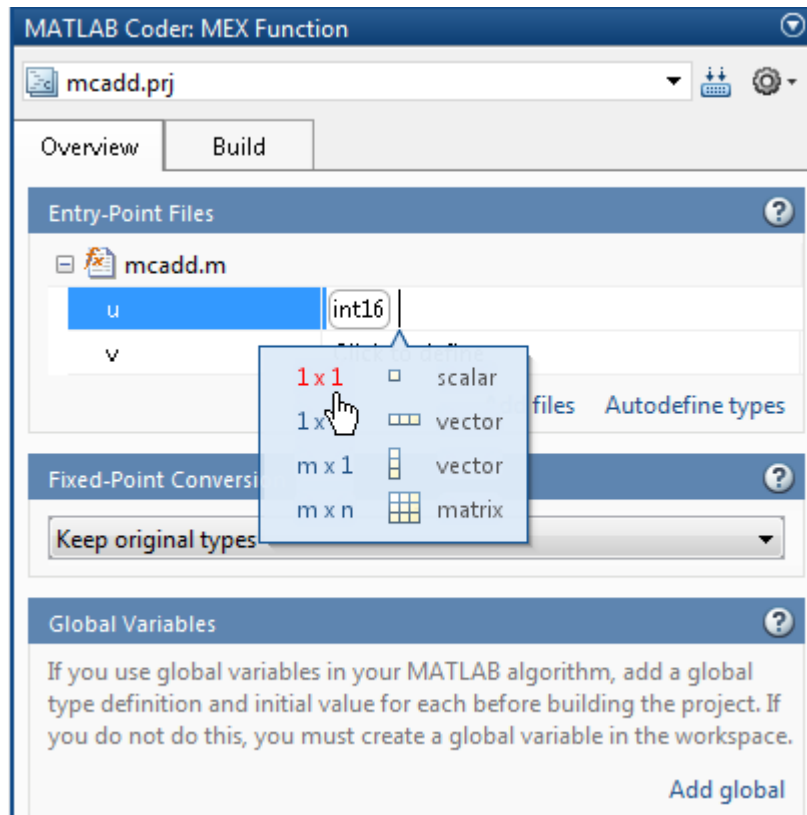
The file is displayed on the **Overview** tab, and both inputs are undefined.



- c On the **Overview** tab, click the field to the right of the input parameter `u` and, from the list of input options, select `int16`.

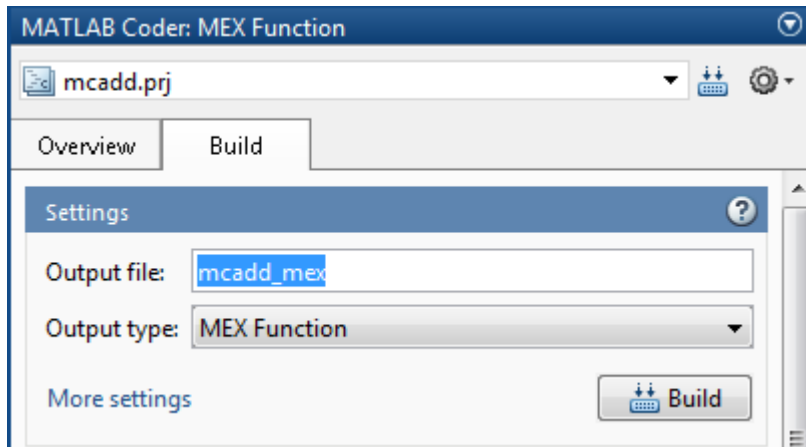


- d From the list of size options, select 1 x 1 to specify that the input is a scalar.



- e Repeat the previous two steps to specify the input  $v$ .
- 3 In the MATLAB Coder project, click the **Build** tab.

By default, the **Output type** is MEX function and the **Output file** is mcadd\_mex.



- 4 On this tab, click the **Build** button to generate a MEX function using the default project settings.

MATLAB Coder builds the project and, by default, generates a MEX function, `mcadd_mex`, in the current folder. MATLAB Coder also generates other supporting files in a subfolder called `codegen/mex/mcadd`. MATLAB Coder uses the name of the MATLAB function as the root name for the generated files and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 19-66.

You can now test your MEX function in MATLAB. For more information, see “Verify MEX Functions in a Project”.

## Configure Project Settings

- 1 On the project **Build** tab, click the **More settings** link to view the project settings for the selected output type.

---

**Note:** MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you change the output type from MEX Function or Instrumented MEX Function to C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

---

- 2 In the **Project Settings** dialog box, select the settings that you want to apply.

---

**Tip** To learn more about the configuration parameters on the current tab of the **Project Settings** dialog box, click the **Help** button.

---

### See Also

- “How to Enable Code Generation Reports in the Project Settings Dialog Box” on page 19-170
- “In the Project Settings Dialog Box” on page 19-114
- “How to Disable Inlining Globally in the Project Settings Dialog Box” on page 19-124
- “Generate Traceable Code” on page 19-84
- “Disabling Run-Time Checks in the Project Settings Dialog Box” on page 23-17

## Build a MATLAB Coder Project

On the project **Build** tab, click the **Build** button to build the project using the specified settings. While MATLAB Coder builds a project, it displays the build progress in the Build dialog box. When the build is complete, MATLAB Coder provides details in the **Build Results** pane.

### Viewing Build Results

The **Build Results** pane provides information about the most recent build. If the code generation report is enabled or build errors occur, MATLAB Coder generates a report that provides detailed information about the most recent build and provides a link to the report.

To view the report, click the **View report** link. After a build completes, this report provides links to your MATLAB code and generated C/C++ files as well as compile-time type information for the variables in your MATLAB code. If build errors occur, it lists errors and warnings.

### Saving Build Results

When MATLAB Coder builds a project, it displays the build progress and results in the Build dialog box. To save the build results, click the **Save to log file** link and specify the log file location.

### **See Also**

- “Code Generation Reports”
- “Generate Code for Multiple Entry-Point Functions” on page 19-70
- “Generate Code for Global Data” on page 19-75

### **See Also**

- “Generate Code for Multiple Entry-Point Functions” on page 19-70
- “Generate Code for Global Data” on page 19-75
- “Specify Output File Name”
- “Specify Output File Locations”

## Generate MEX Functions at the Command Line

### Command-line Workflow for Generating MEX Functions

Step	Action	Details
1	Install prerequisite products.	“Installing Prerequisite Products”
2	Set up your file infrastructure.	“Paths and File Infrastructure Setup” on page 19-65
3	Fix errors detected by the code analyzer.	“Fixing Errors Detected at Design Time” on page 17-4
4	Specify build configuration parameters.	“Specify Build Configuration Parameters” on page 19-25
5	Specify properties of primary function inputs.	“Primary Function Input Specification” on page 19-39
6	Generate the MEX function using <code>codegen</code> with suitable command-line options.	“Generating MEX Functions at the Command Line Using <code>codegen</code> ” on page 17-27

### Generate MEX Functions at the Command Line

In this example, you use the `codegen` function to generate a MEX function from a MATLAB file that adds two inputs. You use the `codegen -args` option to specify that both inputs are `int16`.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 Generate a platform-specific MEX function in the current folder. At the command line, specify that the two input parameters are `int16` using the `-args` option. By default, if you do not use the `-args` option, `codegen` treats inputs as real, scalar doubles.

```
codegen mcadd -args {int16(0), int16(0)}
```

`codegen` generates a MEX function, `mcadd_mex`, in the current folder. `codegen` also generates other supporting files in a subfolder called `codegen/mex/mcadd`. `codegen` uses the name of the MATLAB function as the root name for the generated files



and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 19-66.

## Generating MEX Functions at the Command Line Using `codegen`

You generate a MEX function at the command line using the `codegen` function.

The basic command is:

```
codegen fcn
```

By default, `codegen` generates a MEX function in the current folder as described in “Generate MEX Functions at the Command Line”.

You can modify this default behavior by specifying one or more compiler options with `codegen`, separated by spaces on the command line. For more information, see `codegen`.

### See Also

- “Primary Function Input Specification”
- “MEX Function Generation at the Command Line”
- “Generate Code for Multiple Entry-Point Functions” on page 19-70
- “Generate Code for Global Data” on page 19-75

## Fix Errors Detected at Code Generation Time

When the code generation software detects errors or warnings, it automatically generates an error report. The error report describes the issues and provides links to the MATLAB code with errors.

To fix the errors, modify your MATLAB code to use only those MATLAB features that are supported for code generation. For more information, see “MATLAB Algorithm Design Basics”. Choose a debugging strategy for detecting and correcting code generation errors in your MATLAB code. For more information, see “Debugging Strategies”.

When code generation is complete, the software generates a MEX function that you can use to test your implementation in MATLAB.

If your MATLAB code calls functions on the MATLAB path, unless the code generation software determines that these functions should be extrinsic or you declare them to be extrinsic, it attempts to compile these functions. See “Resolution of Function Calls for Code Generation”. To get detailed diagnostics, add the `%#codegen` directive to each external function that you want `codegen` to compile.

### See Also

- “Code Generation Reports”
- “Why Test MEX Functions in MATLAB?”
- “When to Generate Code from MATLAB Algorithms”
- “Debugging Strategies”
- “Declaring MATLAB Functions as Extrinsic Functions”

## Design Considerations When Writing MATLAB Code for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

You can choose whether the generated code uses static or dynamic memory allocation.

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get best speed, but with higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- Speed

Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

To improve the speed of the generated code:

- Choose a suitable C or C++ compiler. The default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms is not a good compiler for performance.

- Consider disabling run-time checks.

By default, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower MEX function execution. Disabling run-time checks usually results in streamlined generated code and faster MEX function execution. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

### See Also

- “MATLAB Algorithm Design Basics”
- “Data Definition”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data”
- “Control Dynamic Memory Allocation” on page 19-95
- “Control Run-Time Checks”

## Running MEX Functions

When you call a MEX function, pass it the same inputs you use for the original MATLAB algorithm. Do not pass `coder.Constant` or any of the `coder.Type` classes to a MEX function; these classes are for use with the `codegen` function.

To run a MEX function generated by MATLAB Coder, you must have licenses for all the toolboxes that the MEX function requires. For example, if you generate a MEX function from a MATLAB algorithm that uses a Computer Vision System Toolbox function or System object, to run the MEX function, you must have a Computer Vision System Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

## Debugging MEX Functions

You cannot use the `disp` and `save` functions during debugging to inspect the contents of your MEX function variables. Because these functions are not supported for code generation, you must declare them as extrinsic functions. For extrinsic functions, when running the MEX function, MATLAB Coder calls out to MATLAB to run `disp` and `save`, so they save and display the data found in the base workspace, not the MEX-function workspace.

## Debugging Strategies

Before you perform code verification, choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other's functions. The following table describes two general strategies, each of which has advantages and disadvantages.

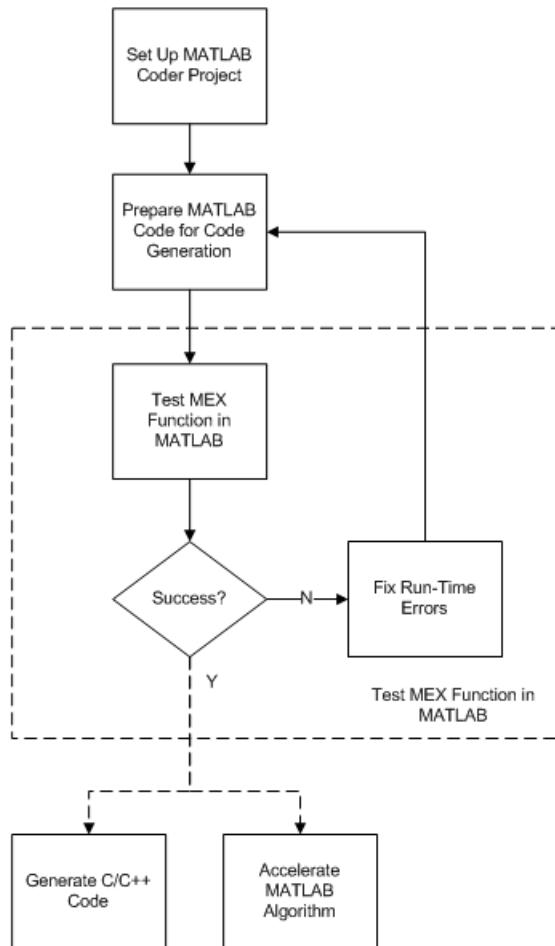
Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	<ol style="list-style-type: none"> <li>1 Verify that your lowest-level (leaf) functions are compliant.</li> <li>2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function.</li> </ol>	<ul style="list-style-type: none"> <li>• Efficient</li> <li>• Unlikely to cause errors</li> <li>• Easy to isolate code generation syntax violations</li> </ul>	Requires application tests that work from the bottom up
Top-down verification	<ol style="list-style-type: none"> <li>1 Declare functions called by the top-level function to be extrinsic so that MATLAB Coder does not compile them. See “Declaring MATLAB Functions as Extrinsic Functions”.</li> <li>2 Verify that your top-level function is compliant.</li> <li>3 Work your way down the function hierarchy incrementally by removing extrinsic declarations one by one to compile and verify each function, ending with the leaf functions.</li> </ol>	You retain your top-level tests	<p>Introduces extraneous code that you must remove after code verification, including:</p> <ul style="list-style-type: none"> <li>• Extrinsic declarations</li> <li>• Additional assignment statements as required to convert opaque values returned by extrinsic functions to nonopaque values (see “Working with mxArray”).</li> </ul>

# Testing MEX Functions in MATLAB

---

- “Workflow for Testing MEX Functions in MATLAB” on page 18-2
- “Why Test MEX Functions in MATLAB?” on page 18-4
- “Running MEX Functions” on page 18-5
- “Verify MEX Functions in a Project” on page 18-6
- “Verify MEX Functions at the Command Line” on page 18-8
- “Debug Run-Time Errors” on page 18-9

## Workflow for Testing MEX Functions in MATLAB



### See Also

- “MATLAB Coder Project Set Up Workflow”
- “Workflow for Preparing MATLAB Code for Code Generation”



- “Why Test MEX Functions in MATLAB?” on page 18-4
- “Debug Run-Time Errors” on page 18-9
- “C/C++ Code Generation”
- “Accelerate MATLAB Algorithms”

## Why Test MEX Functions in MATLAB?

Before generating C/C++ code for your MATLAB code, it is a best practice to test the MEX function to verify that it provides the same functionality as the original MATLAB code. To do this testing, run the MEX function using the same inputs as you used to run the original MATLAB code and compare the results. For more information about how to test a MEX function in a project, see “Verify MEX Functions in a Project” on page 18-6. For more information on how to test a MEX function at the command line, see “Verify MEX Functions at the Command Line” on page 18-8.

In addition, running the MEX function in MATLAB before generating code enables you to detect and fix run-time errors that are much harder to diagnose in the generated code. If you encounter run-time errors in your MATLAB functions, fix them before generating code. For more information, see “Debug Run-Time Errors” on page 18-9.

When you run your MEX function in MATLAB, by default, the following run-time checks execute :

- Memory integrity checks. These checks perform array bounds checking, dimension checking, and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution and provides a diagnostic message.
- Responsiveness checks in code generated for MATLAB functions. These checks enable periodic checks for **Ctrl+C** breaks in code generated for MATLAB functions, allowing you to terminate execution with **Ctrl+C**.

For more information, see “Control Run-Time Checks”.

## Running MEX Functions

When you call a MEX function, pass it the same inputs you use for the original MATLAB algorithm. Do not pass `coder.Constant` or any of the `coder.Type` classes to a MEX function; these classes are for use with the `codegen` function.

To run a MEX function generated by MATLAB Coder, you must have licenses for all the toolboxes that the MEX function requires. For example, if you generate a MEX function from a MATLAB algorithm that uses a Computer Vision System Toolbox function or System object, to run the MEX function, you must have a Computer Vision System Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

## Debugging MEX Functions

You cannot use the `disp` and `save` functions during debugging to inspect the contents of your MEX function variables. Because these functions are not supported for code generation, you must declare them as extrinsic functions. For extrinsic functions, when running the MEX function, MATLAB Coder calls out to MATLAB to run `disp` and `save`, so they save and display the data found in the base workspace, not the MEX-function workspace.

## Verify MEX Functions in a Project

### In this section...

“Using Test Files That Call Only MATLAB Functions” on page 18-6


“Using Test Files That Call MEX Functions” on page 18-7

### Using Test Files That Call Only MATLAB Functions

If you have a test file that calls only your original entry-point MATLAB function, use the following procedure. A test file can be either a MATLAB function or a script. To use this procedure, you should verify that it calls at least one entry-point function. The generated MEX function must be in the same folder as the entry-point functions.

Selecting the **Redirect entry-point calls to MEX function** option directs MATLAB Coder software to replace calls to the MATLAB function with calls to the generated MEX function. This capability allows you to compare the behavior of the MEX function with that of the original function.

If your test file calls the generated MEX function, do not follow this procedure. Instead, follow the procedure in “Using Test Files That Call MEX Functions” on page 18-7.

- 1 On the project **Build** tab **Verification** panel, click the  button to add a test file. Alternatively, if you have already added test files to the project, select one from the list.
- 2 Run the test file calling the original MATLAB algorithm.
  - a Clear **Redirect entry-point calls to MEX function**.
  - b Click the **Run** button.

The test file runs and calls your original MATLAB algorithm.

- 3 Verify that the test results are as expected.
- 4 Run the test file calling the MEX function instead of the original MATLAB algorithm.
  - a Select **Redirect entry-point calls to MEX function**.
  - b Click the **Run** button.


The project builds the MEX function. The test file runs and automatically replaces calls to your original MATLAB algorithm with calls to the generated MEX function.

- 5 Compare the results of the two runs to verify that the MEX function provides the same functionality as the original MATLAB algorithm.

## Using Test Files That Call MEX Functions

If you have a test file that calls the generated MEX function, use the following procedure. If your test file calls both the original MATLAB function and the generated MEX function, you can also use this procedure.

A test file can be either a MATLAB function or a script. To use this procedure, you should verify that it calls at least one MEX function. The MEX function must be in the same folder as the entry-point functions.

- 1 On the project **Build** tab **Verification** panel, click the  button to add a test file. Alternatively, if you have already added test files to the project, select one from the list.
- 2 Run the test file.
  - a Clear **Redirect entry-point calls to MEX function**.

Because the test file already calls the MEX function, you do not want MATLAB Coder to redirect entry-point function calls.

- b Click the **Run** button.

The project builds the MEX function. The test file runs and calls the generated MEX function. If applicable, it also calls the original MATLAB algorithm.

- 3 Use the results of this run to verify that the MEX function provides the same functionality as the original MATLAB algorithm.

## Verify MEX Functions at the Command Line

If you have a test file that calls your original MATLAB function, use `coder.runTest` to verify the MEX function at the command line. `coder.runTest` runs the test file replacing calls to the original MATLAB function with calls to the generated MEX function. If errors occur during the run with `coder.runTest`, call stack information is available for debugging purposes. For more information, see the `coder.runTest` function reference information and “Verifying the MEX Function” in the MATLAB Coder “C Code Generation at the Command Line” tutorial.

# Debug Run-Time Errors

**In this section...**

“Viewing Errors in the Run-Time Stack” on page 18-9

“Handling Run-Time Errors” on page 18-10

If you encounter run-time errors in your MATLAB functions, the run-time stack appears automatically in the MATLAB command window. Use the error message and stack information to learn more about the source of the error and then either fix the issue or add error-handling code. For more information, see “Viewing Errors in the Run-Time Stack” on page 18-9 “Handling Run-Time Errors” on page 18-10.

## Viewing Errors in the Run-Time Stack

### About the Run-Time Stack

The run-time stack is enabled by default for MEX code generation from MATLAB. Use the error message and the following stack information to learn more about the source of the error:

- The name of the function that generated the error
- The line number of the attempted operation
- The sequence of function calls that led up to the execution of the function and the line at which each of these function calls occurred

### Example Run-Time Stack Trace

This example shows the run-time stack trace for MEX function `mlstack_mex`:

```
mlstack_mex(-1)
```

```
Index exceeds matrix dimensions. Index  
value -1 exceeds valid range [1-4] of  
array x.
```

```
Error in mlstack>mayfail (line 31)  
y = x(u);
```

```
Error in mlstack>subfcn1 (line 5)  
switch (mayfail(u))
```

```
Error in mlstack (line 2)
y = subfcn1(u);
```

The stack trace provides the following information:

- The type of error.

```
??? Index exceeds matrix dimensions.
Index value -1 exceeds valid range [1-4] of array x.
```

- Where the error occurred.

```
Error in ==>mlstack>mayfail at 31
y = x(u);
```

- The function call sequence prior to the failure.

```
Error in ==> mlstack>subfcn1 at 5
switch (mayfail(u))
```

```
Error in ==> mlstack at 2
y = subfcn1(u);
```

### When to Use the Run-Time Stack

The run-time stack is useful during debugging to help you find the source of run-time errors. However, when the stack is enabled, the generated code contains instructions for maintaining the run-time stack, which might slow the run time. Consider disabling the run-time stack for faster run time.

### How to Disable the Run-Time Stack

You can disable the run-time stack by disabling the memory integrity checks as described in “How to Disable Run-Time Checks”.

---

**Caution** Before disabling the memory integrity checks, you should verify that all array bounds and dimension checking is unnecessary.

---

## Handling Run-Time Errors

The code generation software propagates error ID's. If you throw an error or warning in your MATLAB code, use the `try-catch` statement in your test bench code to examine



the error information and attempt to recover, or clean up and abort. For example, for the function in “Example Run-Time Stack Trace” on page 18-9, create a test script containing:

```
try
    mlstack_mex(u)
catch
    % Add your error handling code here
end
```

For more information, see “The try/catch Statement”.



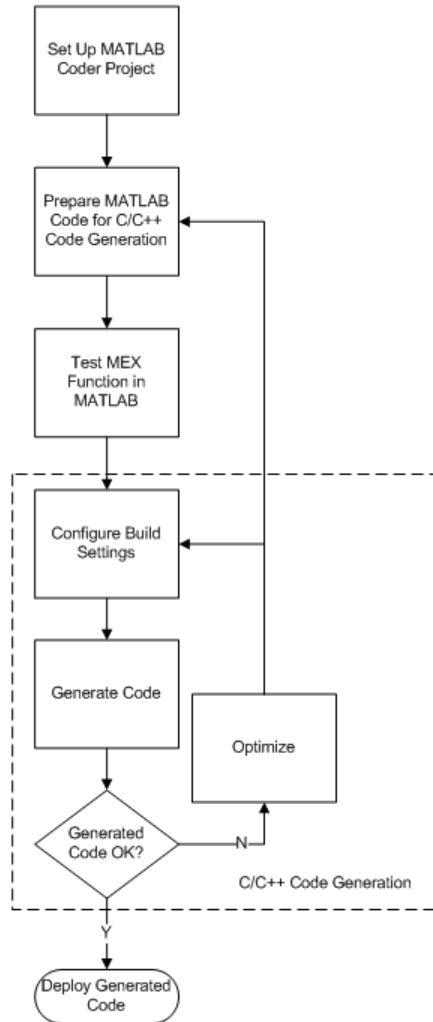
# Generating C/C++ Code from MATLAB Code

---

- “Code Generation Workflow” on page 19-3
- “C/C++ Code Generation” on page 19-5
- “Generating C/C++ Static Libraries from MATLAB Code” on page 19-6
- “Generating C/C++ Dynamically Linked Libraries from MATLAB Code” on page 19-10
- “Generating Standalone C/C++ Executables from MATLAB Code” on page 19-13
- “Build Setting Configuration” on page 19-19
- “Standard Math Libraries” on page 19-32
- “Change the Standard Math Library” on page 19-33
- “Share Build Configuration Settings” on page 19-34
- “Convert MATLAB Coder Project to MATLAB Script” on page 19-37
- “Primary Function Input Specification” on page 19-39
- “Control Constant Inputs in MEX Function Signatures” on page 19-49
- “Define Input Properties Programmatically in the MATLAB File” on page 19-53
- “Speed Up Compilation” on page 19-63
- “Paths and File Infrastructure Setup” on page 19-65
- “Generate Code for Multiple Entry-Point Functions” on page 19-70
- “Generate Code for Global Data” on page 19-75
- “Generation of Traceable Code” on page 19-84
- “Generate Code for Enumerated Types” on page 19-93
- “Generate Code for Variable-Size Data” on page 19-94
- “Code Generation for MATLAB Classes” on page 19-113
- “How MATLAB Coder Partitions Generated Code” on page 19-114

- “Requirements for Signed Integer Representation” on page 19-126
- “Customize the Post-Code-Generation Build Process” on page 19-127
- “Code Generation Reports” on page 19-167
- “Troubleshooting” on page 19-186
- “Package Code For Other Development Environments” on page 19-187

## Code Generation Workflow



### See Also

- “MATLAB Coder Project Set Up Workflow”

- “Workflow for Preparing MATLAB Code for Code Generation”
- “Workflow for Testing MEX Functions in MATLAB”
- “Build Setting Configuration” on page 19-19
- “C/C++ Code Generation” on page 19-5

## C/C++ Code Generation

Using MATLAB Coder, you can generate standalone C/C++ static and dynamic libraries and C/C++ executables. If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes. By default, if MATLAB Coder does not detect errors, it generates a platform-specific MEX function in the current folder.

To learn how to generate...	See...
C/C++ static libraries from your MATLAB code	“Generating C/C++ Static Libraries from MATLAB Code” on page 19-6
C/C++ dynamic libraries from your MATLAB code	“Generating C/C++ Dynamically Linked Libraries from MATLAB Code” on page 19-10
C/C++ executables from your MATLAB code	“Generating Standalone C/C++ Executables from MATLAB Code” on page 19-13
MEX functions from your MATLAB code	“Generate MEX Functions Using the MATLAB Coder Project Interface”

If errors occur, MATLAB Coder does not generate code, but produces an error report and provides a link to this report. For more information, see “Code Generation Reports” on page 19-167.

### Specify Custom Files to Build

In addition to your MATLAB file, you can specify the following types of custom *files* to include in the build for standalone C/C++ code generation.

File Extension	Description
.c	Custom C file
.cpp	Custom C++ file
.h	Custom header file
.o , .obj	Custom object file
.a , .lib, .so	Library
.tmf	Template makefile for custom MATLAB Coder builds

## Generating C/C++ Static Libraries from MATLAB Code

### In this section...

“Generate a C Static Library Using the Project Interface” on page 19-6

“Generate a C Static Library at the Command Line” on page 19-9

### Generate a C Static Library Using the Project Interface

This example shows how to generate a C static library from MATLAB code using a MATLAB Coder project.

In this example, you create a MATLAB function that adds two numbers. You then create a MATLAB Coder project. Use the project user interface to generate a C static library for the MATLAB code.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 In the same folder, set up a MATLAB Coder project.

- a At the MATLAB command line, enter:

```
coder -new mcadd.prj
```

By default, the project opens in the MATLAB workspace on the right side.

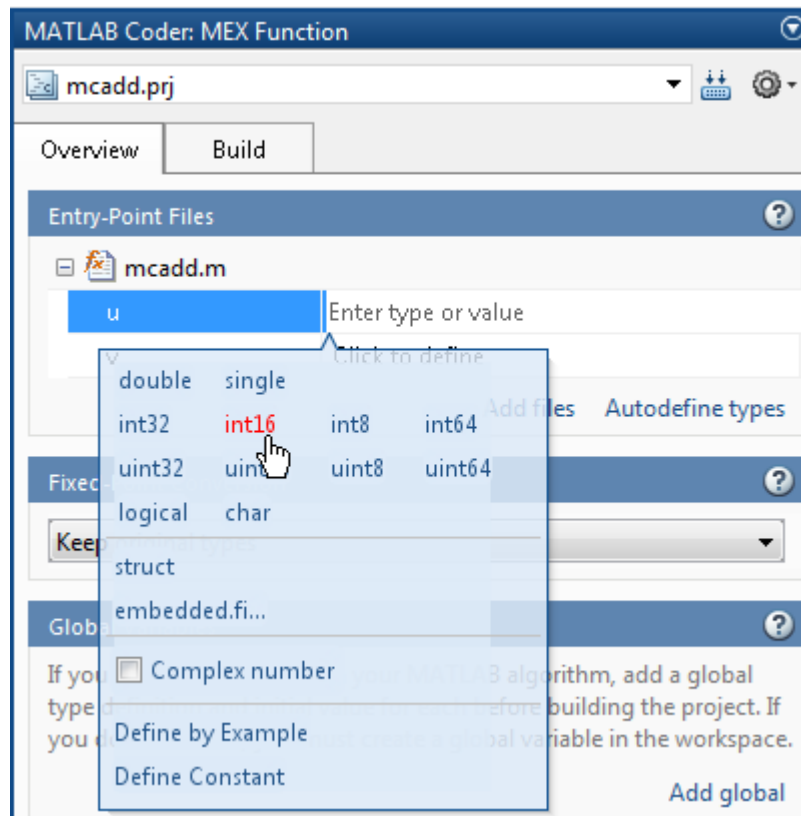
- b On the project **Overview** tab, click the **Add files** link. Browse to the file `mcadd.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. Both inputs are undefined.

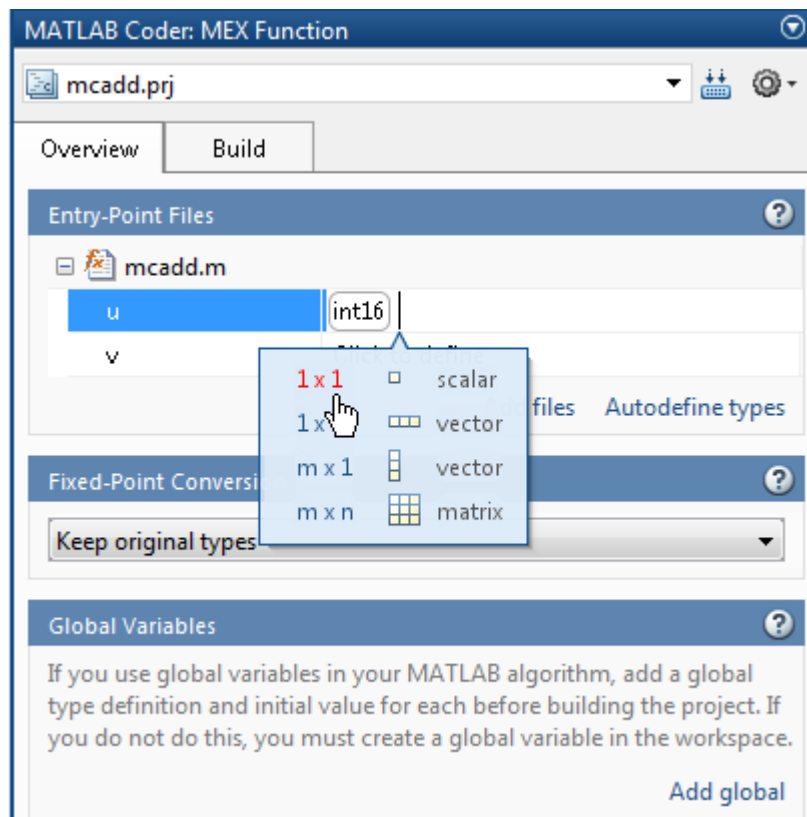
- 3 Define the type of input `u`.

- a On the **Overview** tab, click the field to the right of the input parameter `u` and, from the list of input options, select `int16`.





- b** From the list of size options, select  $1 \times 1$  to specify that the input is a scalar.



- 4 Repeat the previous step for input `v`.
- 5 In the MATLAB Coder project, click the **Build** tab.
- 6 On this tab, set the **Output type** to **C/C++ Static library**.

The default output file name is `mcadd`.

- 7 Click **Build** to generate a library using the default project settings.

MATLAB Coder builds the project and generates a C static library and supporting files in the default folder, `codegen/lib/mcadd`. It generates the minimal set of

`#include` statements for header files required by the selected code replacement library.

## Generate a C Static Library at the Command Line

This example shows how to generate a C static library from MATLAB code at the command line using the `codegen` function.

- 1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

- 2 Using the `config:lib` option, generate C library files. Using the `-args` option, specify that the first input is a 1-by-4 vector of unsigned 16-bit integers and that the second input is a double-precision scalar.

```
codegen -config:lib mcadd -args {zeros(1,4,'uint16'),0}
```

MATLAB Coder generates a C static library with the default name, `mcadd`, and supporting files in the default folder, `codegen/lib/mcadd`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library.

## Generating C/C++ Dynamically Linked Libraries from MATLAB Code

### In this section...

“Dynamic Libraries Generated by MATLAB Coder” on page 19-10

“Generate a C Dynamically Linked Library (DLL) Using the Project Interface” on page 19-10

“Generate a C Dynamic Library at the Command Line” on page 19-12

### Dynamic Libraries Generated by MATLAB Coder

By default, when MATLAB Coder generates a dynamic library (DLL):

- The DLL is suitable for the platform that you are working on.
- The DLL uses the release version of the C run-time library.
- The DLL linkage conforms to the target language, by default, C. If you set the target language to C++, the linkage conforms to C++.
- When the target language is C, the generated header files explicitly declare the exported functions to be `extern "C"` to simplify integration of the DLL into C++ applications.

If you generate a DLL that uses dynamically allocated variable-size data, MATLAB Coder automatically provides exported utility functions to interact with this data in the generated code. For more information, see “Utility Functions for Creating `emxArray` Data Structures”.

### Generate a C Dynamically Linked Library (DLL) Using the Project Interface

This example shows how to generate a C DLL from MATLAB code using a MATLAB Coder project.

In this example, you create a MATLAB function that generates a random scalar value. You then create a MATLAB Coder project. Use the project user interface to generate a C dynamic library for the MATLAB code.

- 1 Write two MATLAB functions, `ep1` takes one input, a single scalar, and `ep2` takes two inputs, both double scalars. In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen  
y = u;
```

In the same folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen  
y = u + v;
```

- 2 In the same folder as the `ep1` and `ep2` files, set up a MATLAB Coder project. At the MATLAB command line, enter:

```
coder -new ep.prj
```

By default, the project opens in the MATLAB workspace on the right side.

- 3 On the project **Overview** tab, click the **Add files** link and browse to the file `ep1.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. MATLAB Coder indicates that input `u` is undefined.

- 4 Define the type of input `u`.

- a On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select **single**.

- b From the list of size options, select `1 x 1` to specify that `u` is a scalar.

- 5 On the project **Overview** tab, click the **Add files** link and browse to the file `ep2.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. MATLAB Coder indicates that inputs `u` and `v` are undefined.

- 6 Define the type of input `u`.

- a On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select **double**.

- b From the list of size options, select `1 x 1` to specify that `u` is a scalar.

- 7 Repeat the previous step for input `v`.

- 8 In the MATLAB Coder project, click the **Build** tab.

- 9 On the **Build** tab, set the **Output type** to **C/C++ Dynamic Library**.

- 10 On the **Build** tab, click the **Build** button to generate a library using these project settings.

On Microsoft® Windows systems, MATLAB Coder generates a C dynamic library, `ep1.dll`, and supporting files, in the default folder, `codegen/dll/ep1`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library. On Linux and Macintosh systems, it generates a shared object (`.so`) file. The DLL linkage conforms to the target language, in this example, C. If you set the target language to C++, the linkage conforms to C++.

## Generate a C Dynamic Library at the Command Line

This example shows how to generate a C dynamic library from MATLAB code at the command line using the `codegen` function.

- 1 Write two MATLAB functions, `ep1` takes one input, a single scalar, and `ep2` takes two inputs, both double scalars. In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

In the same folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

- 2 Generate the C dynamic library.

```
codegen -config:dll ep1 -args single(0) ep2 -args {0,0}
```

On Microsoft Windows systems, `codegen` generates a C dynamic library, `ep1.dll`, and supporting files, in the default folder, `codegen/dll/ep1`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library. On Linux and Macintosh systems, it generates a shared object (`.so`) file. The DLL linkage conforms to the target language, in this example, C. If you set the target language to C++, the linkage conforms to C++.

---

**Note:** The default target language is C. To change the target language to C++, see “Specify a Language for Code Generation” on page 19-21.

---

# Generating Standalone C/C++ Executables from MATLAB Code

## In this section...

“Generate a C Executable Using the Project Interface” on page 19-13

“Generate a C Executable at the Command Line” on page 19-15

“Specifying main Functions for C/C++ Executables” on page 19-16

“Specify main Functions” on page 19-17

## Generate a C Executable Using the Project Interface

In this example, you create a MATLAB function that generates a random scalar value and a main C function that calls this MATLAB function. You then create a MATLAB Coder project. Use the project user interface to specify types for the function input parameters, specify the main function, and generate a C executable for the MATLAB code.

- 1 Write a MATLAB function, `coderand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1):

```
function r = coderand() %#codegen
r = rand();
```

- 2 Write a main C function, `c:\myfiles\main.c`, that calls `coderand`. For example:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_initialize.h"
#include "coderand_terminate.h"

int main()
{
    coderand_initialize();

    printf("coderand=%g\n", coderand());

    coderand_terminate();
}
```

```
    return 0;  
}
```

---

**Note:** In this example, because the default file partitioning method is to generate one file for each MATLAB file, you include `coderand_initialize.h` and `coderand_terminate.h`. If your file partitioning method is set to generate one file for all functions, do **not** include `coderand_initialize.h` and `coderand_terminate.h`.

---

**3** In the same folder as the `coderand` file, set up a MATLAB Coder project.

**a** At the MATLAB command line, enter:

```
coder -new coderand.prj
```

By default, the project opens in the MATLAB workspace on the right side.

**b** On the project **Overview** tab, click the **Add files** link and browse to the file `coderand.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab. MATLAB Coder indicates that the `coderand` function has no inputs.

**4** In the MATLAB Coder project, click the **Build** tab.

**a** Set the **Output type** to **C/C++ Executable**.

**b** Set the output file name to `coderand_exe`.

**5** On the project **Build** tab, click the **More settings** link.

**6** On the Project Settings dialog box **Custom Code** tab, under **Additional files and directories to be built**, set:

**a** **Source files** to `main.c`, which is the name of the C/C++ source file that contains the `main` function.

**b** **Include directories** to the location of `main.c`: `c:\myfiles`.

**c** Close the dialog box.

---

**Note:** When you are building an executable, you must specify the main function. For more information, see “Specifying main Functions for C/C++ Executables” on page 19-16.

---



- 7 On the **Build** tab, click the **Build** button to generate a library using the default project settings.

MATLAB Coder compiles and links the main function with the C code that it generates for the project and, in the current folder, generates an executable, `coderand_exe`. It generates supporting files in the default folder, `codegen/exe/coderand`. MATLAB Coder generates the minimal set of `#include` statements for header files required by the selected code replacement library.

### See Also

- “MATLAB Coder Project Set Up Workflow”
- “Workflow for Preparing MATLAB Code for Code Generation”
- “Workflow for Testing MEX Functions in MATLAB”
- “Build Setting Configuration” on page 19-19
- “C/C++ Code Generation” on page 19-5
- “Optimization Strategies”

## Generate a C Executable at the Command Line

In this example, you create a MATLAB function that generates a random scalar value and a main C function that calls this MATLAB function. You then specify types for the function input parameters, specify the main function, and generate a C executable for the MATLAB code.

- 1 Write a MATLAB function, `coderand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1):

```
function r = coderand() %#codegen
r = rand();
```

- 2 Write a main C function, `c:\myfiles\main.c`, that calls `coderand`. For example:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_initialize.h"
#include "coderand_terminate.h"
```

```
int main()
{
    coderand_initialize();

    printf("coderand=%g\n", coderand());

    coderand_terminate();

    return 0;
}
```

---

**Note:** In this example, because the default file partitioning method is to generate one file for each MATLAB file, you include “`coderand_initialize.h`” and “`coderand_terminate.h`”. If your file partitioning method is set to generate one file for all functions, do **not** include “`coderand_initialize.h`” and “`coderand_terminate.h`”.

---

- 3 Configure your code generation parameters to include the main C function and then generate the C executable:

```
cfg = coder.config('exe');
cfg.CustomSource = 'main.c';
cfg.CustomInclude = 'c:\myfiles';
codegen -config cfg coderand
```

`codegen` generates a C executable, `coderand.exe`, in the current folder. It generates supporting files in the default folder, `codegen/exe/coderand`. `codegen` generates the minimal set of `#include` statements for header files required by the selected code replacement library.

## Specifying main Functions for C/C++ Executables

When you generate an executable, you must provide a `main` function. If you are generating a C executable, provide a C file, `main.c`. If you are generating a C++ executable, provide a C++ file, `main.cpp`. Verify that the folder containing the main function has only one main file. Otherwise, `main.c` takes precedence over `main.cpp`, which causes an error when generating C++ code. You can specify the main file from the project settings dialog box, the command line, or the Code Generation dialog box.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder automatically generates an initialize function and a terminate function.

- If your file partitioning method is set to generate one file for each MATLAB file, you must include the initialize and terminate header functions in `main.c`. Otherwise, do not include them in `main.c`.
- You must call these functions along with the C/C++ function. For more information, see “Calling Initialize and Terminate Functions” on page 22-7.

## Specify main Functions

### Specifying main Functions in the Project Settings Dialog Box

- 1 On the project **Build** tab, click the **More settings** link to open the **Project Settings** dialog box.
- 2 On the **Custom Code** tab, set:
  - a **Additional source files** to the name of the C/C++ source file that contains the `main` function. For example, `main.c`. For more information, see “Specifying main Functions for C/C++ Executables” on page 19-16.
  - b **Additional include directories** to the location of `main.c`. For example, `c:\myfiles`.

### Specifying main Functions at the Command Line

Set the `CustomSource` and `CustomInclude` properties of the code generation configuration object (see “Working with Configuration Objects” on page 19-27). The `CustomInclude` property indicates the location of C/C++ files specified by `CustomSource`.

- 1 Create a configuration object for an executable:

```
cfg = coder.config('exe');
```
- 2 Set the `CustomSource` property to the name of the C/C++ source file that contains the `main` function. (For more information, see “Specifying main Functions for C/C++ Executables” on page 19-16.) For example:

```
cfg.CustomSource = 'main.c';
```
- 3 Set the `CustomInclude` property to the location of `main.c`. For example:

```
cfg.CustomInclude = 'c:\myfiles';
```
- 4 Generate the C/C++ executable using the command line options. For example, if `myFunction` takes one input parameter of type `double`:

```
codegen -config cfg myMFunction -args {0}
```

MATLAB Coder compiles and links the main function with the C/C++ code that it generates from `myMFunction.m`.

# Build Setting Configuration

## In this section...

- “Specify Output Type” on page 19-19
- “Specify a Language for Code Generation” on page 19-21
- “Specify Data Type Used in Generated Code” on page 19-22
- “Specify Output File Name” on page 19-23
- “Specify Output File Locations” on page 19-24
- “Parameter Specification Methods” on page 19-25
- “Specify Build Configuration Parameters” on page 19-25

## Specify Output Type

### Output Types

MATLAB Coder can generate code for the following output types:

- MEX function
- Instrumented MEX function
- Standalone C/C++ code and compile it to a static library
- Standalone C/C++ code and compile it to a dynamically-linked library
- Standalone C/C++ code and compile it to an executable

---

**Note:** When you generate an executable, you must provide a C/C++ file that contains the `main` function, as described in “Specifying main Functions for C/C++ Executables” on page 19-16.

---

### Location of Generated Files

By default, MATLAB Coder generates files in output folders based on your output type. For more information, see “Generated Files and Locations” on page 19-120.

---

**Note:** Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

---

### Specifying the Output Type Using the MATLAB Coder Project Interface

On the MATLAB Coder project **Build** tab, set **Output type** to one of the available output types:

- MEX Function (default)
- Instrumented MEX Function
- C/C++ Static Library
- C/C++ Dynamic Library
- C/C++ Executable

MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you switch the output type between MEX Function or Instrumented MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

### Specifying the Output Type at the Command Line

Call `codegen` with the `-config` option. For example, suppose you have a primary function `foo` that takes no input parameters. The following table shows how to specify different output types when compiling `foo`. If a primary function has input parameters, you must specify these inputs. For more information, see “Primary Function Input Specification” on page 19-39.

---

**Note:** C is the default language for code generation with MATLAB Coder. To generate C++ code, see “Specify a Language for Code Generation” on page 19-21.

---

To Generate:	Use This Command:
MEX function using the default code generation options	<code>codegen foo</code>
MEX function specifying code generation options	<pre> cfg = coder.config('mex'); % Set configuration parameters, for example, % enable a code generation report cfg.GenerateReport=true; % Call codegen, passing the configuration % object </pre>

To Generate:	Use This Command:
	<code>codegen -config cfg foo</code>
Standalone C/C++ code and compile it to a library using the default code generation options	<code>codegen -config:lib foo</code>
Standalone C/C++ code and compile it to a library specifying code generation options	<pre>cfg = coder.config('lib'); % Set configuration parameters, for example, % enable a code generation report cfg.GenerateReport=true; % Call codegen, passing the configuration % object codegen -config cfg foo</pre>
Standalone C/C++ code and compile it to an executable using the default code generation options and specifying the <code>main.c</code> file at the command line	<pre>codegen -config:exe main.c foo</pre> <p><b>Note:</b> You must specify a <code>main</code> function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 19-16</p>
Standalone C/C++ code and compile it to an executable specifying code generation options	<pre>cfg = coder.config('exe'); % Set configuration parameters, for example, % specify main file cfg.CustomSource = 'main.c'; cfg.CustomInclude = 'c:\myfiles'; codegen -config cfg foo</pre> <p><b>Note:</b> You must specify a <code>main</code> function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 19-16</p>

## Specify a Language for Code Generation

- “Specifying a Language for Code Generation in the Project Settings Dialog Box” on page 19-22
- “Specifying a Language for Code Generation at the Command Line” on page 19-22

MATLAB Coder can generate C or C++ libraries and executables. C is the default language. You can specify a language explicitly from the project settings dialog box or at the command line.

### Specifying a Language for Code Generation in the Project Settings Dialog Box

- 1 Select a suitable compiler for your target language.
- 2 On the MATLAB Coder project **Build** tab, click the **More settings** link to open the **Project Settings** dialog box.
- 3 On the **All Settings** tab, in the **Advanced** group, set **Language** to **C** or **C++**.

---

**Note:** If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

---

### Specifying a Language for Code Generation at the Command Line

- 1 Select a suitable compiler for your target language.
- 2 Create a configuration object for code generation. For example, for a library:

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

- 3 Set the `TargetLang` property to 'C' or 'C++'. For example:

```
cfg.TargetLang = 'C++';
```

---

**Note:** If you specify C++, MATLAB Coder wraps the C code into .cpp files. You can then use a C++ compiler and interface with external C++ applications. MATLAB Coder does not generate C++ classes.

---

#### See Also

- “Working with Configuration Objects” on page 19-27
- “Setting Up the C or C++ Compiler”

### Specify Data Type Used in Generated Code

- “Specify Data Type in the Project Settings Dialog Box” on page 19-23
- “Specify Data Type at the Command Line” on page 19-23

MATLAB Coder can use built-in C data types or predefined types from `rtwtypes.h` in generated code. By default, the generated code uses built-in C types when declaring variables.



You can explicitly specify the data type used in generated code in the project settings dialog box or at the command line.

### Specify Data Type in the Project Settings Dialog Box

- 1 On the **Build** tab **Settings** pane, set the **Output type** to **C/C++ Static Library**, **C/C++ Dynamic Library**, or **C/C++ Executable** (depending on your requirements).
- 2 Click the **More settings** link to open the **Project Settings** dialog box.
- 3 To use built-in C types, on the **Code Appearance** tab, set **Data Type Replacement** to **Use built-in C data types in the generated code**. To use predefined types from `rtwtypes.h`, set **Data Type Replacement** to **Use MathWorks typedefs in the generated code**.

### Specify Data Type at the Command Line

- 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:
 

```
cfg = coder.config('lib');
```
- 2 To use built-in C types, set the `DataTypeReplacement` property to `'CBuiltIn'`.
 

```
cfg.DataTypeReplacement = 'CBuiltIn';
```

To use predefined types from `rtwtypes.h`, set the `DataTypeReplacement` property to `'CoderTypedefs'`.

## Specify Output File Name

### Specify Output File Name in a Project

On the project **Build** tab, in the **Output file** field, enter the file name. The file name can include an existing path.

---

**Note:** Do not put spaces in the file name.

---

By default, if the name of the first entry-point MATLAB file is `fcn1`, the output file name is:

- `fcn1` for C/C++ libraries and executables.

- `fcn1_mex` for MEX functions.

By default, MATLAB Coder generates files in the folder `project_folder/codegen/target/fcn1`:

- `project_folder` is your current project folder
- `target` is:
  - `mex` for MEX functions
  - `lib` for static C/C++ libraries
  - `dll` for dynamic C/C++ libraries
  - `exe` for C/C++ executables

### Command Line Alternative

Use the `codegen` function `-o` option.

## Specify Output File Locations

### Specifying Output File Location in a Project

The output file location should not contain:

- Spaces, as this can lead to code generation failures in certain operating system configurations.
- Non 7-bit ASCII characters, such as Japanese characters.

- 1 On the project **Build** tab, click **More settings**.
- 2 In the Project Settings dialog box, click the **Paths** tab.

The default setting for the **Build Folder** field is A subfolder of the project folder. By default, MATLAB Coder generates files in the folder `project_folder/codegen/target/fcn1`:

- `fcn1` is the name of the first entry-point file
- `target` is:
  - `mex` for MEX functions
  - `lib` for static C/C++ libraries
  - `dll` for dynamically-linked C/C++ libraries

- `exe` for C/C++ executables

**3** To change the output location, you can either:

- Set **Build Folder** to A subfolder of the current MATLAB working folder

MATLAB Coder generates files in the *MATLAB\_working\_folder/codegen/target/fcn1* folder

- Set **Build Folder** to Specified folder. In the **Build folder name** field, provide the path to the folder.

### Command Line Alternative

Use the `codegen` function `-d` option.

## Parameter Specification Methods

If you are using...	Use...	Details
A MATLAB Coder project	The Project Settings dialog box	“Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-26
<code>codegen</code> at the command line and want to specify a small number of parameters	Configuration objects	“Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-26
<code>codegen</code> in build scripts		
<code>codegen</code> at the command line and want to specify a large number of parameters	Configuration object dialog boxes	“Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 19-30

## Specify Build Configuration Parameters

- “Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-26
- “Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-26

- “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 19-30

You can specify build configuration parameters from the MATLAB Coder project settings dialog box, the command line, or configuration object dialog boxes.

### Specifying Build Configuration Parameters in the Project Settings Dialog Box

- 1 On the MATLAB Coder project **Build** tab, click **More settings**.

The Project Settings dialog box opens. This dialog box provides the set of configuration parameters applicable to the output type that you select.

---

**Note:** MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you switch the output type between MEX Function or Instrumented MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

---

- 2 Modify the parameters as required. For more information about parameters on a tab, click the **Help** button.

Changes to the parameter settings take place immediately.

- 3 After specifying the build parameters, you can generate code by clicking the **Build** button on the same tab.

### Specifying Build Configuration Parameters at the Command Line Using Configuration Objects

#### Types of Configuration Objects

The `codegen` function uses configuration objects to customize your environment for code generation. The following table lists the available configuration objects.

Configuration Object	Description
<code>coder.CodeConfig</code>	<p>If no Embedded Coder license is available or you disable use of the Embedded Coder license, specifies parameters for C/C++ library or executable generation.</p> <p>For more information, see the class reference information for <code>coder.CodeConfig</code>.</p>

Configuration Object	Description
<code>coder.EmbeddedCodeConfig</code>	<p>If an Embedded Coder license is available, specifies parameters for C/C++ library or executable generation.</p> <p>For more information, see the class reference information for <code>coder.EmbeddedCodeConfig</code>.</p>
<code>coder.HardwareImplementation</code>	<p>Specifies parameters of the target hardware implementation. If not specified, <code>codegen</code> generates code that is compatible with the MATLAB host computer.</p> <p>For more information, see the class reference information for <code>coder.HardwareImplementation</code>.</p>
<code>coder.MexCodeConfig</code>	<p>Specifies parameters for MEX code generation.</p> <p>For more information, see the class reference information for <code>coder.MexCodeConfig</code>.</p>

### Working with Configuration Objects

To use configuration objects to customize your environment for code generation:

- 1 In the MATLAB workspace, define configuration object variables, as described in “Creating Configuration Objects” on page 19-28.

For example, to generate a configuration object for C static library generation:

```
cfg = coder.config('lib');
% Returns a coder.CodeConfig object if no
% Embedded Coder license available.
% Otherwise, returns a coder.EmbeddedCodeConfig object.
```

- 2 Modify the parameters of the configuration object as required, using one of these methods:
  - Interactive commands, as described in “Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-26
  - Dialog boxes, as described in “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 19-30
- 3 Call the `codegen` function with the `-config` option. Specify the configuration object as its argument.

The `-config` option instructs `codegen` to generate code for the target, based on the configuration property values. In the following example, `codegen` generates a C static library from a MATLAB function, `foo`, based on the parameters of a code generation configuration object, `cfg`, defined in the first step:

```
codegen -config cfg foo
```

The `-config` option specifies the type of output that you want to build — in this case, a C static library. For more information, see `codegen`.

### Creating Configuration Objects

You can define a configuration object in the MATLAB workspace.

To Create...	Use a Command Such As...
MEX configuration object <code>coder.MexCodeConfig</code>	<pre>cfg = coder.config('mex');</pre>
Code generation configuration object for generating a standalone C/C++ library or executable <code>coder.CodeConfig</code>	<pre>% To generate a static library cfg = coder.config('lib'); % To generate a dynamic library cfg = coder.config('dll') % To generate an executable cfg = coder.config('exe');</pre> <p><b>Note:</b> If an Embedded Coder license is available, creates a <code>coder.EmbeddedCodeConfig</code> object.</p> <p>If you use concurrent licenses, to disable check out of an Embedded Coder license, use one of the following commands:</p> <pre>cfg = coder.config('lib', 'ecoder', false) cfg = coder.config('dll', 'ecoder', false) cfg = coder.config('exe', 'ecoder', false)</pre>
Code generation configuration object for generating a standalone C/C++ library or executable for an embedded target	<pre>% To generate a static library cfg = coder.config('lib'); % To generate a dynamic library cfg = coder.config('dll')</pre>

To Create...	Use a Command Such As...
<code>coder.EmbeddedCodeConfig</code>	<pre>% To generate an executable cfg = coder.config('exe');</pre> <p><b>Note:</b> Requires an Embedded Coder license; otherwise creates a <code>coder.CodeConfig</code> object.</p>
Hardware implementation configuration object <code>coder.HardwareImplementation</code>	<pre>hwcfg = coder.HardwareImplementation</pre>

Each configuration object comes with a set of parameters, initialized to default values. You can change these settings, as described in “Modifying Configuration Objects at the Command Line Using Dot Notation” on page 19-29.

### Modifying Configuration Objects at the Command Line Using Dot Notation

You can use dot notation to modify the value of one configuration object parameter at a time. Use this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties:

- To specify a main function during C/C++ code generation:

```
cfg = coder.config('exe');
cfg.CustomInclude = 'c:\myfiles';
cfg.CustomSource = 'main.c';
codegen -config cfg foo
```

- To automatically generate and launch code generation reports after generating a C/C++ static library:

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.LaunchReport = true;
codegen -config cfg foo
```

### Saving Configuration Objects

Configuration objects do not automatically persist between MATLAB sessions. Use one of the following methods to preserve your settings:

## Save a configuration object to a MAT-file and then load the MAT-file at your next session

For example, assume you create and customize a MEX configuration object `mexcfg` in the MATLAB workspace. To save the configuration object, at the MATLAB prompt, enter:

```
save mexcfg.mat mexcfg
```

The `save` command saves `mexcfg` to the file `mexcfg.mat` in the current folder.

To restore `mexcfg` in a new MATLAB session, at the MATLAB prompt, enter:

```
load mexcfg.mat
```

The `load` command loads the objects defined in `mexcfg.mat` to the MATLAB workspace.

## Write a script that creates the configuration object and sets its properties.

You can rerun the script whenever you need to use the configuration object again.

### Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes

- 1 Create a configuration object as described in “Creating Configuration Objects” on page 19-28.

For example, to create a `coder.MexCodeConfig` configuration object for MEX code generation:

```
mexcfg = coder.config('mex');
```

- 2 Open the property dialog box using one of these methods:

- In the MATLAB workspace, double-click the configuration object variable.
- At the MATLAB prompt, issue the `open` command, passing it the configuration object variable, as in this example:

```
open mexcfg
```

- 3 In the dialog box, modify configuration parameters as required, then click **Apply**.
- 4 Call the `codegen` function with the `-config` option. Specify the configuration object as its argument:



```
codegen -config mexcfg foo
```

The `-config` option specifies the type of output that you want to build. For more information, see `codegen`.

## Standard Math Libraries

By default, the MATLAB Coder software generates code that calls the C89/C90 (ANSI C) library for math operations. Depending on your language choice, you have the option of changing the standard math library that the code generation software uses. Available libraries include:

Library Name	Language Support	Standard
C89/C90 (ANSI)	C, C++	ANSI C89/C90 (default)
C99 (ISO)	C, C++	ISO/IEC 9899:1990
C++03 (ISO)	C++	ISO/IEC 14882:2003

## Change the Standard Math Library

By default, the MATLAB Coder software uses the ANSI C89/C90 C math library when generating C or C++ code. If your compiler supports newer language standards, you can specify a different supported library. To change the library:

- In a project, on the **Hardware** tab, set the **Standard Math Library** parameter.
- In a code configuration object, set the `TargetLangStandard` parameter.

### See Also

- “Standard Math Libraries” on page 19-32
- “Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-26
- “Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-26

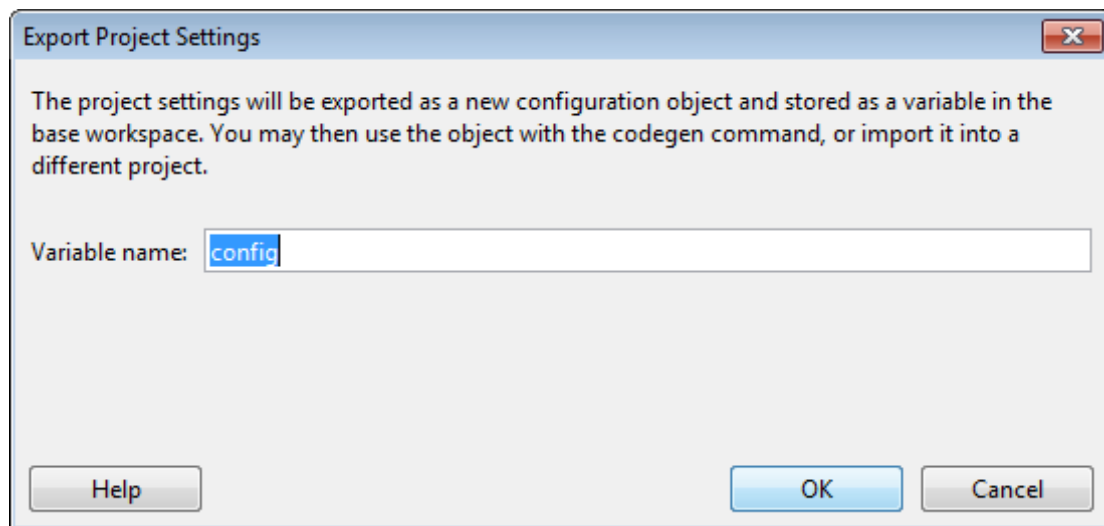
## Share Build Configuration Settings

To share build configuration settings between multiple projects or between the project and command-line workflow, use the project **Export settings** and **Import settings** options.

### Export Settings

To export the current project settings to a code generation configuration object stored in the base workspace:

- 1 In the top right corner of the project, click the **Actions** icon (⚙) and select **Export settings**.
- 2 In the **Export Project Settings** dialog box, specify a name for the configuration object.



MATLAB Coder saves the project settings information in a configuration object with the specified name in the base workspace.

Project Output Type	Configuration Object
MEX Function	<code>coder.MexCodeConfig</code>

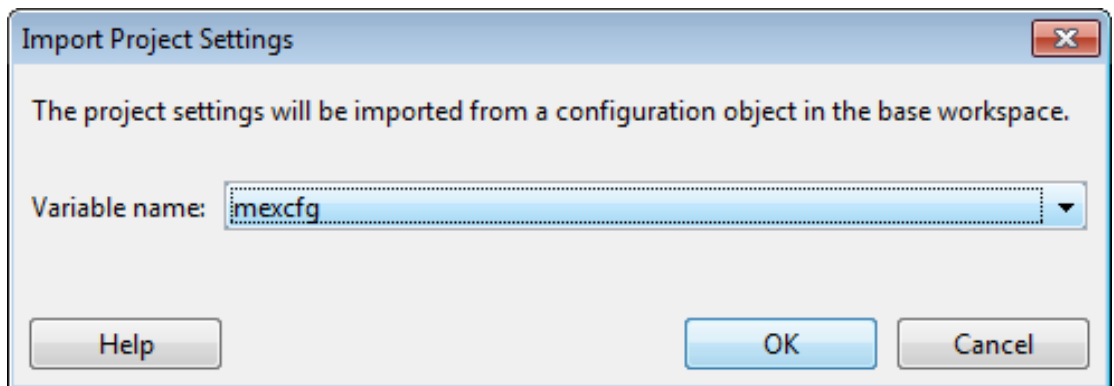
Project Output Type	Configuration Object
Instrumented MEX Function	
C/C++ Static Library	Without an Embedded Coder license: coder.CodeConfig
C/C++ Dynamic Library	With an Embedded Coder license: coder.EmbeddedCodeConfig
C/C++ Executable	

You can then either import these settings into another project or use it with the codegen function -config option to generate code at the command line.

## Import Settings

To import the settings saved in a code generation configuration object stored in the base workspace:

- 1 In the top right corner of the project, click the **Actions** icon (⚙) and select **Import settings**.
- 2 In the **Import Project Settings** dialog box, select the configuration object that you want to use.



MATLAB Coder imports the settings saved in the configuration object and uses them as the current project settings.

---

**Note:** When you import a `coder.MexCodeConfig` object, if the project output type is not already set to `Instrumented MEX Function`, the output type is set to `MEX Function`.

---

### See Also

- “Build Setting Configuration” on page 19-19
- `coder.config`
- “Convert MATLAB Coder Project to MATLAB Script” on page 19-37

## Convert MATLAB Coder Project to MATLAB Script

You can convert a MATLAB Coder project to the equivalent script of MATLAB commands. The script reproduces the project in a configuration object and runs the `codegen` command. You can:

- Move from a project workflow to a command-line workflow.
  - Save the project as a text file that you can share.
- 1 Suppose that the project file, `myproject.prj`, is on the search path. Convert `myproject` to the script named `myscript.m`.

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

`myscript.m` appears in the current working folder. If a file with the name `myscript.m` exists, the `coder` command overwrites it. If you omit the `-script` option, the `coder` command writes the script to the Command Window.

- 2 Make sure that the entry-point functions that are arguments to `codegen` in the script are on the search path.
- 3 Run the script.

```
myscript
```

The following variables appear in the base workspace.

<code>cfg</code>	configuration object
<code>ARGS</code>	types of input arguments, if the project has entry-point function inputs
<code>GLOBALS</code>	initial values of global data, if the project has global data

`cfg`, `ARGS`, and `GLOBALS` appear in the workspace only after you run the script. The type of configuration object depends on the project output type.

Project Output Type	Configuration Object
MEX Function	<code>coder.MexCodeConfig</code>
C/C++ Static Library	Without an Embedded Coder license: <code>coder.CodeConfig</code>
C/C++ Dynamic Library	
C/C++ Executable	

Project Output Type	Configuration Object
	With an Embedded Coder license: <code>coder.EmbeddedCodeConfig</code>

You can import the settings from the configuration object `cfg` into a project. See “Share Build Configuration Settings” on page 19-34.

If a project includes automated fixed-point conversion, the `-tocode` option of the `coder` command generates a pair of scripts for fixed-point conversion and fixed-point code generation. For an example, see “Convert Fixed-Point Conversion Project to MATLAB Scripts”.

### See Also

`coder`



## Primary Function Input Specification

### In this section...

“Why You Must Specify Input Properties” on page 19-39

“Properties to Specify” on page 19-39

“Rules for Specifying Properties of Primary Inputs” on page 19-42

“Methods for Defining Properties of Primary Inputs” on page 19-43

“Define Input Properties by Example at the Command Line” on page 19-44

“Specify Constant Inputs at the Command Line” on page 19-46

“Specify Variable-Size Inputs at the Command Line” on page 19-47

### Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, MATLAB Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, to MATLAB Coder. If your primary function has no input parameters, MATLAB Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the primary function.

If you use the tilde (~) character to specify unused function inputs:

- In MATLAB Coder projects, if you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.
- When generating code with `codegen`, you must specify the type of these inputs using the `-args` option.

### Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For...	Specify properties...				
	Class	Size	Complexity	numerictype	fimath

For...	Specify properties...				
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	<p><b>Specify properties for each field according to its class</b></p> <p>When a primary input is a structure, the code generation software treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition:</p> <ul style="list-style-type: none"> <li>• For each field of input structures, specify class, size, and complexity.</li> <li>• For each field that is fixed-point class, also specify <code>numericType</code>, and <code>fimath</code>.</li> </ul>				
Other inputs	✓	✓	✓		

### Default Property Values

MATLAB Coder assigns the following default values for properties of primary function inputs.

Property	Default
class	double
size	scalar
complexity	real
numericType	No default
fimath	MATLAB default <code>fimath</code> object

### Specifying Default Values for Structure Fields

In most cases, when you don't explicitly specify values for properties, MATLAB Coder uses defaults except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you might need to specify default values for properties of structure fields. For examples, see “Specifying Class and Size of Scalar Structure” and “Specifying Class and Size of Structure Array”.

### Specifying Default `fimath` Values for MEX Functions

MEX functions generated with MATLAB Coder use the default `fimath` value in effect at compile time. If you do not specify a default `fimath` value, MATLAB Coder uses the MATLAB default `fimath`. The MATLAB factory default has the following properties:

```

RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
CastBeforeSum: true

```

For more information, see “`fimath` for Sharing Arithmetic Rules”.

When running MEX functions that depend on the default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time warning, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```

function y = test %#codegen
y = fi(0);

```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` relies on the default `fimath` object in effect at compile time. At the MATLAB prompt, generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`:

```

codegen test
% codegen generates a MEX function, test_mex,
% in the current folder

```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```

test_mex

ans =

    0

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 16
      FractionLength: 15

```

Now create a local MATLAB `fimath` value. so you no longer use the default setting:

```

F = fimath('RoundingMethod','Floor');

```

Finally, clear the MEX function from memory and rerun it:

```

clear test_mex
test_mex

```

The mismatch is detected and causes an error:

```
??? This function was generated with a different default
fimath than the current default.
```

```
Error in ==> test_mex
```

### Supported Classes

The following table presents the class names supported by MATLAB Coder.

Class Name	Description
logical	Logical array of true and false values
char	Character array
int8	8-bit signed integer array
uint8	8-bit unsigned integer array
int16	16-bit signed integer array
uint16	16-bit unsigned integer array
int32	32-bit signed integer array
uint32	32-bit unsigned integer array
int64	64-bit signed integer array
uint64	64-bit unsigned integer array
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
struct	Structure array
embedded.fi	Fixed-point number array

### Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules.

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.

- For each primary function input whose class is fixed point (`fi`), you must specify the input `numericType` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

## Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
“Specifying Properties of Primary Function Inputs in a Project”	<ul style="list-style-type: none"> <li>• If you are working in a MATLAB Coder project, easy to use</li> <li>• Does not alter original MATLAB code</li> <li>• MATLAB Coder saves the definitions in the project file</li> </ul>	<ul style="list-style-type: none"> <li>• Not efficient for specifying memory-intensive inputs such as large structures and arrays</li> </ul>
“Define Input Properties by Example at the Command Line” on page 19-44  <b>Note:</b> If you define input properties programmatically in the MATLAB file, you cannot use this method	<ul style="list-style-type: none"> <li>• Easy to use</li> <li>• Does not alter original MATLAB code</li> <li>• Designed for prototyping a function that has a small number of primary inputs</li> </ul>	<ul style="list-style-type: none"> <li>• Must be specified at the command line every time you invoke <code>codegen</code> (unless you use a script)</li> <li>• Not efficient for specifying memory-intensive inputs such as large structures and arrays</li> </ul>
“Define Input Properties Programmatically in the MATLAB File”	<ul style="list-style-type: none"> <li>• Integrated with MATLAB code; no need to redefine properties each time you invoke MATLAB Coder</li> <li>• Provides documentation of property specifications in the MATLAB code</li> <li>• Efficient for specifying memory-intensive inputs such as large structures</li> </ul>	<ul style="list-style-type: none"> <li>• Uses complex syntax</li> <li>• MATLAB Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.</li> </ul>

## Define Input Properties by Example at the Command Line

- “Command Line Option `-args`” on page 19-44
- “Rules for Using the `-args` Option” on page 19-44
- “Specifying Properties of Primary Inputs by Example at the Command Line” on page 19-45
- “Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line” on page 19-45

### Command Line Option `-args`

The `codegen` function provides a command-line option `-args` for specifying the properties of primary (entry-point) function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you generate code for the MATLAB function with `codegen`.

If you have a test function or script that calls the entry-point MATLAB function with the required types, you can use `coder.getArgTypes` to determine the types of the function inputs. `coder.getArgTypes` returns a cell array of `coder.Type` objects that you can pass to `codegen` using the `-args` option. See “Specifying General Properties of Primary Inputs” for `codegen`.

### Rules for Using the `-args` Option

When using the `-args` command-line option to define properties by example, follow these rules:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

---

**Note:** If you specify an empty cell array with the `-args` option, `codegen` interprets this to mean that the function takes no inputs; a compile-time error occurs if the function does have inputs.

---

## Specifying Properties of Primary Inputs by Example at the Command Line

Consider a MATLAB function that adds its two inputs:

```
function y = mcf(u,v)
%#codegen
y = u + v;
```

The following examples show how to specify different properties of the primary inputs `u` and `v` by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

```
codegen mcf -args {0,0}
```

- Use a literal cell array of constants to specify that input `u` is an unsigned 16-bit, 1-by-4 vector and input `v` is a scalar double:

```
codegen mcf -args {zeros(1,4,'uint16'),0}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = uint8([1;2;3;4])
b = uint8([5;6;7;8])
ex = {a,b}
codegen mcf -args ex
```

## Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line

To generate a MEX function or C/C++ code for fixed-point MATLAB code, you must install Fixed-Point Designer software.

Consider a MATLAB function that calculates the square root of a fixed-point number:

```
%#codegen
function y = sqrtfi(x)
y = sqrt(x);
```

To specify the properties of the primary fixed-point input `x` by example, follow these steps:

- 1 Define the `numericType` properties for `x`, for example:

```
T = numericType('WordLength',32,...
```

```
'FractionLength',23,...  
'Signed',true);
```

- 2 Define the `fimath` properties for `x`, for example:

```
F = fimath('SumMode','SpecifyPrecision',...  
          'SumWordLength',32,...  
          'SumFractionLength',23,...  
          'ProductMode','SpecifyPrecision',...  
          'ProductWordLength',32,...  
          'ProductFractionLength',23);
```

- 3 Create a fixed-point variable with the `numericType` and `fimath` properties that you just defined, for example:

```
myeg = { fi(4.0,T,F) };
```

- 4 Compile the function `sqrtfi` using the `codegen` command, passing the variable `myeg` as the argument to the `-args` option, for example:

```
codegen sqrtfi -args myeg;
```

## Specify Constant Inputs at the Command Line

If you know that your primary inputs will not change at run time, you can reduce overhead in the generated code by specifying that the primary inputs are constant values. Constant inputs are commonly used for flags that control how an algorithm executes and values that specify the sizes or types of data.

To specify that inputs are constants, use the `-args` command-line option with a `coder.Constant` object. To specify that an input is a constant with the size, class, complexity, and value of `constant_input`, use the following syntax:

```
-args {coder.Constant(constant_input)}
```

### Calling Functions with Constant Inputs

The code generation software compiles constant function inputs into the generated code. In the generated C or C++ code, function signatures do not contain the constant inputs. By default, MEX function signatures contain the constant inputs. When you call a MEX function, you must provide the compile-time constant values. The constant input values must match the compile-time values. You can control whether a MEX function signature includes constant inputs and whether the constant input values must match the compile-time values. See “Control Constant Inputs in MEX Function Signatures”.



### Specifying a Structure as a Constant Input

Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
y = zeros(p.rows,p.cols) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
codegen rowcol -args {0,coder.Constant(tmp)}
```

### Specify Variable-Size Inputs at the Command Line

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. You can define inputs to have one or more variable-size dimensions — and specify their upper bounds — using the `-args` option and `coder.typeof` function:

```
-args {coder.typeof(example_value, size_vector, variable_dims)}
Specifies a variable-size input with:
```

- Same class and complexity as *example\_value*
- Same size and upper bounds as *size\_vector*
- Variable dimensions specified by *variable\_dims*

When you enable dynamic memory allocation, you can specify `Inf` in the size vector for dimensions with unknown upper bounds at compile time.

When *variable\_dims* is a scalar, it is applied to all the dimensions, with the following exceptions:

- If the dimension is 1 or 0, which are fixed.
- If the dimension is unbounded, which is always variable size.

For more information, see `coder.typeof` and “Generate Code for Variable-Size Data”.

### Specifying a Variable-Size Vector Input

- 1 Write a function that computes the average of every `n` elements of a vector `A` and stores them in a vector `B`:

```
function B = nway(A,n) %#codegen
% Compute average of every N elements of A and put them in B.

coder.extrinsic('error');
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
        B(i) = mean(A(k + (0:n-1)));
        k = k + n;
    end
else
    B = zeros(1,0);
    error('n <= 0 or does not divide number of elements evenly');
end
```

- 2 Specify the first input `A` as a vector of double values. Its first dimension stays fixed in size and its second dimension can grow to an upper bound of 100. Specify the second input `n` as a double scalar.

```
codegen -report nway -args {coder.typeof(0,[1 100],1),1}
```

- 3 As an alternative, assign the `coder.typeof` expression to a MATLAB variable, then pass the variable as an argument to `-args`:

```
vareg = coder.typeof(0,[1 100],1)
codegen -report nway -args {vareg, 0}
```

## Control Constant Inputs in MEX Function Signatures

### In this section...

- “Control MEX Function Signature Using the Project Interface” on page 19-49
- “Control MEX Function Signature at the Command-Line Interface” on page 19-49
- “Options for Controlling Constant Inputs in MEX Function Signatures” on page 19-49
- “Call MEX Function with a Constant Input” on page 19-51
- “See Also” on page 19-52

You can control whether a generated MEX function signature includes constant inputs. If you want to use the same test file to run the original MATLAB function and the MEX function, then the MEX function signature must contain the constant inputs. You can also control whether the run-time values of the constant inputs must match the compile-time values. Checking that the values match can slow down execution speed.

### Control MEX Function Signature Using the Project Interface

- 1 On the **Build** tab **Settings** pane, set **Output type** to MEX Function.
- 2 On the Project Settings dialog box **All Settings** tab, set **Constant Inputs** to one of the menu options. See “Options for Controlling Constant Inputs in MEX Function Signatures” on page 19-49.

### Control MEX Function Signature at the Command-Line Interface

- 1 Create a code configuration object for MEX code generation.
- 2 Set the `ConstantInputs` parameter to `'CheckValues'`, `'IgnoreValues'`, or `'Remove'` For example:

```
mexcfg.ConstantInputs = 'IgnoreValues';
```

For a description of the options, see “Options for Controlling Constant Inputs in MEX Function Signatures” on page 19-49

### Options for Controlling Constant Inputs in MEX Function Signatures

The following table lists the options for the:

- **Constant Inputs** setting in a project with **Output Type** set to MEX.
- `ConstantInputs` property in a configuration object for MEX code generation.

Constant Inputs (Project)	ConstantInputs (Configuration Object)	Description
Check values at run time (default)	'CheckValues'	<ul style="list-style-type: none"> <li>• The MEX function signature includes the constant inputs. When you call the function, you must provide the constant inputs.</li> <li>• The run-time values of the constant inputs must match the compile-time values. When you call the function, you must provide the value that was used at compile-time.</li> <li>• Allows you to use the same test file to run the original MATLAB algorithm and the MEX function.</li> <li>• Slows down execution of the MEX function.</li> <li>• This setting is the default.</li> </ul>
Ignore input value	'IgnoreValues'	<ul style="list-style-type: none"> <li>• The MEX function signature includes the constant inputs. When you call the function, you must provide the constant inputs.</li> <li>• The run-time values of the constant inputs can differ from the compile-time values.</li> <li>• Allows you to use the same test file to run the original MATLAB algorithm and the MEX function.</li> </ul>

Constant Inputs (Project)	ConstantInputs (Configuration Object)	Description
Remove from MEX signature	'Remove'	The MEX function signature does not include the constant inputs. When you call the function, you do not provide the constant inputs.

## Call MEX Function with a Constant Input

This example shows how to call MEX functions that have constant inputs. It shows how to use the `ConstantInputs` parameter to control whether the MEX function signature includes constant inputs and whether the constant input values must match the compile-time values.

Write a function `identity` that copies its input to its output.

```
function y = identity(u) %#codegen
y = u;
```

Create a code configuration object for MEX code generation.

```
cfg = coder.config('mex');
```

Generate a MEX function `identity_mex` with the constant input 42.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex`. You must provide the input 42.

```
identity_mex(42)
```

```
ans =
```

```
    42
```

Configure `ConstantInputs` so that the MEX function does not check that the input value matches the compile-time value.

```
cfg.ConstantInputs = 'IgnoreValues';
```

Generate `identity_mex` with the new configuration.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex` with a constant input value other than 42 .

```
identity_mex(50)
```

```
ans =
```

```
42
```

The MEX function ignored the input value 50.

Configure `ConstantInputs` so that the MEX function does not include the constant input.

```
cfg.ConstantInputs = 'Remove';
```

Generate `identity_mex` with the new configuration.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex`. Do not provide the input value .

```
identity_mex()
```

```
ans =
```

```
42
```

## See Also

- “Specify Constant Inputs at the Command Line”
- “Define Constant Input Parameters in a Project”

## Define Input Properties Programmatically in the MATLAB File

With MATLAB Coder, you use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

### In this section...

“How to Use `assert` with MATLAB Coder” on page 19-53

“Rules for Using `assert` Function” on page 19-59

“Specifying General Properties of Primary Inputs” on page 19-59

“Specifying Properties of Primary Fixed-Point Inputs” on page 19-60

“Specifying Class and Size of Scalar Structure” on page 19-61

“Specifying Class and Size of Structure Array” on page 19-62

### How to Use `assert` with MATLAB Coder

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

You must use one of the following methods when specifying input properties using the `assert` function. Use the exact syntax that is provided; do not modify it.

- “Specify Any Class” on page 19-54
- “Specify `fi` Class” on page 19-54
- “Specify Structure Class” on page 19-55
- “Specify Fixed Size” on page 19-55
- “Specify Scalar Size” on page 19-55
- “Specify Upper Bounds for Variable-Size Inputs” on page 19-56
- “Specify Inputs with Fixed- and Variable-Size Dimensions” on page 19-56
- “Specify Size of Individual Dimensions” on page 19-56
- “Specify Real Input” on page 19-57
- “Specify Complex Input” on page 19-57
- “Specify `numericType` of Fixed-Point Input” on page 19-57
- “Specify `fimath` of Fixed-Point Input” on page 19-58

- “Specify Multiple Properties of Input” on page 19-58

### Specify Any Class

```
assert ( isa ( param, 'class_name' ) )
```

Sets the input parameter *param* to the MATLAB class *class\_name*. For example, to set the class of input *U* to a 32-bit signed integer, call:

```
...  
assert(isa(U, 'int32'));  
...
```

If you set the class of an input parameter to *fi*, you must also set its *numericType*, see “Specify numericType of Fixed-Point Input” on page 19-57. You can also set its *fimath* properties, see “Specify fimath of Fixed-Point Input” on page 19-58. If you do not set the *fimath* properties, *codegen* uses the MATLAB default *fimath* value.

If you set the class of an input parameter to *struct*, you must specify the properties of all fields in the order that they appear in the structure definition.

### Specify fi Class

```
assert ( isfi ( param ) )  
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class *fi* (fixed-point numeric object). For example, to set the class of input *U* to *fi*, call:

```
...  
assert(isfi(U));  
...  
  
or  
  
...  
assert(isa(U, 'embedded.fi'));  
...
```

If you set the class of an input parameter to *fi*, you must also set its *numericType*, see “Specify numericType of Fixed-Point Input” on page 19-57. You can also set its *fimath* properties, see “Specify fimath of Fixed-Point Input” on page 19-58. If you do not set the *fimath* properties, *codegen* uses the MATLAB default *fimath* value.



**Specify Structure Class**

```
assert ( isstruct ( param ) )
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U, 'struct'));
...
```

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

**Specify Fixed Size**

```
assert ( all ( size (param) == [dims ] ) )
```

Sets the input parameter *param* to the size specified by dimensions *dims*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

**Specify Scalar Size**

```
assert ( isscalar (param) )
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. To set the size of input `U` to scalar, call:

```
...
assert(isscalar(U));
...
or
...

```

```
assert(all(size(U)== [1]));  
...
```

### Specify Upper Bounds for Variable-Size Inputs

```
assert ( all(size(param)<=[N0 N1 ...]));  
assert ( all(size(param)<[N0 N1 ...]));
```

Sets the upper-bound size of each dimension of input parameter *param*. To set the upper-bound size of input *U* to be less than or equal to a 3-by-2 matrix, call:

```
assert(all(size(U)<=[3 2]));
```

---

**Note:** You can also specify upper bounds for variable-size inputs using `coder. varsize`.

---

### Specify Inputs with Fixed- and Variable-Size Dimensions

```
assert ( all(size(param)>=[M0 M1 ...]));  
assert ( all(size(param)<=[N0 N1 ...]));
```

When you use `assert(all(size(param)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:

- You must also specify an upper-bound size for each dimension of the input parameter.
- For each dimension, *k*, the lower-bound *M<sub>k</sub>* must be less than or equal to the upper-bound *N<sub>k</sub>*.
- To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
- Bounds must be non-negative.

To fix the size of the first dimension of input *U* to 3 and set the second dimension as variable size with upper-bound of 2, call:

```
assert(all(size(U)>=[3 0]));  
assert(all(size(U)<=[3 2]));
```

### Specify Size of Individual Dimensions

```
assert (size(param, k)==Nk);  
assert (size(param, k)<=Nk);  
assert (size(param, k)<Nk);
```

You can specify individual dimensions as well as specifying all dimensions simultaneously or instead of specifying all dimensions simultaneously. The following rules apply:

- You must specify the size of each dimension at least once.
- The last dimension specification takes precedence over earlier specifications.

Sets the upper-bound size of dimension *k* of input parameter *param*. To set the upper-bound size of the first dimension of input *U* to 3, call:

```
assert(size(U,1)<=3)
```

To fix the size of the second dimension of input *U* to 2, call:

```
assert(size(U,2)==2)
```

### Specify Real Input

```
assert ( isreal ( param ) )
```

Specifies that the input parameter *param* is real. To specify that input *U* is real, call:

```
...
assert(isreal(U));
...
```

### Specify Complex Input

```
assert ( ~isreal ( param ) )
```

Specifies that the input parameter *param* is complex. To specify that input *U* is complex, call:

```
...
assert(~isreal(U));
...
```

### Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the `numerictype` properties of `fi` input parameter *fiparam* to the `numerictype` object *T*. For example, to specify the `numerictype` property of fixed-point input *U* as a signed `numerictype` object *T* with 32-bit word length and 30-bit fraction length, use the following code:

```
 %#codegen
 ...
 % Define the numerictype object.
 T = numerictype(1, 32, 30);

 % Set the numerictype property of input U to T.
 assert(isequal(numerictype(U),T));
 ...
```

### Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter `fiparam` to the `fimath` object `F`. For example, to specify the `fimath` property of fixed-point input `U` so that it saturates on integer overflow, use the following code:

```
 %#codegen
 ...
 % Define the fimath object.
 F = fimath('OverflowMode','saturate');

 % Set the fimath property of input U to F.
 assert(isequal(fimath(U),F));
 ...
```

If you do not specify the `fimath` properties using `assert`, `codegen` uses the MATLAB default `fimath` value.

### Specify Multiple Properties of Input

```
assert ( function1 ( params ) &&
         function2 ( params ) &&
         function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
 %#codegen
 ...
 assert(isa(U,'double') &&
        ~isreal(U) &&
        all(size(U) == [3 3]) &&
        isa(V,'uint16'));
 ...
```

## Rules for Using `assert` Function

When using the `assert` function to specify the properties of primary function inputs, follow these rules:

- Call `assert` functions at the beginning of the primary function, before control-flow operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- Use the `assert` function with MATLAB Coder only for specifying properties of primary function inputs before converting your MATLAB code to C/C++ code.
- If you set the class of an input parameter to `fi`, you must also set its `numericType`. See “Specify `numericType` of Fixed-Point Input” on page 19-57. You can also set its `fimath` properties. See “Specify `fimath` of Fixed-Point Input” on page 19-58. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order that they appear in the structure definition.
- When you use `assert(all(size(param) >= [M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:
  - You must also specify an upper-bound size for each dimension of the input parameter.
  - For each dimension,  $k$ , the lower-bound  $M_k$  must be less than or equal to the upper-bound  $N_k$ .
  - To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
  - Bounds must be non-negative.
- If you specify individual dimensions, the following rules apply:
  - You must specify the size of each dimension at least once.
  - The last dimension specification takes precedence over earlier specifications.

## Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `mcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs:

Input	Property	Value
pennywhistle	class	int16
	size	220500-by-1 vector
	complexity	real (by default)
win	class	double
	size	1024-by-1 vector
	complexity	real (by default)

```

%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...

```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands:

```

%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...

```

## Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Designer software.

In the following example, the primary MATLAB function `mcsqrtfi` takes one fixed-point input `x`. The code specifies the following properties for this input.

Property	Value
class	fi

Property	Value
numericity	numericity object T, as specified in the primary function
fimath	fimath object F, as specified in the primary function
size	scalar
complexity	real (by default)

```
function y = mcsqrtfi(x) %#codegen
T = numericity('WordLength',32,'FractionLength',23,...
              'Signed',true);
F = fimath('SumMode','SpecifyPrecision',...
          'SumWordLength',32,'SumFractionLength',23,...
          'ProductMode','SpecifyPrecision',...
          'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numericity(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

## Specifying Class and Size of Scalar Structure

Assume you have defined **S** as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

Here is code that specifies the class and size of **S** and its fields when passed as an input to your MATLAB function:

```
function y = fcn(S) %#codegen

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8'));
...
```

In most cases, when you don't explicitly specify values for properties, MATLAB Coder uses defaults — except for structure fields. The only way to name a field in a structure

is to set at least one of its properties. As a minimum, you must specify the class of a structure field

## Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined **S** as the following 1-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input **S** using the first element of the array:

```
 %#codegen
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [1 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. As a minimum, you must specify the class of all fields.



# Speed Up Compilation

## In this section...

“Generate Code Only” on page 19-63

“Disable Compiler Optimization” on page 19-63

## Generate Code Only

If you select this option, MATLAB Coder does not invoke the make command or generate compiled object code. When you want to iterate rapidly between modifying MATLAB code and generating C/C++ code and you want to inspect the generated code, this option saves you time during the development cycle .

### In the Project Interface

On the project **Build** tab, select **Generate code only**.

### At the Command Line

Use the `codegen -c` option to only generate code without invoking the make command. For example, to generate code only for a function, `foo`, that takes one single, scalar input:

```
codegen -c foo -args {single(0)}
```

For more information and a complete list of compilation options, see `codegen`.

## Disable Compiler Optimization

Turning compiler optimizations off shortens compile time, but increases run time.

### In the Project Interface

- 1 On the MATLAB Coder project **Build** tab, verify that the **Output type** is **C/C++ Static Library**, **C/C++ Dynamic Library** or **C/C++ Executable**.
- 2 On the **Build** tab, click the **More settings** link.
- 3 In the **Project Settings** dialog box **All Settings** tab, under **Advanced**, set **Compiler optimization level** to **Off**.

### At the Command Line

- 1 Create a code generation configuration object for C/C++ library or executable. For example, for a static library:

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

- 2 Set the CCompilerOptimization to Off.

```
cfg.CCompilerOptimization='Off';
```

## Paths and File Infrastructure Setup

### In this section...

“Compile Path Search Order” on page 19-65

“Specifying Folders to Search for Custom Code” on page 19-65

“Naming Conventions” on page 19-66

### Compile Path Search Order

MATLAB Coder resolves MATLAB functions by searching first on the *code generation path* and then on the MATLAB path. The code generation path contains the current folder and the code generation libraries. By default, unless MATLAB Coder determines that a function should be extrinsic or you explicitly declare the function to be extrinsic, MATLAB Coder tries to compile and generate code for functions it finds on the path. MATLAB Coder does not compile extrinsic functions, but rather dispatches them to the MATLAB interpreter for execution. See “Resolution of Function Calls for Code Generation”.

### Specifying Folders to Search for Custom Code

If you want to integrate custom code — such as source, header, and library files — with the generated code, you can specify additional folder to search. The following table describes how to specify these search paths. The path should not contain spaces, as this can lead to code generation failures in certain operating system configurations. If the path contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not be able to find files on this path.

To specify additional folders:	Do this:
Using the MATLAB Coder interface	<p>On the MATLAB Coder project <b>Build</b> tab:</p> <ol style="list-style-type: none"> <li><b>1</b> Click the <b>More settings</b> link.</li> <li><b>2</b> In the <b>Project Settings</b> dialog box, click the <b>Paths</b> tab.</li> <li><b>3</b> For the <b>Search paths</b> field, either browse to add a folder to the search path or enter the full path. The search path must not contain spaces.</li> </ol>
At the command line	Use the <code>codegen</code> function -I option.

## Naming Conventions

MATLAB Coder enforces naming conventions for MATLAB functions and generated files.

- “Reserved Prefixes” on page 19-66
- “Reserved Keywords” on page 19-66
- “Conventions for Naming Generated files” on page 19-69

### Reserved Prefixes

MATLAB Coder reserves the prefix `eml` for global C/C++ functions and variables in generated code. For example, MATLAB for code generation run-time library function names begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C/C++ functions or primary MATLAB functions with the prefix `eml`.

### Reserved Keywords

- “C Reserved Keywords” on page 19-66
- “C++ Reserved Keywords” on page 19-67
- “Reserved Keywords for Code Generation” on page 19-67
- “MATLAB Coder Code Replacement Library Keywords” on page 19-68

MATLAB Coder software reserves certain words for its own use as keywords of the generated code language. MATLAB Coder keywords are reserved for use internal to MATLAB Coder software and should not be used in MATLAB code as identifiers or function names. C reserved keywords should also not be used in MATLAB code as identifiers or function names. If your MATLAB code contains reserved keywords, the code generation build does not complete and an error message is displayed. To address this error, modify your code to use identifiers or names that are not reserved.

If you are generating C++ code using the MATLAB Coder software, in addition, your MATLAB code must not contain the “C++ Reserved Keywords” on page 19-67.

### C Reserved Keywords

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>

char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

### C++ Reserved Keywords

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t
export	private	throw	

### Reserved Keywords for Code Generation

abs	fortran	localZCE	rtNaN
asm	HAVESTDIO	localZCSV	SeedFileBuffer
bool	id_t	matrix	SeedFileBufferLen
boolean_T	int_T	MODEL	single
byte_T	int8_T	MT	TID01EQ
char_T	int16_T	NCSTATES	time_T
cint8_T	int32_T	NULL	true
cint16_T	int64_T	NUMST	TRUE
cint32_T	INTEGER_CODE	pointer_T	uint_T
creal_T	LINK_DATA_BUFFER_SIZE	PROFILING_ENABLED	uint8_T
creal32_T	LINK_DATA_STREAM	PROFILING_NUM_SAMPLES	uint16_T
creal64_T	localB	real_T	uint32_T
cuint8_T	localC	real32_T	uint64_T

cuint16_T	localDWork	real64_T	UNUSED_PARAMETER
cuint32_T	localP	RT	USE_RTMODEL
ERT	localX	RT_MALLOC	VCAST_FLUSH_DATA
false	localXdis	rtInf	vector
FALSE	localXdot	rtMinusInf	

### MATLAB Coder Code Replacement Library Keywords

The list of code replacement library (CRL) reserved keywords for your development environment varies depending on which CRLs currently are registered. Beyond the default ANSI, ISO, and GNU® CRLs provided with MATLAB Coder software, additional CRLs might be registered and available for use if you have installed other products that provide CRLs (for example, a target product), or if you have used Embedded Coder APIs to create and register custom CRLs.

To generate a list of reserved keywords for the CRLs currently registered in your environment, use the following MATLAB function:

```
cr1_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers()
```

This function returns an array of CRL keyword strings. Specifying the return argument is optional.

---

**Note:** To list the CRLs currently registered in your environment, use the MATLAB command `RTW.viewTf1`.

---

To generate a list of reserved keywords for the CRL that you are using to generate code, call the function passing the name of the CRL as displayed in the **Code replacement library** menu on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. For example,

```
cr1_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU99 (GNU)')
```

Here is a partial example of the function output:

```
>> cr1_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU99 (GNU)')
cr1_ids =
    'exp10'
    'exp10f'
```

```

'acosf'
'acoshf'
'asinf'
'asinhf'
'atanf'
'atanhf'
...
'rt_lu_cplx'
'rt_lu_cplx_sgl'
'rt_lu_real'
'rt_lu_real_sgl'
'rt_mod_boolean'
'rt_rem_boolean'
'strcpy'
'utAssert'
    
```

---

**Note:** Some of the returned keyword strings appear with the suffix `$N`, for example, `'rt_atan2$N'`. `$N` expands into the suffix `_snf` only if nonfinite numbers are supported. For example, `'rt_atan2$N'` represents `'rt_atan2_snf'` if nonfinite numbers are supported and `'rt_atan2'` if nonfinite numbers are not supported. As a precaution, you should treat both forms of the keyword as reserved.

---

### Conventions for Naming Generated files

The following table describes how MATLAB Coder names generated files. MATLAB Coder follows MATLAB conventions by providing platform-specific extensions for MEX files.

Platform	MEX File Extension	MATLAB Coder Library Extension	MATLAB Coder Executable Extension
Linux Torvalds' Linux (32-bit)	.mexglx	.a	None
Linux x86-64	.mexa64	.a	None
Microsoft Windows (32-bit)	.mexw32	.lib	.exe
Windows x64	.mexw64	.lib	.exe

## Generate Code for Multiple Entry-Point Functions

### In this section...

“Advantages of Generating Code for More Than One Entry-Point Function” on page 19-70

“Generating Code for More Than One Entry-Point Function Using the Project Interface” on page 19-70

“Generating Code for More Than One Entry-Point Function at the Command Line” on page 19-72

“How to Call an Entry-Point Function in a MEX Function” on page 19-74

“How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code” on page 19-74

### Advantages of Generating Code for More Than One Entry-Point Function

Generating a single C/C++ library for more than one entry-point MATLAB function allows you to:

- Create C/C++ libraries containing multiple, compiled MATLAB files to integrate with larger C/C++ applications.
- Share code efficiently between library functions.
- Communicate between library functions using shared memory.

Generating a MEX function for more than one entry-point function allows you to validate entry-point interactions in MATLAB before creating a C/C++ library.

### Generating Code for More Than One Entry-Point Function Using the Project Interface

In the project, in the **Entry-Point Files** pane on the **Overview** tab, click the **Add files** link. Browse to the file that you want to add. Repeat this action for each entry-point file.

By default, MATLAB Coder:

- Lists the entry-point files alphabetically.
- Generates a MEX function in the current folder. MATLAB Coder names the MEX function , *fun\_1\_mex*. *fun\_1* is the name of the first entry-point function.



- Stores generated files in the subfolder `codegen/mex/fun_1/`.

### Generating a MEX Function with Two Entry-Point Functions Using the Project Interface

Generate a MEX function with two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector.

- 1 In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

- 2 In the same folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

- 3 In the same folder, set up a MATLAB Coder project.

- a At the MATLAB command line, enter:

```
coder -new ep.prj
```

By default, the project opens in the MATLAB workspace on the right side.

- b On the project **Overview** tab, click the **Add files** link. Browse to the file `ep1.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab, and the input is undefined.

- c Define the type of input `u`.

- i On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select **single**.

- ii From the list of size options, select `1 x 1` to specify that `u` is a scalar.

- d On the project **Overview** tab, click the **Add files** link. Browse to the file `ep2.m`. Click **OK** to add the file to the project.

The file is displayed on the **Overview** tab, and the inputs are undefined.

- e Define the type of input `u`.

- i On the **Overview** tab, click the field to the right of the input parameter `u` and then, from the list of input options, select **double**.

- ii From the list of size options, select `1 x 1` to specify that `u` is a scalar.

- f** Repeat the previous step for input `v`, setting the **Size** to `2x1`.
- 4** In the MATLAB Coder project, click the **Build** tab.
- By default, the **Output type** is MEX function and the **Output file** is `ep1_mex`.
- 5** On this tab, click the **Build** button to generate a MEX function using the default project settings.

MATLAB Coder builds the project and, by default, generates a MEX function, `ep1_mex`, in the current folder. MATLAB Coder also generates other supporting files in a subfolder called `codegen/mex/ep1_mex`. MATLAB Coder uses the name of the MATLAB function as the root name for the generated files and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 19-66.

You can now test your MEX function in MATLAB. For more information, see “How to Call an Entry-Point Function in a MEX Function” on page 19-74.

### Generating a C Static Library with Two Entry-Point Functions Using the Project Interface

You can generate a C static library with two entry-point functions, `ep1` and `ep2`, following the same project setup steps that you use to generate a MEX function. (See Generating a MEX Function with Two Entry-Point Functions Using the Project Interface.) When you build the project, set the **Output type** to `C/C++ Static Library`.

MATLAB Coder builds the project and generates a C library, `ep1`, and supporting files in the default folder, `codegen/lib/ep1`.

You can now test your library. For more information, see “How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code” on page 19-74.

### Generating Code for More Than One Entry-Point Function at the Command Line

To generate code for more than one entry-point function, use the following syntax, where `global_options` applies to functions, `fun_1` through `fun_n`, and `options_n` applies only to the preceding function `fun_n`.

```
codegen -global_options fun_1 -options_1 ... fun_n -options_n
```

By default, `codegen`:

- Generates a MEX function in the current folder. `codegen` names the MEX function, `fun_mex`. `fun` is the name of the alphabetically first entry-point function.

Stores generated files in the subfolder `codegen/mex/fun_1`. `fun_1` is the name of the first entry-point function.

You can specify the output file name and subfolder name using the `-o` option.

```
codegen -o out_fun fun_1 -options_1 ... fun_n -options_n
```

In this case, `codegen`:

- Generates a MEX function named `out_fun_mex` in the current folder.
- Stores generated files in the subfolder `codegen/mex/out_fun`.

For more information on setting build options at the command line, see `codegen`.

### Generating a MEX Function with Two Entry-Point Functions at the Command Line

Generate a MEX function with two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Using the `-o` option, name the generated MEX function `sharedmex`.

```
codegen -o sharedmex ep1 -args single(0) ep2 -args { 0, zeros(1,1024) }
```

`codegen` generates a MEX function named `sharedmex.mex` in the current folder and stores generated files in the subfolder `codegen/mex/sharedmex`.

### Generating a C/C++ Static Library with Two Entry-Point Functions at the Command Line

Generate standalone C/C++ code and compile it to a library for two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Use the `-config:lib` option to specify that the target is a library. Using the `-o` option, name the generated library `sharedlib`.

```
codegen -config:lib -o sharedlib ep1 -args single(0) ep2 ...
    -args { 0, zeros(1,1024) }
```

`codegen` generates C/C++ library code in the `codegen/lib/sharedlib` folder.

For information on viewing entry-point functions in the code generation report, see “Code Generation Reports” on page 19-167.

## How to Call an Entry-Point Function in a MEX Function

To call an entry-point function in a MEX function that has more than one entry point, use this syntax:

```
MEX_Function('entry_point_function_name',  
    ... entry_point_function_param1,  
    ... , entry_point_function_paramn)
```

### Calling an Entry-Point Function in a MEX Function

Consider a MEX function, `sharedmex`, that has entry-point functions `ep1` and `ep2`. Entry-point function `ep1` takes one single scalar input and `ep2` takes two inputs, a double scalar and a double vector.

To call `ep1` with an input parameter `u`, enter:

```
sharedmex('ep1', u)
```

To call `ep2` with input parameters `u` and `v`, enter:

```
sharedmex('ep2', u, v)
```

## How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code

To call an entry-point function in a C/C++ library function from C/C++ code, write a `main` function in C/C++ that:

- Includes the generated header files, which contain the function prototypes for the entry-point functions.
- Calls the `initialize` function before calling the entry-point functions for the first time.
- Calls the `terminate` function after calling the entry-point functions for the last time.
- Configures your target to integrate this custom C/C++ main function with your generated code, as described in “Specify External File Locations” on page 22-12.
- Generates the C/C++ executable using `codegen`.

See the example, “Call a C Static Library Function from C Code” on page 22-2.

## Generate Code for Global Data

### In this section...

“Workflow” on page 19-75

“Declare Global Variables” on page 19-75

“Define Global Data” on page 19-76

“Synchronizing Global Data with MATLAB” on page 19-77

“Define Constant Global Data” on page 19-80

“Limitations of Using Global Data” on page 19-83

### Workflow

To generate C/C++ code from MATLAB code that uses global data:

- 1 Declare the variables as global in your code.
- 2 Before using the global data, define and initialize it.

For more information, see “Define Global Data” on page 19-76.

- 3 Generate code from the MATLAB Coder project interface or using `codegen`.

If you use global data, you must also specify whether you want to synchronize this data between MATLAB and the generated MEX function. For more information, see “Synchronizing Global Data with MATLAB” on page 19-77.

### Declare Global Variables

When using global data, you must first declare the global variables in your MATLAB code. Consider the `use_globals` function that uses two global variables `AR` and `B`:

```
function y = use_globals(u)
%#codegen
% Turn off inlining to make
% generated code easier to read
coder.inline('never');
% Declare AR and B as global variables
global AR;
global B;
AR(1) = u + B(1);
```

```
y = AR * 2;
```

## Define Global Data

You can define global data either in the MATLAB global workspace, in a MATLAB Coder project, or at the command line. If you do not initialize global data in a project or at the command line, MATLAB Coder looks for the variable in the MATLAB global workspace. If the variable does not exist, MATLAB Coder generates an error.

### Defining Global Data in the MATLAB Global Workspace

To generate a MEX function for the `use_globals` function described in “Declare Global Variables” on page 19-75 using `codegen`:

- 1 In the MATLAB workspace, define and initialize the global data. At the MATLAB prompt, enter:

```
global AR B;  
AR = ones(4);  
B=[1 2 3];
```

- 2 Generate a MEX file.

```
codegen use_globals -args {0}  
% Use the -args option to specify that the input u  
% is a real, scalar, double  
% By default, codegen generates a MEX function,  
% use_globals_mex, in the current folder
```

### Defining Global Data in a MATLAB Coder Project

- 1 On the project **Overview** tab, click **Add global** and enter a name for the global variable.

By default, MATLAB Coder names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, etc.

- 2 After adding a global variable, before building the project, specify its type and initial value. For more information, see “Specifying Global Variable Type and Initial Value in a Project”.

---

**Note:** If you do not specify the type, you must create a variable with the same name in the global workspace.

---

## Defining Global Data at the Command Line

To define global data at the command line, use the `codegen -globals` option. For example, to compile the `use_globals` function described in “Declare Global Variables” on page 19-75, specify two global inputs `AR` and `B` at the command line. Use the `-args` option to specify that the input `u` is a real, scalar double. By default, `codegen` generates a MEX function, `use_globals_mex`, in the current folder.

```
codegen -globals {'AR',ones(4),'B',[1 2 3]} use_globals -args {0}
```

Alternatively, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`.

## Defining Variable-Size Global Data

To provide initial values for variable-size global data, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For example, to specify a global variable `g1` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
codegen foo -globals {'g1', {coder.typeof(0, [2 2],1),[1 1]}}
```

For a detailed explanation of the syntax, see `coder.typeof`.

## Synchronizing Global Data with MATLAB

### Why Synchronize Global Data?

The generated MEX function and MATLAB each have their own copies of global data. To make these copies consistent, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ. The level of interaction determines when to synchronize global data. For more information, see “When to Synchronize Global Data” on page 19-78.

When global data is constant, you cannot synchronize the global data with MATLAB. By default, the MEX function tests for consistency between the compile-time constant global values and the MATLAB values at function entry and after extrinsic function calls. If the MATLAB values differ from the compile-time constant global values, the MEX function ends with an error. For information about controlling when the MEX function tests for consistency between the compile-time constant global values and the MATLAB values, see “Consistency Between MATLAB and Constant Global Data” on page 19-82.

## When to Synchronize Global Data

By default, synchronization between the MEX function's global data and MATLAB occurs at MEX function entry and exit and for extrinsic calls. Use this synchronization method for maximum consistency between the MEX function and MATLAB.

To improve performance, you can:

- Select to synchronize only at MEX function entry and exit points.
- Disable synchronization when the global data does not interact.
- Choose whether to synchronize before and after each extrinsic call.

The following table summarizes which global data synchronization options to use. To learn how to set these options, see “How to Synchronize Global Data” on page 19-79.

### Global Data Synchronization Options

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Have maximum consistency when all extrinsic calls modify global data.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Default behavior.
Have maximum consistency when most extrinsic calls modify global data, but a few do not.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Use the <code>coder.extrinsic - sync:off</code> option to turn off synchronization for the extrinsic calls that do not change global data.
Have maximum consistency when most extrinsic calls do not modify global data, but a few do.	At MEX-function entry and exit	Yes. Use the <code>coder.extrinsic -sync:on</code> option to synchronize only the calls that modify global data.
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	At MEX-function entry and exit	No.



If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Communicate between generated MEX functions only. No interaction between MATLAB and MEX function global data.	Disabled	No.

## How to Synchronize Global Data

To control global data synchronization, set the global data synchronization mode and select whether to synchronize extrinsic functions. For guidelines on which options to use, see “When to Synchronize Global Data” on page 19-78.

You can control the global data synchronization mode from the project settings dialog box, the command line, or a MEX configuration dialog box. You control the synchronization of data with extrinsic functions using the `coder.extrinsic -sync:on` and `-sync:off` options.

### Controlling the Global Data Synchronization Mode in the Project Settings Dialog Box

- 1 On the MATLAB Coder project **Build** tab, verify that **Output type** is set to MEX Function and then click the **More settings** link.
- 2 On the **Project Settings** dialog box **Memory** tab, set **Global data synchronization mode** to At MEX-function entry and exit or Disabled, as applicable.

### Controlling the Global Data Synchronization Mode from the Command Line

- 1 In the MATLAB workspace, define the code generation configuration object. At the MATLAB command line, enter:

```
mexcfg = coder.config('mex');
```

- 2 At the MATLAB command line, set the `GlobalDataSyncMethod` property to `SyncAtEntryAndExits` or `NoSync`, as applicable. For example:

```
mexcfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
```

- 3 When compiling your code, use the `mexcfg` configuration object. For example, to generate a MEX function for function `f00` that has no inputs:

```
codegen -config mexcfg foo
```

### Controlling Synchronization for Extrinsic Function Calls

To control whether synchronization between MATLAB and MEX function global data occurs before and after you call an extrinsic function, use the `coder.extrinsic-sync:on` and `-sync:off` options.

By default, global data is:

- Synchronized before and after each extrinsic call, if the global data synchronization mode is `At MEX-function entry, exit and extrinsic calls`. If you are sure that certain extrinsic calls do not change global data, turn off synchronization for these calls using the `-sync:off` option. For example, if functions `foo1` and `foo2` do not change global data, turn off synchronization for these functions:

```
coder.extrinsic('-sync:off', 'foo1', 'foo2');
```

- Not synchronized, if the global data synchronization mode is `At MEX-function entry and exit`. If the code has a few extrinsic calls that change global data, turn on synchronization for these calls using the `-sync:on` option. For example, if functions `foo1` and `foo2` change global data, turn on synchronization for these functions:

```
coder.extrinsic('-sync:on', 'foo1', 'foo2');
```

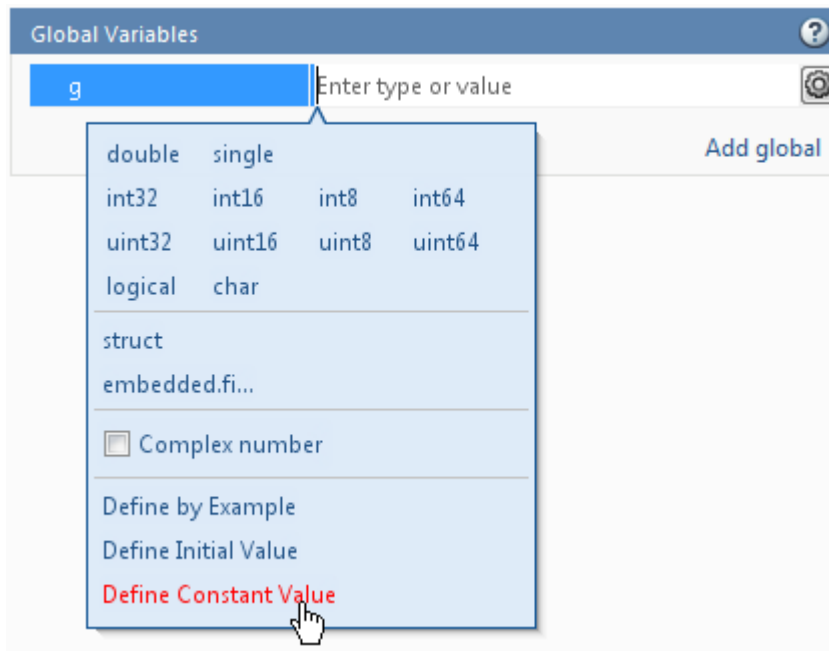
- Not synchronized, if the global data synchronization mode is `Disabled`. When synchronization is disabled, you cannot use the `-sync:on` option to control the synchronization for specific extrinsic calls.

## Define Constant Global Data

If you know that the value of a global variable does not change at run time, you can reduce overhead in the generated code by specifying that the global variable has a constant value. You cannot write to the constant global variable.

### Define Constant Global Data Using the Project Interface

- 1 On the project **Overview** tab, click the field to the right of the global variable.



- 2 Select Define Constant Value.
- 3 in the field to the right of the global variable, enter a MATLAB expression.

### Define Constant Global Data at the Command-Line Interface

To specify that a global variable is constant using the `codegen` command, use the `-globals` option with the `coder.Constant` class.

- 1 Define a configuration object for the code generation output type that you want. For example, define a configuration object for MEX code generation:

```
cfg = coder.config('mex');
```

- 2 Use `coder.Constant` to specify that a global variable has a constant value. For example, the following code specifies that the global variable `g` has initial value 4 and that global variable `gc` has the constant value 42.

```
global_values = {'g', 4, 'gc', coder.Constant(42)};
```

- 3 Generate the code using the `-globals` option. For example, generate code for `myfunction` specifying that the global variables are defined in the cell array `global_values`.

```
codegen -config cfg -globals global_values myfunction
```

### Consistency Between MATLAB and Constant Global Data

By default, the generated MEX function verifies that the values of constant global data in the MATLAB workspace are consistent with the compile-time values in the generated MEX. It tests for consistency at function entry and after calls to extrinsic functions. If the MEX function detects an inconsistency, it ends with an error. To control when the MEX function tests for consistency, use the `global` synchronization mode and the `coder.extrinsic` synchronization options.

The following table shows how the global data synchronization mode and the `coder.extrinsic` synchronization option setting determine when a MEX function verifies consistency between the compile-time constant global data values and MATLAB.

Global Data Synchronization Mode (Project)	GlobalDataSyncMethod (MEX Configuration Object)	Verify Consistency of Constant Global Values at MEX Function Entry	coder.extrinsic synchronization option	Verify Consistency of Constant Global Values After Extrinsic Function Call
At MEX-function entry, exit and extrinsic calls (default)	'SyncAlways'	yes	'sync:on' (default)	yes
			'sync:off'	no
At MEX-function entry and exit	'SyncAtEntryAndExits'	yes	'sync:on'	yes
			'sync:off' (default)	no
Disabled	'NoSync'	no	N/A	N/A

### Constant Global Data in a Code Generation Report

The code generation report provides the following information about a constant global variable:

- Type of Global on the **Variables** tab.
- Highlighted variable name in the **Function** pane.

See “Viewing Variables in Your MATLAB Code”.

## **Limitations of Using Global Data**

You cannot use global data with the `coder.cstructname` function.

## Generation of Traceable Code

### In this section...

- “About Code Traceability” on page 19-84
- “Generate Traceable Code” on page 19-84
- “Format of Traceability Tags” on page 19-87
- “Location of Comments in Generated Code” on page 19-87
- “Traceability Limitations” on page 19-91

### About Code Traceability

You can configure MATLAB Coder to generate C code and MEX functions that include the MATLAB source code as comments. Including this information in the generated code enables you to:

- Correlate the generated code with your source code.
- Understand how the generated code implements your algorithm.
- Evaluate the quality of the generated code.

In these automatically generated comments, a traceability tag immediately precedes each line of source code. This traceability tag provides details about the location of the source code. For more information, see “Format of Traceability Tags” on page 19-87.

For Embedded Coder projects, (requires an Embedded Coder license), you can also generate C/C++ code that includes the MATLAB function help text. The function help text is the first comment after the MATLAB function signature. It is displayed in the function banner of the generated code. The function help text provides information about the capabilities of the function and how to use it. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

### Generate Traceable Code

To generate more traceable code, include MATLAB source code as comments in the generated code from the **Project Settings** dialog box, the command line, or a MEX configuration dialog box.

### In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.

---

**Note:** MEX functions use a different set of configuration parameters than C/C++ libraries and executables. When you switch the output type between MEX Function and C/C++ Static Library, C/C++ Dynamic Library or C/C++ Executable, verify these settings. For more information, see “Changing Output Type” on page 16-42.

---

- 3 In the **Project Settings** dialog box, click the **Comments** tab.
- 4 On the **Code Appearance** tab, select **MATLAB source code as comments** and then close the dialog box.

### At the Command Line

#### For MEX Targets

Use the `MATLABSourceComments` option of the MEX configuration object. For example, to compile the file `foo.m` and include the source code as comments in the generated MEX function:

- 1 In the MATLAB workspace, define the MEX configuration object by issuing a constructor command:
 

```
mexcfg = coder.config('mex');
```
- 2 From the command line, enable the `MATLABSourceComments`:
 

```
mexcfg.MATLABSourceComments = true;
```
- 3 Using the `-config` option, pass the configuration object to `codegen`. For example, to generate a MEX function for a function `foo` that has no input parameters:
 

```
codegen -config mexcfg foo
```

#### For C/C++ Libraries

Use the `MATLABSourceComments` option of the code generation configuration object. For example, to compile the file `foo.m` and include the source code as comments in the generated code for a C static library:

- 1 Create a code generation configuration object and enable the `MATLABSourceComments` option. For example, to create a configuration object for a static library:

```
cfg = coder.config('lib');  
% If an Embedded Coder license is available,  
% cfg is a coder.EmbeddedCodeConfig object,  
% otherwise it's a coder.CodeConfig object  
cfg.MATLABSourceComments = true;
```

- 2 Using the `-config` option, pass the configuration object to `codegen`. For example, to generate a library for a function `foo` that has no input parameters:

```
codegen -config cfg foo
```

For Embedded Coder projects (requires an Embedded Coder license), you can also include the function help text in the generated code function banner using the `MATLABFcnDesc` option. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

### For C/C++ Executables

Use the `MATLABSourceComments` option of the code generation configuration object. For example, to compile the file `foo.m` and include the source code as comments in the generated code for a C executable:

- 1 Create a code generation configuration object and enable the `MATLABSourceComments` option. For example, to create a configuration object for a library:

```
cfg = coder.config('exe');  
% If an Embedded Coder license is available,  
% cfg is a coder.EmbeddedCodeConfig object,  
% otherwise it's a coder.CodeConfig object  
cfg.MATLABSourceComments = true;
```

- 2 Using the `-config` option, pass the configuration object to `codegen`. For example, to generate an executable for a function `foo` that has no input parameters:

```
codegen -config cfg main.c foo  
% You must specify a main file when generating an executable
```

For Embedded Coder projects, (requires an Embedded Coder license), you can also include the function help text in the function banner of the generated code using the



MATLABFcnDesc option. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

## Format of Traceability Tags

In the generated code, traceability tags appear immediately before the MATLAB source code in the comment. The format of the tag is:

```
<filename>:<line number>.
```

For example, the comment indicates that the code `x = r * cos(theta);` appears at line 4 in the source file `straightline.m`.

```
/* 'straightline:4' x = r * cos(theta); */
```

---

**Note:** With an Embedded Coder license, the traceability tags in the code generation report are hyperlinks to the MATLAB source code. For more information, see “Tracing Between Generated C Code and MATLAB Code”.

---

## Location of Comments in Generated Code

The auto-generated comments containing the source code and traceability tag appear in the generated code as follows.

### Straight-Line Source Code

In straight-line source code without `if`, `while`, `for` or `switch` statements, the comment containing the source code precedes the generated code that implements the source code statement. This comment appears after user comments that precede the generated code.

For example, in the following code, the user comment, `/* Convert polar to Cartesian */`, appears before the automatically generated comment containing the first line of source code, together with its traceability tag, `/* 'straightline:4' x = r * cos(theta); */`.

### MATLAB Code

```
function [x, y] = straightline(r,theta)
%#codegen
% Convert polar to Cartesian
```

```
x = r * cos(theta);  
y = r * sin(theta);
```

### Commented C Code

```
void straightline(double r, double theta, double *x, double *y)  
{  
    /* Convert polar to Cartesian */  
    /* 'straightline:4' x = r * cos(theta); */  
    *x = r * cos(theta);  
  
    /* 'straightline:5' y = r * sin(theta); */  
    *y = r * sin(theta);  
}
```

### If Statements

The comment for the `if` statement immediately precedes the code that implements the statement. This comment appears after user comments that precede the generated code. The comments for the `elseif` and `else` clauses appear immediately after the code that implements the clause, and before the code generated for statements in the clause.

### MATLAB Code

```
function y = ifstmt(u,v)  
%#codegen  
if u > v  
    y = v + 10;  
elseif u == v  
    y = u * 2;  
else  
    y = v - 10;  
end
```

### Commented C Code

```
double ifstmt(double u, double v)  
{  
    double y;  
  
    /* 'ifstmt:3' if u > v */  
    if (u > v) {  
        /* 'ifstmt:4' y = v + 10; */  
        y = v + 10.0;  
    } else if (u == v) {
```

```

    /* 'ifstmt:5' elseif u == v */
    /* 'ifstmt:6' y = u * 2; */
    y = u * 2.0;
} else {
    /* 'ifstmt:7' else */
    /* 'ifstmt:8' y = v - 10; */
    y = v - 10.0;
}

return y;
}

```

### For Statements

The comment for the `for` statement header immediately precedes the generated code that implements the header. This comment appears after user comments that precede the generated code.

#### MATLAB Code

```

function y = forstmt(u)
%#codegen
y = 0;
for i=1:u
    y = y + 1;
end

```

#### Commented C Code

```

double forstmt(double u)
{
    double y;
    int i;

    /* 'forstmt:3' y = 0; */
    y = 0.0;

    /* 'forstmt:4' for i=1:u */
    for (i = 0; i < (int)u; i++) {
        /* 'forstmt:5' y = y + 1; */
        y++;
    }

    return y;
}

```

## While Statements

The comment for the `while` statement header immediately precedes the generated code that implements the statement header. This comment appears after user comments that precede the generated code.

### MATLAB Code

```
function y = subfcn(y)
coder.inline('never');
while y < 100
    y = y + 1;
end
```

### Commented C Code

```
void subfcn(double *y)
{
    /* 'subfcn:2' coder.inline('never'); */
    /* 'subfcn:3' while y < 100 */
    while (*y < 100.0) {
        /* 'subfcn:4' y = y + 1; */
        (*y)++;
    }
}
```

## Switch Statements

The comment for the `switch` statement header immediately precedes the generated code that implements the statement header. This comment appears after user comments that precede the generated code. The comments for the `case` and `otherwise` clauses appear immediately after the generated code that implements the clause, and before the code generated for statements in the clause.

### MATLAB Code

```
function y = switchstmt(u)
%#codegen
y = 0;
switch u
    case 1
        y = y + 1;
    case 3
        y = y + 2;
    otherwise
```

```

        y = y - 1;
end

```

### Commented C Code

```

double switchstmt(double u)
{
    double y;

    /* 'switchstmt:3' y = 0; */
    /* 'switchstmt:4' switch u */
    switch ((int)u) {
        case 1:
            /* 'switchstmt:5' case 1 */
            /* 'switchstmt:6' y = y + 1; */
            y = 1.0;
            break;

        case 3:
            /* 'switchstmt:7' case 3 */
            /* 'switchstmt:8' y = y + 2; */
            y = 2.0;
            break;

        default:
            /* 'switchstmt:9' otherwise */
            /* 'switchstmt:10' y = y - 1; */
            y = -1.0;
            break;
    }

    return y;
}

```

## Traceability Limitations

For MATLAB Coder, there are traceability limitations:

- You cannot include MATLAB source code as comments for:
  - MathWorks toolbox functions
  - P-code
- The appearance or location of comments can vary depending on the following conditions:

- Even if the implementation code is eliminated, for example, due to constant folding, comments might still appear in the generated code.
- If a complete function or code block is eliminated, comments might be eliminated from the generated code.
- For certain optimizations, the comments might be separated from the generated code.
- Even if you do not choose to include source code comments in the generated code, the generated code includes legally required comments from the MATLAB source code.

## Generate Code for Enumerated Types

The basic workflow for generating code for enumerated types in MATLAB code is:

- 1 Define an enumerated data type that inherits from a base type that code generation supports. See “Enumerated Types Supported for Code Generation”.
- 2 Save the enumerated data type in a file on the MATLAB path.
- 3 Write a MATLAB function that uses the enumerated type.
- 4 Specify enumerated type inputs using the project or the command-line interface.
- 5 Generate code.

### See Also

- “Use Enumerated Types in LED Control Function”
- “Define Enumerated Data for Code Generation”
- “Specifying an Enumerated Type Input Parameter by Example”
- “Specifying an Enumerated Type Input Parameter by Type”

## Generate Code for Variable-Size Data

### In this section...

“Disable Support for Variable-Size Data” on page 19-94

“Control Dynamic Memory Allocation” on page 19-95

“Generating Code for MATLAB Functions with Variable-Size Data” on page 19-97

“Generate Code for a MATLAB Function That Expands a Vector in a Loop” on page 19-98

“Using Dynamic Memory Allocation for an "Atoms" Simulation” on page 19-104

Variable-size data is data whose size might change at run time. You can use MATLAB Coder to generate C/C++ code from MATLAB code that uses variable-size data. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. By default, for MEX and C/C++ code generation, support for variable-size data is enabled and dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to a configurable threshold.

### Disable Support for Variable-Size Data

By default, for MEX and C/C++ code generation, support for variable-size data is enabled. You modify variable sizing settings from the project settings dialog box, the command line, or using dialog boxes.

#### In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **General** tab.
- 4 On the **Memory** tab, select or clear **Enable variable-sizing**. Close the dialog box.

#### At the Command Line

- 1 Create a configuration object for code generation. For example, for a library:



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

- 2 Set the `EnableVariableSizing` option:

```
cfg.EnableVariableSizing = false;
```

- 3 Using the `-config` option, pass the configuration object to `codegen` :

```
codegen -config cfg foo
```

## Control Dynamic Memory Allocation

By default, dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to a configurable threshold. If you disable support for variable-size data (see “Disable Support for Variable-Size Data” on page 19-94), you also disable dynamic memory allocation. You can modify dynamic memory allocation settings from the project settings dialog box or the command line.

### In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **Memory** tab.
- 4 On the **Memory** tab, set **Dynamic memory allocation** to one of the following options:

Setting	Action
Never	Dynamic memory allocation is disabled. Variable-size data is allocated statically on the stack.
For all variable-sized arrays	Dynamic memory allocation is enabled for variable-size arrays. Variable-size data is allocated dynamically on the heap.
For arrays with maximum size at or above threshold	Dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to the <b>Dynamic memory allocation threshold</b> . Variable-size arrays whose size is less

Setting	Action
	than this threshold are allocated on the stack.

- 5 Optionally, if you set **Dynamic memory allocation** to For arrays with maximum size at or above threshold, configure **Dynamic memory allocation threshold** to fine tune memory allocation.
- 6 Close the dialog box.

### At the Command Line

- 1 Create a configuration object for code generation. For example, for a MEX function:  

```
mexcfg = coder.config('mex');
```
- 2 Set the `DynamicMemoryAllocation` option:

Setting	Action
<code>mexcfg.DynamicMemoryAllocation='Off';</code>	Dynamic memory allocation is disabled. Variable-size data is allocated statically on the stack.
<code>mexcfg.DynamicMemoryAllocation='AllVariableSizeArrays';</code>	Dynamic memory allocation is enabled for variable-size arrays. Variable-size data is allocated dynamically on the heap.
<code>mexcfg.DynamicMemoryAllocation='Threshold';</code>	Dynamic memory allocation is enabled for variable-size arrays whose size (in bytes) is greater than or equal to the value specified using the <b>Dynamic memory allocation threshold</b> parameter. Variable-size arrays whose size is less than this threshold are allocated on the stack.

- 3 Optionally, if you set `Dynamic memory allocation` to 'Threshold', configure `Dynamic memory allocation threshold` to fine tune memory allocation.

- 4 Using the `-config` option, pass the configuration object to `codegen`:

```
codegen -config mexcfg foo
```

## Generating Code for MATLAB Functions with Variable-Size Data

Here is a basic workflow that first generates MEX code for verifying the generated code and then generates standalone code after you are satisfied with the result of the prototype.

To work through these steps with a simple example, see “Generate Code for a MATLAB Function That Expands a Vector in a Loop” on page 19-98

- 1 In the MATLAB Editor, add the compilation directive `%#codegen` at the top of your function.

This directive:

- Indicates that you intend to generate code for the MATLAB algorithm
- Turns on checking in the MATLAB Code Analyzer to detect potential errors during code generation

- 2 Address issues detected by the Code Analyzer.

In some cases, the MATLAB Code Analyzer warns you when your code assigns data a fixed size but later grows the data, such as by assignment or concatenation in a loop. If that data is supposed to vary in size at run time, you can ignore these warnings.

- 3 Generate a MEX function using `codegen` to verify the generated code. Use the following command-line options:

- `-args {coder.typeof...}` if you have variable-size inputs
- `-report` to generate a code generation report

For example:

```
codegen -report foo -args {coder.typeof(0,[2 4],1)}
```

This command uses `coder.typeof` to specify one variable-size input for function `foo`. The first argument, `0`, indicates the input data type (`double`) and complexity (`real`). The second argument, `[2 4]`, indicates the size, a matrix with two dimensions. The third argument, `1`, indicates that the input is variable sized. The upper bound is 2 for the first dimension and 4 for the second dimension.

---

**Note:** During compilation, `codegen` detects variables and structure fields that change size after you define them, and reports these occurrences as errors. In addition, `codegen` performs a run-time check to generate errors when data exceeds upper bounds.

---

4 Fix size mismatch errors:

Cause:	How To Fix:	For More Information:
You try to change the size of data after its size has been locked.	Declare the data to be variable sized.	See “Diagnosing and Fixing Size Mismatch Errors”.

5 Fix upper bounds errors

Cause:	How To Fix:	For More Information:
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	See “Specifying Upper Bounds for Variable-Size Data” and “Diagnosing and Fixing Size Mismatch Errors”.
MATLAB attempts to compute an upper bound for unbounded variable-size data.	If the data is unbounded, enable dynamic memory allocation.	See “Control Dynamic Memory Allocation” on page 19-95.

6 Generate C/C++ code using the `codegen` function.

## Generate Code for a MATLAB Function That Expands a Vector in a Loop

- “About the MATLAB Function `unique_tol`” on page 19-99
- “Step 1: Add Compilation Directive for Code Generation” on page 19-99
- “Step 2: Address Issues Detected by the Code Analyzer” on page 19-99
- “Step 3: Generate MEX Code” on page 19-100
- “Step 4: Fix the Size Mismatch Error” on page 19-101
- “Step 5: Generate C Code” on page 19-103
- “Step 6: Change the Dynamic Memory Allocation Threshold” on page 19-104

## About the MATLAB Function `uniquetol`

This example uses the function `uniquetol`. This function returns in vector `B` a version of input vector `A`, where the elements are unique to within tolerance `tol` of each other. In vector `B`,  $\text{abs}(B(i) - B(j)) > \text{tol}$  for all `i` and `j`. Initially, assume input vector `A` can store up to 100 elements.

```
function B = uniquetol(A, tol)
A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

### Step 1: Add Compilation Directive for Code Generation

Add the `%#codegen` compilation directive at the top of the function:

```
function B = uniquetol(A, tol) %#codegen
A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

### Step 2: Address Issues Detected by the Code Analyzer

The Code Analyzer detects that variable `B` might change size in the `for`-loop. It issues this warning:

The variable 'B' appears to change size on every loop iteration. Consider preallocating for speed.

In this function, vector `B` should expand in size as it adds values from vector `A`. Therefore, you can ignore this warning.

### Step 3: Generate MEX Code

To generate MEX code, use the `codegen` function.

- 1 Generate a MEX function for `uniquetol`:

```
codegen -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

## What do these command-line options mean?

The `-args` option specifies the class, complexity, and size of each input to function `uniquetol`:

- The first argument, `coder.typeof`, defines a variable-size input. The expression `coder.typeof(0,[1 100],1)` defines input `A` as a real double vector with a fixed upper bound. Its first dimension is fixed at 1 and its second dimension can vary in size up to 100 elements.

For more information, see “Specify Variable-Size Inputs at the Command Line”.

- The second argument, `coder.typeof(0)`, defines input `tol` as a real double scalar.

The `-report` option instructs `codegen` to generate a code generation report, regardless of whether errors or warnings occur.

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

Executing this command generates a compiler error:

```
??? Size mismatch (size [1 x 1] ~= size [1 x 2]).  
The size to the left is the size  
of the left-hand side of the assignment.
```

- 2 Open the error report and select the **Variables** tab.

```

1 function B = uniquetol(A, tol) %#codegen
2 A = sort(A);
3 B = A(1);
4 k = 1;
5 for i = 2:length(A)
6     if abs(A(k) - A(i)) > tol
7         B = [B A(i)];
8         k = i;
9     end
10 end

```

Summary	All Messages (1)	Variables			
Order	Variable	Type	Size	Complex	Class
1	B	Output	1 x 1	No	double
2	A > 1	Input	1 x :100	No	double
3	A > 2	Local	1 x :?	No	double
4	tol	Input	1 x 1	No	double
5	k	Local	1 x 1	No	double
6	i	Local	1 x 1	No	double

The error indicates a size mismatch between the left-hand side and right-hand side of the assignment statement `B = [B A(i)];`. The assignment `B = A(1)` establishes the size of `B` as a fixed-size scalar (1 x 1). Therefore, the concatenation of `[B A(i)]` creates a 1 x 2 vector.

#### Step 4: Fix the Size Mismatch Error

To fix this error, declare `B` to be a variable-size vector.

- 1 Add this statement to the `uniquetol` function:

```
coder.varsize('B');
```

It should appear before `B` is used (read). For example:

```
function B = uniquetol(A, tol) %#codegen
```

```
A = sort(A);  
  
coder.varsize('B');  
  
B = A(1);  
k = 1;  
for i = 2:length(A)  
    if abs(A(k) - A(i)) > tol  
        B = [B A(i)];  
        k = i;  
    end  
end
```

The function `coder.varsize` declares every instance of `B` in `uniquetol` to be variable sized.

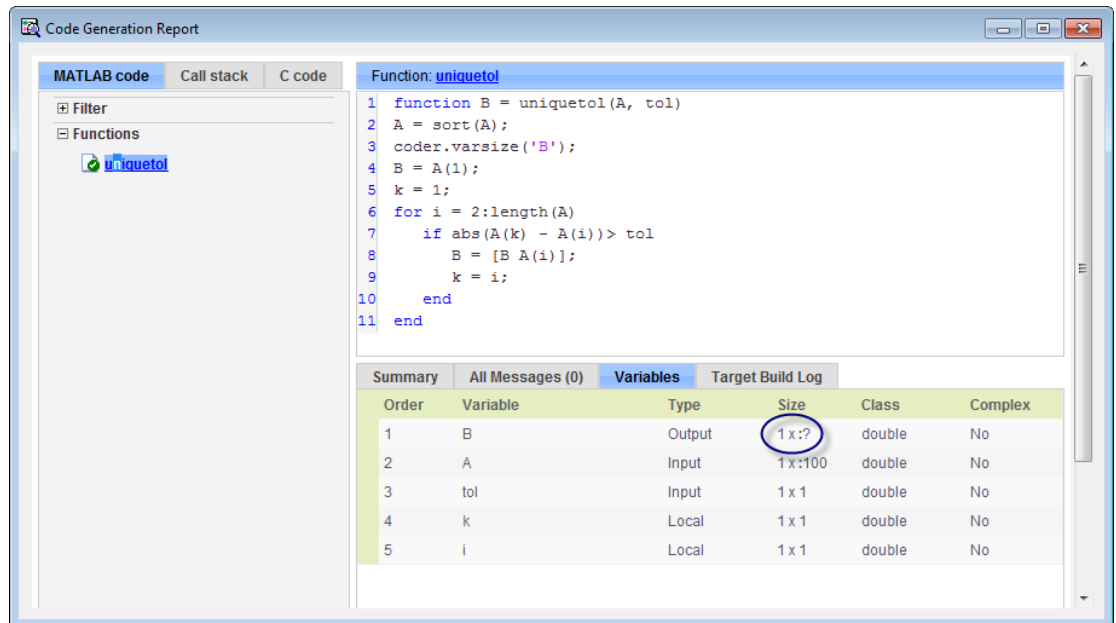
- 2 Generate code again using the same command:

```
codegen -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

In the current folder, `codegen` generates a MEX function for `uniquetol` and provides a link to the code generation report.

- 3 Click the **View report** link.
- 4 In the code generation report, select the **Variables** tab.





The size of variable B is 1 x:?, indicating that it is variable size with no upper bounds.

### Step 5: Generate C Code

Generate C code for variable-size inputs. By default, `codegen` allocates memory statically for data whose size is less than the dynamic memory allocation threshold of 64 kilobytes. If the size of the data is greater than or equal to the threshold or is unbounded, `codegen` allocates memory dynamically on the heap.

- 1 Create a configuration option for C library generation:

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

- 2 Issue this command:

```
codegen -config cfg -report uniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

`codegen` generates a static library in the default location, `codegen\lib\uniquetol` and provides a link to the code generation report.

- 3 Click the **View report** link.
- 4 In the code generation report, click the **C code** tab.
- 5 On the **C code** tab, click the link to `uniquetol.h`.

The function declaration is:

```
extern void uniquetol(const double A_data[100], const int A_size[2],...  
    double tol, emxArray_real_T *B);
```

`codegen` computes the size of `A` and, because its maximum size is less than the default dynamic memory allocation threshold of 64k bytes, allocates this memory statically. The generated code contains two pieces of information about `A`:

- `double A_data[100]`: the maximum size of input `A` (where 100 is the maximum size specified using `coder.typeof`).
- `int A_size[2]`: the actual size of the input.

Because `B` is variable size with unknown upper bounds, in the generated code, `codegen` represents `B` as `emxArray_real_T`. MATLAB provides utility functions for creating and interacting with `emxArrays` in your generated code. For more information, see “C Code Interface for Arrays”.

### Step 6: Change the Dynamic Memory Allocation Threshold

In this step, you reduce the dynamic memory allocation threshold and generate code for an input that exceeds this threshold.

- 1 Set the dynamic memory allocation threshold to 4 kilobytes and generate code where the size of input `A` exceeds this threshold.

```
cfg.DynamicMemoryAllocationThreshold=4096;  
codegen -config cfg -report uniquetol -args {coder.typeof(0,[1 10000],1),coder.typeof(0)}
```

- 2 View the generated code in the report. Because the maximum size of input `A` now exceeds the dynamic memory allocation threshold, `codegen` allocates `A` dynamically on the heap and represents `A` as `emxArray_real_T`.

```
extern void uniquetol(const emxArray_real_T *A, ...  
    double tol, emxArray_real_T *B);
```

## Using Dynamic Memory Allocation for an "Atoms" Simulation

This example shows how to generate code for a MATLAB algorithm that runs a simulation of bouncing "atoms" and returns the result after a number of iterations. There are no upper bounds on the number of atoms that the algorithm accepts, so this example takes advantage of dynamic memory allocation.

### Prerequisites

There are no prerequisites for this example.

### Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will contain only the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), change your working folder.

### Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_atoms');
```

### About the 'run\_atoms' Function

The run\_atoms.m function runs a simulation of bouncing atoms (also applying gravity and energy loss).

```
help run_atoms
```

```
atoms = run_atoms(atoms,n)
atoms = run_atoms(atoms,n,iter)
Where 'atoms' the initial and final state of atoms (can be empty)
      'n' is the number of atoms to simulate.
      'iter' is the number of iterations for the simulation
            (if omitted it is defaulted to 3000 iterations.)
```

### Set Up Code Generation Options

Create a code generation configuration object

```
cfg = coder.config;
% Enable dynamic memory allocation for variable size matrices.
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

### Set Up Example Inputs

Create a template structure 'Atom' to provide the compiler with the necessary information about input parameter types. An atom is a structure with four fields (x,y,vx,vy) specifying position and velocity in Cartesian coordinates.

```
atom = struct('x', 0, 'y', 0, 'vx', 0, 'vy', 0);
```

### Generate a MEX Function for Testing

Use the command 'codegen' with the following arguments:

'-args {coder.typeof(atom, [1 Inf]),0,0}' indicates that the first argument is a row vector of atoms where the number of columns is potentially infinite. The second and third arguments are scalar double values.

'-config cfg' enables dynamic memory allocation, defined by workspace variable cfg

```
codegen run_atoms -args {coder.typeof(atom, [1 Inf]),0,0} -config cfg -o run_atoms_mex
```

### Run the MEX Function

The MEX function simulates 10000 atoms in approximately 1000 iteration steps given an empty list of atoms. The return value is the state of all the atoms after simulation is complete.

```
atoms = run_atoms_mex([],10000,1000)
```

```
Iteration: 50  
Iteration: 100  
Iteration: 150  
Iteration: 200  
Iteration: 250  
Iteration: 300  
Iteration: 350  
Iteration: 400  
Iteration: 450  
Iteration: 500  
Iteration: 550  
Iteration: 600  
Iteration: 650  
Iteration: 700  
Iteration: 750  
Iteration: 800  
Iteration: 850  
Iteration: 900  
Iteration: 950
```

```
Iteration: 1000
Completed iterations: 1000

atoms =

1x10000 struct array with fields:

    x
    y
    vx
    vy
```

### Run the MEX Function Again

Continue the simulation with another 500 iteration steps

```
atoms = run_atoms_mex(atoms,10000,500)
```

```
Iteration: 50
Iteration: 100
Iteration: 150
Iteration: 200
Iteration: 250
Iteration: 300
Iteration: 350
Iteration: 400
Iteration: 450
Iteration: 500
Completed iterations: 500
```

```
atoms =

1x10000 struct array with fields:

    x
    y
    vx
    vy
```

### Generate a Standalone C Code Library

To generate a C library, create a standard configuration object for libraries:

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

Enable dynamic memory allocation

```
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

In MATLAB the default data type is double. However, integers are usually used in C code, so pass int32 integer example values to represent the number of atoms and iterations.

```
codegen run_atoms -args {coder.typeof(atom, [1 Inf]),int32(0),int32(0)} -config cfg
```

### Inspect Generated Code

When creating a library the code is generated in the folder codegen/lib/run\_atoms/ The code in this folder is self contained. To interface with the compiled C code you need only the generated header files and the library file.

```
dir codegen/lib/run_atoms
```

```
.          rt_nonfinite.obj          run_atoms_initialize.c
..         rtw_proj.tmw          run_atoms_initialize.h
buildInfo.mat  rtwtypes.h          run_atoms_initialize.obj
codeInfo.mat  run_atoms.c          run_atoms_ref.rsp
interface     run_atoms.h          run_atoms_rtw.bat
rtGetInf.c    run_atoms.lib        run_atoms_rtw.mk
rtGetInf.h    run_atoms.obj        run_atoms_terminate.c
rtGetInf.obj  run_atoms_emxAPI.c   run_atoms_terminate.h
rtGetNaN.c    run_atoms_emxAPI.h   run_atoms_terminate.obj
rtGetNaN.h    run_atoms_emxAPI.obj run_atoms_types.h
rtGetNaN.obj  run_atoms_emxutil.c
rt_nonfinite.c  run_atoms_emxutil.h
rt_nonfinite.h  run_atoms_emxutil.obj
```

### Write a C Main Function

Typically, the main function is platform-dependent code that performs rendering or some other processing. In this example, a pure ANSI-C function produces a file 'run\_atoms\_state.m' which (when run) contains the final state of the atom simulation.

```
type run_atoms_main.c
```

```
/* Include standard C libraries */
#include <stdio.h>
```

```

/* The interface to the main function we compiled. */
#include "codegen/lib/run_atoms/run_atoms.h"

/* The interface to EMX data structures. */
#include "codegen/lib/run_atoms/run_atoms_emxAPI.h"

int main(int argc, char **argv)
{
    int i;
    emxArray_Atom *atoms;

    /* Main arguments unused */
    (void) argc;
    (void) argv;

    /* Initially create an empty row vector of atoms (1 row, 0 columns) */
    atoms = emxCreate_Atom(1, 0);

    /* Call the function to simulate 10000 atoms in 1000 iteration steps */
    run_atoms(atoms, 10000, 1000);

    /* Call the function again to do another 500 iteration steps */
    run_atoms(atoms, 10000, 500);

    /* Print the result to standard output */
    for (i = 0; i < atoms->size[1]; i++) {
        printf("%f %f %f %f\n",
            atoms->data[i].x, atoms->data[i].y, atoms->data[i].vx, atoms->data[i].vy);
    }

    /* Free memory */
    emxDestroyArray_Atom(atoms);
    return(0);
}

```

### Create a Configuration Object for Executables

```

cfg = coder.config('exe');
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';

```

### Generate a Standalone Executable

You must pass the function (run\_atoms.m) as well as custom C code (run\_atoms\_main.c)  
The 'codegen' command automatically generates C code from the MATLAB code,

then calls the C compiler to bundle this generated code with the custom C code (run\_atoms\_main.c).

```
codegen run_atoms run_atoms_main.c -args {coder.typeof(atom, [1 Inf]),int32(0),int32(0)}
```

### Run the Executable

After simulation is complete, this produces the file 'atoms\_state.mat'. The MAT file is a 10000x4 matrix, where each row is the position and velocity of an atom (x, y, vx, vy) representing the current state of the whole system.

```
[~,atoms_data] = system(['.' filesep 'run_atoms']);  
fh = fopen('atoms_state.mat', 'w');  
fprintf(fh, '%s', atoms_data);  
fclose(fh);
```

### Fetch the State

Running the executable produced 'atoms\_state.mat'. Now, recreate the structure array from the saved matrix

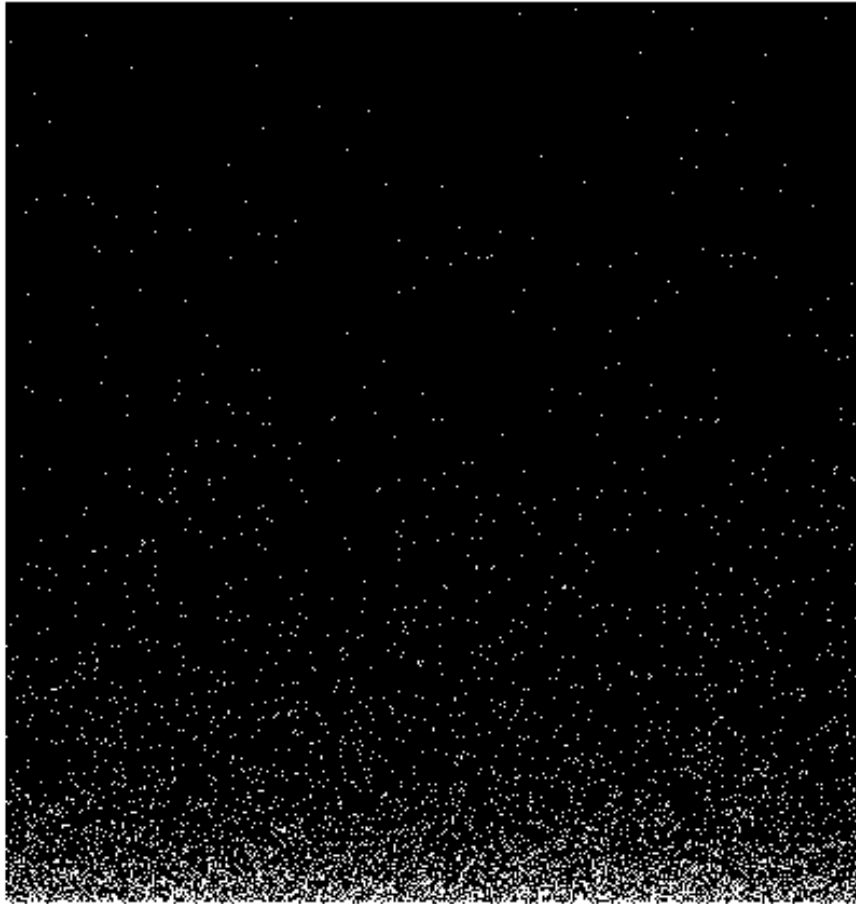
```
load atoms_state.mat -ascii  
clear atoms  
for i = 1:size(atoms_state,1)  
    atoms(1,i).x = atoms_state(i,1);  
    atoms(1,i).y = atoms_state(i,2);  
    atoms(1,i).vx = atoms_state(i,3);  
    atoms(1,i).vy = atoms_state(i,4);  
end
```

### Render the State

Call 'run\_atoms\_mex' with zero iterations to render only

```
run_atoms_mex(atoms, 10000, 0);
```





### **Clean Up**

Remove files and return to original folder

**Run Command: Cleanup**

```
if ispc
    delete run_atoms.exe
else
    delete run_atoms
end
delete atoms_state.mat
cleanup
```

## Code Generation for MATLAB Classes

You can generate code for MATLAB value and handle classes and user-defined System objects that inherit from a handle class. For more information, see “MATLAB Classes”.

## How MATLAB Coder Partitions Generated Code

### In this section...

“Partitioning Generated Files” on page 19-114

“How to Select the File Partitioning Method” on page 19-114

“Partitioning Generated Files with One C/C++ File Per MATLAB File” on page 19-115

“Generated Files and Locations” on page 19-120

“File Partitioning and Inlining” on page 19-122

### Partitioning Generated Files

By default, during code generation, MATLAB Coder partitions the code to match your MATLAB file structure. This one-to-one mapping lets you easily correlate your files generated in C/C++ with the compiled MATLAB code. MATLAB Coder cannot produce the same one-to-one correspondence for MATLAB functions that are inlined in generated code (see “File Partitioning and Inlining” on page 19-122).

Alternatively, you can select to generate all C/C++ functions into a single file. For more information, see “How to Select the File Partitioning Method” on page 19-114. This option facilitates integrating your code with existing embedded software.

### How to Select the File Partitioning Method

#### In the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On the **Build** tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **Code Appearance** tab.
- 4 On the **Code Appearance** tab, set the **Generated file partitioning method** to **Generate one file for each MATLAB file** or **Generate all functions into a single file**. Close the dialog box.

#### At the Command Line

Use the codegen configuration object `FilePartitionMethod` option. For example, to compile the function `foo` that has no inputs and generate one C/C++ file for each MATLAB function:

- 1 Create a MEX configuration object and set the `FilePartitionMethod` option:

```
mexcfg = coder.config('mex');
mexcfg.FilePartitionMethod = 'MapMFileToCFile';
```

- 2 Using the `-config` option, pass the configuration object to `codegen`:

```
codegen -config mexcfg -O disable:inline foo
% Disable inlining to generate one C/C++ file for each MATLAB function
```

## Partitioning Generated Files with One C/C++ File Per MATLAB File

By default, for MATLAB functions that are not inlined, MATLAB Coder generates one C/C++ file for each MATLAB file. In this case, MATLAB Coder partitions generated C/C++ code so that it corresponds to your MATLAB files.

### How MATLAB Coder Partitions Entry-Point MATLAB Functions

For each entry-point (top-level) MATLAB function, MATLAB Coder generates one C/C++ source, header, and object file with the same name as the MATLAB file.

For example, suppose you define a simple function `foo` that calls the function `identity`. The source file `foo.m` contains the following code:

```
function y = foo(u,v) %#codegen
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

Here is the code for `identity.m`:

```
function y = identity(u) %#codegen
y = u;
```

In the MATLAB Coder project interface, to generate a C static library for `foo.m`:

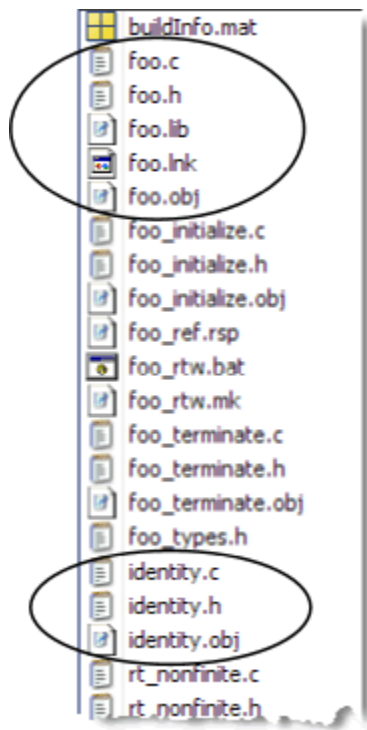
- 1 First, define the inputs `u` and `v`. For more information, see “Specifying Properties of Primary Function Inputs in a Project”.
- 2 In the MATLAB Coder project, click the **Build** tab.
- 3 On the **Build** tab:
  - a Set the **Output type** to **C/C++ Static Library**.

- b** Click the **More settings** link to view the project settings for the selected output type.
  - c** In the **Project Settings** dialog box, click the **All Settings** tab.
  - d** On this tab, under **Function Inlining**, set the **Inline threshold** parameter to 0.
- 4** Click **Build** to generate a library.

To generate a C static library for `foo.m` at the command line, enter:

```
codegen -config:lib -O disable:inline foo -args {0, 0}
% Use the -args option to specify that u and v are both
% real, scalar doubles
```

MATLAB Coder generates source, header, and object files for `foo` and `identity` in your output folder.



## How MATLAB Coder Partitions Local Functions

For each local function, MATLAB Coder generates code in the same C/C++ file as the calling function. For example, suppose you define a function `foo` that calls a local function `identity`:

```
function y = foo(u,v) %#codegen
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);

function y = identity(u)
y = u;
```

To generate a C++ library, before generating code, select a C++ compiler and set C++ as your target language. For example, at the command line:

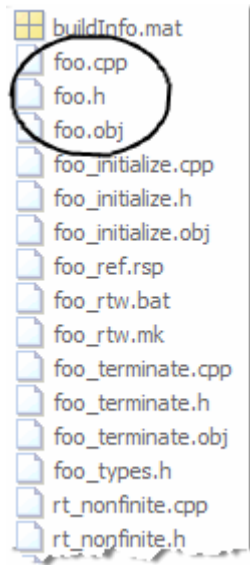
- 1 Select C++ as your target language:

```
cfg = coder.config('lib')
cfg.TargetLang='C++'
```

- 2 Generate the C++ library:

```
codegen -config cfg foo -args {0, 0}
% Use the -args option to specify that u and v are both
% real, scalar doubles
```

In the primary function `foo`, MATLAB Coder inlines the code for the `identity` local function.



---

**Note:** If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

---

Here is an excerpt of the generated code in `foo.cpp`:

```
...
/* Function Definitions */
double foo(double u, double v)
{
    return (double)(float)u + v;
}
...
```

### How MATLAB Coder Partitions Overloaded Functions

An overloaded function is a function that has multiple implementations to accommodate different classes of input. For each implementation (that is not inlined), MATLAB Coder generates a separate C/C++ file with a unique numeric suffix.



For example, suppose you define a simple function `multiply_defined`:

```
 %#codegen
function y = multiply_defined(u)

y = u+1;
```

You then add two more implementations of `multiply_defined`, one to handle inputs of type `single` (in an `@single` subfolder) and another for inputs of type `double` (in an `@double` subfolder).

To call each implementation, define the function `call_multiply_defined`:

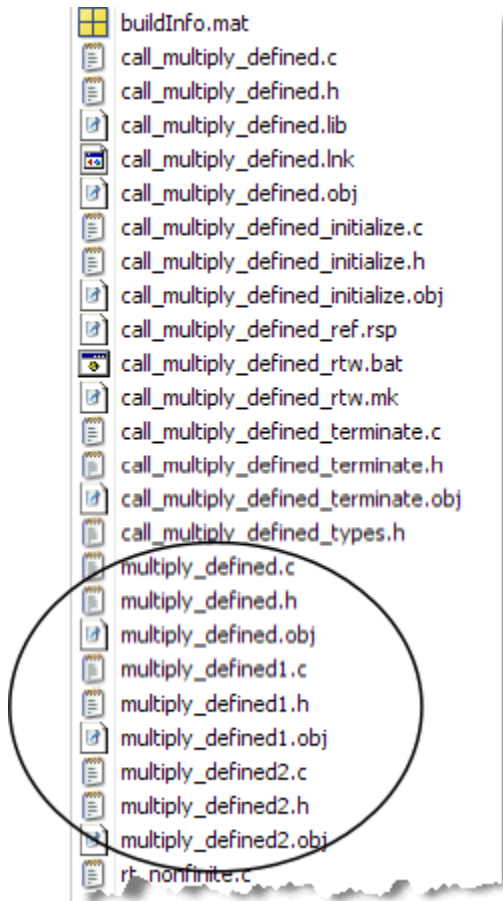
```
 %#codegen
function [y1,y2,y3] = call_multiply_defined

y1 = multiply_defined(int32(2));
y2 = multiply_defined(2);
y3 = multiply_defined(single(2));
```

Next, generate C code for the overloaded function `multiply_defined`. For example, at the MATLAB command line, enter:

```
codegen -o disable:inline -config:lib call_multiply_defined
```

MATLAB Coder generates C source, header, and object files for each implementation of `multiply_defined`, as highlighted. Use numeric suffixes to create unique file names.



## Generated Files and Locations

The types and locations of generated files depend on the target that you specify. For all targets, if errors or warnings occur during build or if you explicitly request a report, MATLAB Coder generates reports.

Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

### Generated Files for MEX Targets

By default, MATLAB Coder generates the following files for MEX function (mex) targets.

Type of Files	Location
Platform-specific MEX files	Current folder
MEX, and C/C++ source, header, and object files	codegen/mex/ <i>function_name</i>
HTML reports	codegen/mex/ <i>function_name</i> /html

### Generated Files for C/C++ Static Library Targets

By default, MATLAB Coder generates the following files for C/C++ static library targets.

Type of Files	Location
C/C++ source, library, header, and object files	codegen/lib/ <i>function_name</i>
HTML reports	codegen/lib/ <i>function_name</i> /html

### Generated Files for C/C++ Dynamic Library Targets

By default, MATLAB Coder generates the following files for C/C++ dynamic library targets.

Type of Files	Location
C/C++ source, library, header, and object files	codegen/dll/ <i>function_name</i>
HTML reports	codegen/dll/ <i>function_name</i> /html

### Generated Files for C/C++ Executable Targets

By default, MATLAB Coder generates the following files for C/C++ executable targets.

Type of Files	Location
C/C++ source, header, and object files	codegen/exe/ <i>function_name</i>
HTML reports	codegen/exe/ <i>function_name</i> /html

## Changing Names and Locations of Generated Files

### In the Project Settings Dialog Box

To change the...	Do this...
Output file name	On the <b>Build</b> tab, enter the name in the <b>Output file</b> field.
Output file location	<p>On the <b>Build</b> tab:</p> <ol style="list-style-type: none"> <li>1 Click the <b>More settings</b> link.</li> <li>2 In the Project Settings dialog box, click the <b>Paths</b> tab.</li> <li>3 On this tab, set <b>Build folder</b> to <b>Specified folder</b>.</li> </ol> <p>The <b>Build folder name</b> field appears.</p> <ol style="list-style-type: none"> <li>4 For this field, either browse to the output file location or enter the full path. The path must not contain spaces.</li> </ol> <hr/> <p><b>Note:</b> The output file location should not contain:</p> <ul style="list-style-type: none"> <li>• Spaces, as this can lead to code generation failures in certain operating system configurations.</li> <li>• Non 7-bit ASCII characters, such as Japanese characters.</li> </ul>

### At the Command Line

You can change the name and location of generated files by using the `codegen` options `-o` and `-d`.

## File Partitioning and Inlining

How MATLAB Coder partitions generated C/C++ code depends on whether you choose to generate one C/C++ file for each MATLAB file and whether you inline your MATLAB functions.

If you...	MATLAB Coder...
Generate all C/C++ functions into a single file and disable inlining	Generates a single C/C++ file without inlining functions.

<b>If you...</b>	<b>MATLAB Coder...</b>
Generate all C/C++ functions into a single file and enable inlining	Generates a single C/C++ file. Inlines functions whose sizes fall within the inlining threshold.
Generate one C/C++ file for each MATLAB file and disable inlining	Partitions generated C/C++ code to match MATLAB file structure. See “Partitioning Generated Files with One C/C++ File Per MATLAB File” on page 19-115.
Generate one C/C++ file for each MATLAB file and enable inlining	Places inlined functions in the same C/C++ file as the function into which they are inlined.  Even when you enable inlining, MATLAB Coder inlines only those functions whose sizes fall within the inlining threshold. For MATLAB functions that are not inlined, MATLAB Coder partitions the generated C/C++ code, as described.

### Tradeoffs Between File Partitioning and Inlining

Weighing file partitioning against inlining represents a trade-off between readability, efficiency, and ease of integrating your MATLAB code with existing embedded software.

<b>If You Generate...</b>	<b>Generated C/C++ Code</b>	<b>Advantages</b>	<b>Disadvantages</b>
All C/C++ functions into a single file	Does not match MATLAB file structure	Easier to integrate with existing embedded software	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and enable inlining	Does not exactly match MATLAB file structure	Program executes faster	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and disable inlining	Matches MATLAB file structure	Easy to map C/C++ code to original MATLAB file	Program runs less efficiently

### How Disabling Inlining Affects File Partitioning

Inlining is enabled by default. Therefore, to generate one C/C++ file for each top-level MATLAB function, you must:

- Select to generate one C/C++ file for each top-level MATLAB function. For more information, see “How to Select the File Partitioning Method” on page 19-114.
- Explicitly disable inlining, either globally or for individual MATLAB functions.

#### How to Disable Inlining Globally in the Project Settings Dialog Box

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On this tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the **Project Settings** dialog box, click the **All Settings** tab.
- 4 On this tab, under **Function Inlining** set the **Inlining threshold** to zero. Close the dialog box.

#### How to Disable Inlining Globally at the Command Line

To disable inlining of functions, use the `-O disable:inline` option with `codegen`. For example, to disable inlining and generate a MEX function for a function `foo` that has no inputs:

```
codegen -O disable:inline foo
```

For more information, see the description of `codegen`.

#### How to Disable Inlining for Individual Functions

To disable inlining for an individual MATLAB function, add the directive `coder.inline('never');` on a separate line in the source MATLAB file, after the function signature.

```
function y = foo(u,v) %#codegen
coder.inline('never');
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

`codegen` does not inline entry-point functions.

The `coder.inline` directive applies only to the function in which it appears. In this example, inlining is disabled for function `foo`, but not for `identity`, a top-level function defined in a separate MATLAB file and called by `foo`. To disable inlining for `identity`, add this directive after its function signature in the source file `identity.m`. For more information, see `coder.inline`.

For a more efficient way to disable inlining for both functions, see “How to Disable Inlining Globally at the Command Line” on page 19-124.

### **Correlating C/C++ Code with Inlined Functions**

To correlate the C/C++ code that you generate with the original inlined functions, add comments in the MATLAB code to identify the function. These comments will appear in the C/C++ code and help you map the generated code back to the original MATLAB functions.

### **Modifying the Inlining Threshold**

To change inlining behavior, adjust the inlining threshold parameter.

#### **Modifying the Inlining Threshold in the Project Settings Dialog Box**

On the **Project Settings** dialog box **All Settings** tab, under **Function Inlining**, set the value of the **Inline threshold** parameter.

#### **Modifying the Inlining Threshold at the Command Line**

Set the value of the `InlineThreshold` parameter of the configuration object. See `coder.MexCodeConfig`, `coder.CodeConfig`, `coder.EmbeddedCodeConfig`.

## Requirements for Signed Integer Representation

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.



## Customize the Post-Code-Generation Build Process

For certain applications, you might want to control aspects of the build process that occur after code generation but before compilation. For example, you might want to specify compiler or linker options. You can customize build processing that occurs after code generation using MATLAB Coder for MEX functions, C/C++ libraries and C/C++ executables.

You can customize your build using:

- The `coder.updateBuildInfo` function in your MATLAB code
- A post-code-generation command

### In this section...

“Customize Build Using `coder.updateBuildInfo`” on page 19-127

“Customize Build Using Post-Code-Generation Command” on page 19-127

“Build Information Object” on page 19-128

“Build Information Methods” on page 19-128

“Write Post-Code-Generation Command” on page 19-164

“Use Post-Code-Generation Command to Customize Build” on page 19-165

“Write and Use Post-Code-Generation Command at the Command Line” on page 19-165

### Customize Build Using `coder.updateBuildInfo`

To customize the post-code-generation build from your MATLAB code:

- 1 In your MATLAB code, call `coder.updateBuildInfo` to update the build information object. You specify a build information object method and the input arguments for the method. See `coder.updateBuildInfo` and “Build Information Methods”.
- 2 Generate code from your MATLAB code using the `codegen` command or from the project interface.

### Customize Build Using Post-Code-Generation Command

To customize your build using the post-code-generation command:

- 1 “Write Post-Code-Generation Command”. Typically, you use this command to get the project name and build information or to add data to the build information object.
- 2 “Use Post-Code-Generation Command to Customize Build”.

## Build Information Object

At the start of a build, the MATLAB Coder build process logs the following project, build option, and dependency information to a build information object, `RTW.BuildInfo`:

- Compiler options
- Preprocessor identifier definitions
- Linker options
- Source files and paths
- Include files and paths
- Precompiled external libraries

Use the “Build Information Methods” to access this information in the build information object. “Write Post-Code-Generation Command” on page 19-164 explains how to use the functions to control a post-code-generation build.

When code generation is complete, MATLAB Coder creates a `buildInfo.mat` file in the build folder.

## Build Information Methods

Use these methods to access or write data to the build information object. The syntax is:

```
buildInfo.method_name(input_arg1, ..., input_argn)
```

## addCompileFlags

- Purpose: Add compiler options to build information.
- Syntax: `addCompileFlags(buildinfo, options, groups)`  
*groups* is optional.
- Arguments:  
*buildinfo*

Build information stored in `RTW.BuildInfo`.

### *options*

A character array or cell array of character arrays that specifies the compiler options to be added to the build information. The function adds each option to the end of a compiler option vector. If you specify multiple options within a single character array, for example `'-Zi -Wall'`, the function adds the string to the vector as a single element. For example, if you add `'-Zi -Wall'` and then `'-O3'`, the vector consists of two elements, as shown below.

```
'-Zi -Wall'    '-O3'
```

### *groups* (optional)

A character array or cell array of character arrays that groups specified compiler options. You can use groups to

- Document the use of specific compiler options
- Retrieve or apply collections of compiler options

You can apply

- A single group name to one or more compiler options
- Multiple group names to collections of compiler options (available for nonmakefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to compiler options	Character array.
Apply different group names to compiler options	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>options</i> .

- Description:

The `addCompileFlags` function adds specified compiler options to the project's build information. MATLAB Coder stores the compiler options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *options* arguments, you can use an optional *groups* argument to group your options.

## addDefines

- Purpose: Add preprocessor macro definitions to build information.
- Syntax: `addDefines(buildinfo, macrodefs, groups)`

*groups* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*macrodefs*

A character array or cell array of character arrays that specifies the preprocessor macro definitions to be added to the object. The function adds each definition to the end of a compiler option vector. If you specify multiple definitions within a single character array, for example `'-DRT -DDEBUG'`, the function adds the string to the vector as a single element. For example, if you add `'-DPROTO -DDEBUG'` and then `'-DPRODUCTION'`, the vector consists of two elements, as shown below.

```
'-DPROTO -DDEBUG'    '-DPRODUCTION'
```

*groups* (optional)

A character array or cell array of character arrays that groups specified definitions. You can use groups to

- Document the use of specific macro definitions
- Retrieve or apply groups of macro definitions

You can apply

- A single group name to one or more macro definitions
- Multiple group names to collections of macro definitions (available for nonmakefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to macro definitions	Character array.

To...	Specify <i>groups</i> as a...
Apply different group names to macro definitions	Cell array of character arrays such that the number of group names matches the number elements you specify for <i>macrodefs</i> .

- Description:

The `addDefines` function adds specified preprocessor macro definitions to the project's build information. The MATLAB Coder software stores the definitions in a vector. The function adds definitions to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *macrodefs* arguments, you can use an optional *groups* argument to group your options.

## addIncludeFiles

- Purpose: Add include files to build information.
- Syntax: `addIncludeFiles(buildinfo, filenames, paths, groups)`

*paths* and *groups* are optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*filenames*

A character array or cell array of character arrays that specifies names of include files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are `'*. *'`, `'*.h'`, and `'*.h*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate include file entries that

- You specify as input

- Already exist in the include file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

*paths* (optional)

A character array or cell array of character arrays that specifies paths to the include files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

*groups* (optional)

A character array or cell array of character arrays that groups specified include files. You can use groups to

- Document the use of specific include files
- Retrieve or apply groups of include files

You can apply

- A single group name to an include file
- A single group name to multiple include files
- Multiple group names to collections of multiple include files

To...	Specify <i>groups</i> as a...
Apply one group name to include files	Character array.
Apply different group names to include files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

- Description:

The `addIncludeFiles` function adds specified include files to the project's build information. The MATLAB Coder software stores the include files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the include files it adds to the build information
Cell array of character arrays	Pairs each character array with a specified include file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string ( ' ') for *paths*.

## addIncludePaths

- Purpose: Add include paths to build information.
- Syntax: `addIncludePaths(buildinfo, paths, groups)`

*groups* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*paths*

A character array or cell array of character arrays that specifies include file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

The function removes duplicate include file entries that

- You specify as input
- Already exist in the include path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

*groups* (optional)

A character array or cell array of character arrays that groups specified include paths. You can use groups to

- Document the use of specific include paths
- Retrieve or apply groups of include paths

You can apply

- A single group name to an include path
- A single group name to multiple include paths
- Multiple group names to collections of multiple include paths

To...	Specify <i>groups</i> as a...
Apply one group name to include paths	Character array.
Apply different group names to include paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>paths</i> .

- Description:

The `addIncludePaths` function adds specified include paths to the project's build information. The MATLAB Coder software stores the include paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

#### If You Specify an Optional Argument as The Function...

a...

Character array	Applies the character array to the include paths it adds to the build information.
-----------------	--



**If You Specify an Optional Argument as The Function...****a...**

Cell array of character arrays

Pairs each character array with a specified include path. Thus, the length of the cell array must match the length of the cell array you specify for *paths*.

## addLinkFlags

- Purpose: Add link options to build information.
- Syntax: `addLinkFlags(buildinfo, options, groups)`

*groups* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*options*

A character array or cell array of character arrays that specifies the linker options to be added to the build information. The function adds each option to the end of a linker option vector. If you specify multiple options within a single character array, for example `'-MD -Gy'`, the function adds the string to the vector as a single element. For example, if you add `'-MD -Gy'` and then `'-T'`, the vector consists of two elements, as shown below.

```
'-MD -Gy'    '-T'
```

*groups* (optional)

A character array or cell array of character arrays that groups specified linker options. You can use groups to

- Document the use of specific linker options
- Retrieve or apply groups of linker options

You can apply

- A single group name to one or more linker options

- Multiple group names to collections of linker options (available for nonmakefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to linker options	Character array.
Apply different group names to linker options	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>options</i> .

- Description:

The `addLinkFlags` function adds specified linker options to the project's build information. The MATLAB Coder software stores the linker options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *options* arguments, you can use an optional *groups* argument to group your options.

## addLinkObjects

- Purpose: Add link objects to build information.
- Syntax: `addLinkObjects(buildinfo, linkobjs, paths, priority, precompiled, linkonly, groups)`

The arguments except *buildinfo*, *linkobjs*, and *paths* are optional. If you specify an optional argument, you must specify the optional arguments preceding it.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*linkobjs*

A character array or cell array of character arrays that specifies the filenames of linkable objects to be added to the build information. The function adds the filenames that you specify in the function call to a vector that stores the object filenames in priority order. If you specify multiple objects that have the same

priority (see *priority* below), the function adds them to the vector based on the order in which you specify the object filenames in the cell array.

The function removes duplicate link objects that

- You specify as input
- Already exist in the linkable object filename vector
- Have a path that matches the path of a matching linkable object filename

A duplicate entry consists of an exact match of a path string and corresponding linkable object filename.

### *paths*

A character array or cell array of character arrays that specifies paths to the linkable objects. If you specify a character array, the path string applies to all linkable objects.

### *priority* (optional)

A numeric value or vector of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority. The default priority is 1000.

### *precompiled* (optional)

The logical value **true** or **false** or a vector of logical values that indicates whether each specified link object is precompiled.

Specify **true** if the link object has been prebuilt for faster compiling and linking and exists in a specified location.

If precompiled is **false** (the default), the MATLAB Coder build process creates the link object in the build folder.

This argument is ignored if *linkonly* equals **true**.

### *linkonly* (optional)

The logical value **true** or **false** or a vector of logical values that indicates whether each specified link object is to be used only for linking.

Specify **true** if the MATLAB Coder build process should not build, nor generate rules in the makefile for building, the specified link object, but should include it when linking the final executable. For example, you can use this to incorporate

link objects for which source files are not available. If *linkonly* is true, the value of *precompiled* is ignored.

If *linkonly* is false (the default), rules for building the link objects are added to the makefile. In this case, the value of *precompiled* determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (true) or `START_EXPAND_LIBRARIES` (false). The software performs the expansion of the `START_PRECOMP_LIBRARIES` or `START_EXPAND_LIBRARIES` macro only if your code generation target uses the template makefile approach for building code.

#### *groups* (optional)

A character array or cell array of character arrays that groups specified link objects. You can use groups to

- Document the use of specific link objects
- Retrieve or apply groups of link objects

You can apply

- A single group name to a linkable object
- A single group name to multiple linkable objects
- Multiple group name to collections of multiple linkable objects

To...	Specify <i>groups</i> as a...
Apply one group name to link objects	Character array.
Apply different group names to link objects	Cell array of character arrays such that the number of group names matches the number elements you specify for <i>linkobjs</i> .

The default value of *groups* is { ' ' }.

- Description:

The `addLinkObjects` function adds specified link objects to the project's build information. The MATLAB Coder software stores the link objects in a vector in relative priority order. If multiple objects have the same priority or you do not specify priorities, the function adds the objects to the vector based on the order in which you specify them.

In addition to the required *buildinfo*, *linkobjs*, and *paths* arguments, you can specify the optional arguments *priority*, *precompiled*, *linkonly*, and *groups*. You can specify *paths* and *groups* as a character array or a cell array of character arrays.

If You Specify <i>paths</i> or <i>groups</i> as a...	The Function...
Character array	Applies the character array to the objects it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified object. Thus, the length of the cell array must match the length of the cell array you specify for <i>linkobjs</i> .

Similarly, you can specify *priority*, *precompiled*, and *linkonly* as a value or vector of values.

If You Specify <i>priority</i> , <i>precompiled</i> , or <i>linkonly</i> as a...	The Function...
Value	Applies the value to the objects it adds to the build information.
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for <i>linkobjs</i> .

If you choose to specify an optional argument, you must specify the optional arguments preceding it. For example, to specify that objects are precompiled using the *precompiled* argument, you must specify the *priority* argument that precedes *precompiled*. You could pass the default priority value 1000, as shown below.

```
addLinkObjects(myBuildInfo, 'test1', '/proj/lib/lib1', 1000, true);
```

## addNonBuildFiles

- Purpose: Add nonbuild-related files to build information.
- Syntax: `addNonBuildFiles(buildinfo, filenames, paths, groups)`

*paths* and *groups* are optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*filenames*

A character array or cell array of character arrays that specifies names of nonbuild-related files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are `'*. *'`, `'*.DLL'`, and `'*.D*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate nonbuild file entries that

- Already exist in the nonbuild file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

*paths* (optional)

A character array or cell array of character arrays that specifies paths to the nonbuild files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

*groups* (optional)

A character array or cell array of character arrays that groups specified nonbuild files. You can use groups to

- Document the use of specific nonbuild files
- Retrieve or apply groups of nonbuild files

You can apply

- A single group name to a nonbuild file

- A single group name to multiple nonbuild files
- Multiple group names to collections of multiple nonbuild files

To...	Specify <i>groups</i> as a...
Apply one group name to nonbuild files	Character array.
Apply different group names to nonbuild files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

- Description:

The `addNonBuildFiles` function adds specified nonbuild-related files, such as DLL files required for a final executable, or a README file, to the project's build information. The MATLAB Coder software stores the nonbuild files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the nonbuild files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified nonbuild file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string ( ' ') for *paths*.

## addSourceFiles

- Purpose: Add source files to build information.
- Syntax: `addSourceFiles(buildinfo, filenames, paths, groups)`

*paths* and *groups* are optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*filenames*

A character array or cell array of character arrays that specifies names of the source files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are `'*.*'`, `'*.c'`, and `'*.c*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

*paths* (optional)

A character array or cell array of character arrays that specifies paths to the source files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

*groups* (optional)

A character array or cell array of character arrays that groups specified source files. You can use groups to

- Document the use of specific source files
- Retrieve or apply groups of source files

You can apply



- A single group name to a source file
- A single group name to multiple source files
- Multiple group names to collections of multiple source files

To...	Specify <i>group</i> as a...
Apply one group name to source files	Character array.
Apply different group names to source files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

- Description:

The `addSourceFiles` function adds specified source files to the project's build information. The MATLAB Coder software stores the source files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the source files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string (' ') for *paths*.

## addSourcePaths

- Purpose: Add source paths to build information.
- Syntax: `addSourcePaths(buildinfo, paths, groups)`

*groups* is optional.

- Arguments:

*buildinfo*

Build information stored in RTW.`BuildInfo`.

*paths*

A character array or cell array of character arrays that specifies source file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

---

**Note:** The MATLAB Coder software does not check whether a specified path string is valid.

---

*groups* (optional)

A character array or cell array of character arrays that groups specified source paths. You can use groups to

- Document the use of specific source paths
- Retrieve or apply groups of source paths

You can apply

- A single group name to a source path
- A single group name to multiple source paths
- Multiple group names to collections of multiple source paths

To...	Specify <i>groups</i> as a...
Apply one group name to source paths	Character array.
Apply different group names to source paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>paths</i> .

- Description:

The `addSourcePaths` function adds specified source paths to the project's build information. The MATLAB Coder software stores the source paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument . You can specify *groups* as a character array or a cell array of character arrays.

#### If You Specify an Optional Argument as The Function... a...

Character array	Applies the character array to the source paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source path. Thus, the length of the character array or cell array must match the length of the cell array you specify for <i>paths</i> .

---

**Note:** The MATLAB Coder software does not check whether a specified path string is valid.

---

## addTMFTokens

- Purpose: Add template makefile (TMF) tokens that provide build-time information for makefile generation.
- Syntax: `addTMFTokens(buildinfo, tokennames, tokenvalues, groups)`

*groups* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*tokennames*

A character array or cell array of character arrays that specifies names of TMF tokens (for example, '|>CUSTOM\_OUTNAME<|') to be added to the build information. The function adds the token names to the end of a vector in the order that you specify them.

If you specify a token name that already exists in the vector, the first instance takes precedence and its value used for replacement.

*tokenvalues*

A character array or cell array of character arrays that specifies TMF token values corresponding to the previously-specified TMF token names. The function adds the token values to the end of a vector in the order that you specify them.

*groups* (optional)

A character array or cell array of character arrays that groups specified TMF tokens. You can use groups to

- Document the use of specific TMF tokens
- Retrieve or apply groups of TMF tokens

You can apply

- A single group name to a TMF token
- A single group name to multiple TMF tokens
- Multiple group names to collections of multiple TMF tokens

To...	Specify <i>groups</i> as a...
Apply one group name to TMF tokens	Character array.
Apply different group names to TMF tokens	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>tokennames</i> .

- Description:

Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled appropriately in the template makefile (TMF) for the target selected for your project. For example, if your post code generation command calls `addTMFTokens` to add a TMF token named `|>CUSTOM_OUTNAME<|` that specifies an output file name for the build, the TMF must act on the value of `|>CUSTOM_OUTNAME<|` to achieve the desired result.

The `addTMFTokens` function adds specified TMF token names and values to the project's build information. The MATLAB Coder software stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

In addition to the required *buildinfo*, *tokennames*, and *tokenvalues* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

#### If You Specify an Optional Argument The Function... as a...

Character array	Applies the character array to the TMF tokens it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified TMF token. Thus, the length of the cell array must match the length of the cell array you specify for <i>tokennames</i> .

## findIncludeFiles

- Purpose: Find and add include (header) files to build information.
- Syntax: `findIncludeFiles(buildinfo, extPatterns)`

*extPatterns* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*extPatterns* (optional)

A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern

- Must start with `*`.
- Can include a combination of alphanumeric and underscore (`_`) characters

The default pattern is `*.h`.

Examples of valid patterns include

```
*.h  
*.hpp  
*.x*
```

- Description:

The `findIncludeFiles` function

- Searches for include files, based on specified file name extension patterns, in the source and include paths recorded in a project's build information object
- Adds the files found, along with their full paths, to the build information object
- Deletes duplicate entries

## getCompileFlags

- Purpose: Get compiler options from build information.
- Syntax: `options = getCompileFlags(buildinfo, includeGroups, excludeGroups)`

*includeGroups* and *excludeGroups* are optional.

- Input arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of compiler flags you want the function to return.

*excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of compiler flags you do not want the function to return.

- Output arguments:

Compiler options stored in the project's build information.

- Description:

The `getCompileFlags` function returns compiler options stored in the project's build information. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of options the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ( ' ') for *includeGroups*.

## getDefines

- Purpose: Get preprocessor macro definitions from build information.
- Syntax: [*macrodefs*, *identifiers*, *values*] = `getDefines(buildinfo, includeGroups, excludeGroups)`

*includeGroups* and *excludeGroups* are optional.

- Input arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of macro definitions you want the function to return.

*excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of macro definitions you do not want the function to return.

- Output arguments:

Preprocessor macro definitions stored in the project's build information. The function returns the macro definitions in three vectors.

Vector	Description
<i>macrodefs</i>	Complete macro definitions with -D prefix
<i>identifiers</i>	Names of the macros
<i>values</i>	Values assigned to the macros (anything specified to the right of the first equals sign) ; the default is an empty string ( ' ' )

- Description:

The `getDefines` function returns preprocessor macro definitions stored in the project's build information. When the function returns a definition, it automatically

- Prepends a -D to the definition if the -D was not specified when the definition was added to the build information
- Changes a lowercase -d to -D

Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of definitions the function is to return.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ( ' ' ) for *includeGroups*.

## getFullFileList

- Purpose: Get All files from project's build information.
- Syntax: [*fPathNames*, *names*] = `getFullFileList(buildinfo, fcase)`

*fcase* is optional.

- Input arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*fcase* (optional)

The string 'source', 'include', or 'nonbuild'. If the argument is omitted, the function returns all the files from the build information object.



If You Specify...	The Function...
'source '	Returns source files from the build information object.
'include '	Returns include files from the build information object.
'nonbuild '	Returns nonbuild files from the build information object.

- Output arguments:

Fully-qualified file paths and file names for files stored in the project's build information.

---

**Note:** Usually it is unnecessary to resolve the path of every file in the project build information, because the makefile for the project build will resolve file locations based on source paths and rules. Therefore, `getFullFileList` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

---

- Description:

The `getFullFileList` function returns the fully-qualified paths and names of all files, or files of a selected type (source, include, or nonbuild), stored in the project's build information.

## getIncludeFiles

- Purpose: Get include files from build information.
- Syntax: `files = getIncludeFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`

`includeGroups` and `excludeGroups` are optional.

- Input arguments:

`buildinfo`

Build information stored in `RTW.BuildInfo`.

*concatenatePaths*

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Concatenates and returns each filename with its corresponding path.
<code>false</code>	Returns only filenames.

*replaceMatlabroot*

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
<code>false</code>	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

*includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of include files you want the function to return.

*excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of include files you do not want the function to return.

- Output arguments:

Names of include files stored in the project's build information.

- Description:

The `getIncludeFiles` function returns the names of include files stored in the project's build information. Use the *concatenatePaths* and *replaceMatlabroot* arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of include files the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ( ' ') for *includeGroups*.

## getIncludePaths

- Purpose: Get include paths from build information.
- Syntax: `files=getIncludePaths(buildinfo, replaceMatlabroot, includeGroups, excludeGroups)`

*includeGroups* and *excludeGroups* are optional.

- Input arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*replaceMatlabroot*

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
<code>false</code>	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

*includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of include paths you want the function to return.

*excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of include paths you do not want the function to return.

- Output arguments:

Paths of include files stored in the build information object.

- Description:

The `getIncludePaths` function returns the names of include file paths stored in the project's build information. Use the `replaceMatlabroot` argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of include file paths the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string ( ' ') for `includeGroups`.

## getLinkFlags

- Purpose: Get link options from build information.
- Syntax: `options=getLinkFlags(buildinfo, includeGroups, excludeGroups)`

`includeGroups` and `excludeGroups` are optional.

- Input arguments:

`buildinfo`

Build information stored in `RTW.BuildInfo`.

`includeGroups` (optional)

A character array or cell array that specifies groups of linker flags you want the function to return.

`excludeGroups` (optional)

A character array or cell array that specifies groups of linker flags you do not want the function to return. To exclude groups and not include specific groups, specify an empty cell array ( ' ') for `includeGroups`.

- Output arguments:

Linker options stored in the project's build information.

- Description:

The `getLinkFlags` function returns linker options stored in the project's build information. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of options the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ( ' ') for *includeGroups*.

## getNonBuildFiles

- Purpose: Get nonbuild-related files from build information.
- Syntax: `files=getNonBuildFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`

*includeGroups* and *excludeGroups* are optional.

- Input arguments:

### *buildinfo*

Build information stored in `RTW.BuildInfo`.

### *concatenatePaths*

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Concatenates and returns each filename with its corresponding path.
<code>false</code>	Returns only filenames.

### *replaceMatlabroot*

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
<code>false</code>	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

### *includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you want the function to return.

*excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you do not want the function to return.

- Output arguments:

Names of nonbuild files stored in the project's build information.

- Description:

The `getNonBuildFiles` function returns the names of nonbuild-related files, such as DLL files required for a final executable, or a README file, stored in the project's build information. Use the `concatenatePaths` and `replaceMatlabroot` arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of nonbuild files the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string ( ' ') for `includeGroups`.

## getSourceFiles

- Purpose: Get source files from project's build information.
- Syntax: `srcfiles=getSourceFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`

`includeGroups` and `excludeGroups` are optional.

- Input arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*concatenatePaths*

The logical value `true` or `false`.

If You Specify...	The Function...
<code>true</code>	Concatenates and returns each filename with its corresponding path.

If You Specify...	The Function...
false	Returns only filenames.

---

**Note:** Usually it is unnecessary to resolve the path of every file in the project build information, because the makefile for the project build will resolve file locations based on source paths and rules. Therefore, specifying true for `concatenatePaths` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

---

### *replaceMatlabroot*

The logical value true or false.

If You Specify...	The Function...
true	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
false	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

### *includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of source files you want the function to return.

### *excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of source files you do not want the function to return.

- Output arguments:

Names of source files stored in the project's build information.

- Description:

The `getSourceFiles` function returns the names of source files stored in the project's build information. Use the `concatenatePaths` and `replaceMatlabroot` arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional `includeGroups` and

*excludeGroups* arguments, you can selectively include or exclude groups of source files the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ( ' ') for *includeGroups*.

## getSourcePaths

- Purpose: Get source paths from build information.
- Syntax: `files=getSourcePaths(buildinfo, replaceMatlabroot, includeGroups, excludeGroups)`

*includeGroups* and *excludeGroups* are optional.

- Input arguments:

*buildinfo*

Build information stored in RTW.BuildInfo.

*replaceMatlabroot*

The logical value true or false.

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

*includeGroups* (optional)

A character array or cell array of character arrays that specifies groups of source paths you want the function to return.

*excludeGroups* (optional)

A character array or cell array of character arrays that specifies groups of source paths you do not want the function to return.

- Output arguments:

Paths of source files stored in the project's build information.

- Description:



The `getSourcePaths` function returns the names of source file paths stored in the project build information. Use the `replaceMatlabroot` argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of source file paths that the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string ( ' ') for `includeGroups`.

## packNGo

- Purpose: Package generated code in zip file for relocation.
- Syntax: `packNGo(buildinfo, propVals...)`

`propVals` is optional.

- Arguments:

`buildinfo`

Build information loaded from the build folder.

`propVals` (optional)

A cell array of property-value pairs that specify packaging details.

To...	Specify Property...	With Value...
Package generated code files in a zip file as a single, flat folder.	'packType'	'flat' (default)
Package generated code files hierarchically in a primary zip file.  The value of the 'nestedZipFiles' property determines whether the primary zip file contains secondary zip files or folders.	'packType'	'hierarchical'
Create a primary zip file that contains three secondary zip files:	'nestedZipFiles'	true (default)

To...	Specify Property...	With Value...
<ul style="list-style-type: none"> <li>• <code>mlrFiles.zip</code> — files in your <i>matlabroot</i> folder tree</li> <li>• <code>sDirFiles.zip</code> — files in and under your build folder</li> <li>• <code>otherFiles.zip</code> — required files not in the <i>matlabroot</i> or <i>start</i> folder trees</li> </ul> <p>Paths for files in the secondary zip files are relative to the root folder of the primary zip file.</p>		
Create a primary zip file that contains folders, for example, your build folder and <i>matlabroot</i> .	'nestedZipFiles'	false
Specify a file name for the primary zip file.	'fileName'	'string'  Default: 'untitled.zip' If you omit the .zip file extension, the function adds it.
Include only the minimal header files required to build the code in the zip file.	'minimalHeaders'	true (default)
Include header files found on the include path in the zip file.	'minimalHeaders'	false
Include the <code>html</code> folder for your code generation report.	'includeReport'	true (default is false)
Direct <code>packNGO</code> not to error out on parse errors.	'ignoreParseError'	true (default is false)
Direct <code>packNGO</code> not to error out if files are missing.	'ignoreFileMissing'	true (default is false)

- Description:

The `packNGO` function packages the following code files in a compressed zip file so you can relocate, unpack, and rebuild them in another development environment.

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- Nonbuild-related files (for example, `.dll` files required for a final executable file and `.txt` informational files)
- MAT-file that contains the build information object (`.mat` file)

Use this function to relocate files so that they can be recompiled for a specific target environment, or rebuilt in a development environment in which MATLAB is not installed.

By default, the `packNGo` function packages the files as a flat folder structure in a zip file, `foo.zip`. The zip file is located in the current working folder.

You can customize the output by specifying property name and value pairs as previously described.

After relocating the zip file, use a standard zip utility to unpack the compressed file.

- Limitations:

The following limitations apply to use of the `packNGo` function:

- The function operates on source files only, such as `*.c`, `*.cpp`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if they are not used.
- `packNGo` does not package the code generated for MEX targets.

- See Also:

- “Package Generated Code at the Command Line” on page 19-188
- “Package Code For Other Development Environments” on page 19-187

## updateFilePathsAndExtensions

- Purpose: Update files in project build information with missing paths and file extensions.
- Syntax: `updateFilePathsAndExtensions(buildinfo, extensions)`

*extensions* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*extensions* (optional)

A cell array of character arrays that specifies the extensions (file types) of files for which to search and include in the update processing. By default, the function searches for files with a `.c` extension. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For example, if you specify `{'.c' '.cpp'}`, and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.

- Description:

Using paths that already exist in a project's build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information need to be updated with a path or file extension. This function can be particularly useful for

- Maintaining build information for a toolchain that requires the use of file extensions
- Updating multiple customized instances of build information for a given project

## updateFilePathsAndExtensions

- Purpose: Update files in project build information with missing paths and file extensions
- Syntax: `updateFilePathsAndExtensions(buildinfo, extensions)`

*extensions* is optional.

- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*extensions* (optional)

A cell array of character arrays that specifies the extensions (file types) of files for which to search and include in the update processing. By default, the function searches for files with a `.c` extension. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For example, if you specify `{'.c' '.cpp'}`, and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.

- Description:

Using paths that already exist in a project's build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information need to be updated with a path or file extension. This function can be particularly useful for

- Maintaining build information for a toolchain that requires the use of file extensions
- Updating multiple customized instances of build information for a given project

## updateFileSeparator

- Purpose: Change file separator used in project's build information.
- Syntax: `updateFileSeparator(buildinfo, separator)`
- Arguments:

*buildinfo*

Build information stored in `RTW.BuildInfo`.

*separator*

A character array that specifies the file separator `\` (Windows) or `/` (UNIX<sup>®</sup>) to be applied to file path specifications.

- Description:

The `updateFileSeparator` function changes instances of the current file separator (`/` or `\`) in a project's build information to the specified file separator.

The default value for the file separator matches the value returned by the MATLAB command `filesep`. For makefile based builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile. If the `GenerateMakefile` parameter is set, the MATLAB Coder software overrides

the default separator and updates the build information after evaluating the `PostCodeGenCommand` configuration parameter.

## Write Post-Code-Generation Command

A post-code-generation command is a MATLAB file that typically calls functions that get data from or add data to the build information object. For example, you can access the project name in the variable `projectName` and the `RTW.BuildInfo` object in the variable `buildInfo`. You can write the command as a script or a function.

If You Write the Command as a...	Then the...
Script	Script can gain access to the project (top-level function) name and the build information directly.
Function	Function can receive the project name and the build information as arguments.

If your post-code-generation command calls user-defined functions, make sure that the functions are on the MATLAB path. If the build process cannot find a function that you use in your command, the process fails.

You can call combinations of build information functions to customize the post-code-generation build. See “Write and Use Post-Code-Generation Command at the Command Line” on page 19-165

### Write Post-Code-Generation Command as a Script

Set `PostCodeGenCommand` to the script name.

At the command line, enter:

```
cfg = coder.config('lib');
cfg.PostCodeGenCommand = 'ScriptName';
```

### Write Post-Code-Generation Command as a Function

Set `PostCodeGenCommand` to the function signature. When you define the command as a function, you can specify an arbitrary number of input arguments. If you want to access the project name, include `projectName` as an argument. If you want to modify or access build information, add `buildInfo` as an argument.

At the command line, enter:

```
cfg = coder.config('lib');  
cfg.PostCodeGenCommand = 'FunctionName(projectName, buildInfo)';
```

## Use Post-Code-Generation Command to Customize Build

After you have written a post-code-generation command, you must include this command in the build processing. You can include the command from the project settings dialog box or the command line.

### Use Post-Code-Generation Command in the Project Settings Dialog Box.

- 1 In the MATLAB Coder project, click the **Build** tab.
- 2 On this tab, click the **More settings** link to view the project settings for the selected output type.
- 3 In the Project Settings dialog box, click the **Custom Code** tab.
- 4 On this tab, set the **Post-code-generation command** parameter. Close the dialog box.

How you use the `PostCodeGenCommand` option depends on whether you write the command as a script or a function. See “Use Post-Code-Generation Command at the Command Line” on page 19-165 and “Use Post-Code-Generation Command in the Project Settings Dialog Box.” on page 19-165.

### Use Post-Code-Generation Command at the Command Line

Set the `PostCodeGenCommand` option for the code generation configuration object (`coder.MexCodeConfig`, `coder.CodeConfig` or `coder.EmbeddedCodeConfig`).

How you use the `PostCodeGenCommand` option depends on whether you write the command as a script or a function. See “Use Post-Code-Generation Command at the Command Line” on page 19-165 and “Use Post-Code-Generation Command in the Project Settings Dialog Box.” on page 19-165.

## Write and Use Post-Code-Generation Command at the Command Line

The following example shows how to write and use a post-code-generation command as a function. The `setbuildargs` function takes the build information object as a parameter, sets up link options, and adds them to the build information object.

- 1 Create a post-code-generation command as a function, `setbuildargs`, which takes the `buildInfo` object as a parameter:

```
function setbuildargs(buildInfo)
% The example being compiled requires pthread support.
% The -lpthread flag requests that the pthread library be included
% in the build
    linkFlags = {'-lpthread'};
    buildInfo.addLinkFlags(linkFlags);
```

- 2 Create a code generation configuration object. Set the `PostCodeGenCommand` option to `'setbuildargs(buildInfo)'` so that this command is included in the build processing:

```
cfg = coder.config('mex');
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';
```

- 3 Using the `-config` option, generate a MEX function passing the configuration object to `codegen`. For example, for the function `foo` that has no input parameters:

```
codegen -config cfg foo
```



# Code Generation Reports

**In this section...**

- “About Code Generation Reports” on page 19-167
- “Enable Code Generation Reports” on page 19-170
- “View Your MATLAB Code in a Report” on page 19-170
- “Viewing Call Stack Information” on page 19-172
- “View Generated C and C++ Code in a Report” on page 19-174
- “View the Build Summary Information” on page 19-174
- “View Errors and Warnings in a Report” on page 19-175
- “Viewing Variables in Your MATLAB Code” on page 19-176
- “View Target Build Information” on page 19-182
- “Keyboard Shortcuts for the Code Generation Report” on page 19-183
- “Report Limitations” on page 19-184

## About Code Generation Reports

At code-generation time, MATLAB Coder produces reports to help you debug your MATLAB code and to verify that your MATLAB code is suitable for code generation.

### Report Generation

If MATLAB Coder detects errors or warnings, the software automatically produces a code generation report. You can also use an option to request reports even if MATLAB Coder does not detect errors or warnings.

The report provides links to your MATLAB code and C/C++ code files. It also provides compile-time type information for the variables and expressions in your MATLAB code. This information simplifies finding sources of error messages and aids understanding of type propagation rules.

### Names and Locations of Code Generation Reports

MATLAB Coder produces code generation reports in the following locations. The top-level html file at each location is `index.html`.

- For MEX functions:

```
output_folder  
/mex/primary_function_name/html
```

- For C/C++ executables:

```
output_folder/exe/primary_function_name/html
```

- For C/C++ libraries:

```
output_folder/lib/primary_function_name/html
```

---

**Note:** The default output folder is `codegen`, but you can specify a different folder. For more information, see “Specify Output File Locations” on page 16-41.

---

## Opening Code Generation Reports

### Opening Code Generation Reports in the Project Interface

On the project **Build** tab, the **Build Results** pane provides information about the most recent build. If the code generation report is enabled or build errors occur, MATLAB Coder generates a report that provides detailed information about the most recent build and provides a link to the report.

To view the report, click the **View report** link. After a build completes, this report provides links to your MATLAB code and generated C/C++ files as well as compile-time type information for the variables in your MATLAB code. If build errors occur, it lists errors and warnings.

### Opening Code Generation Reports at the Command Line

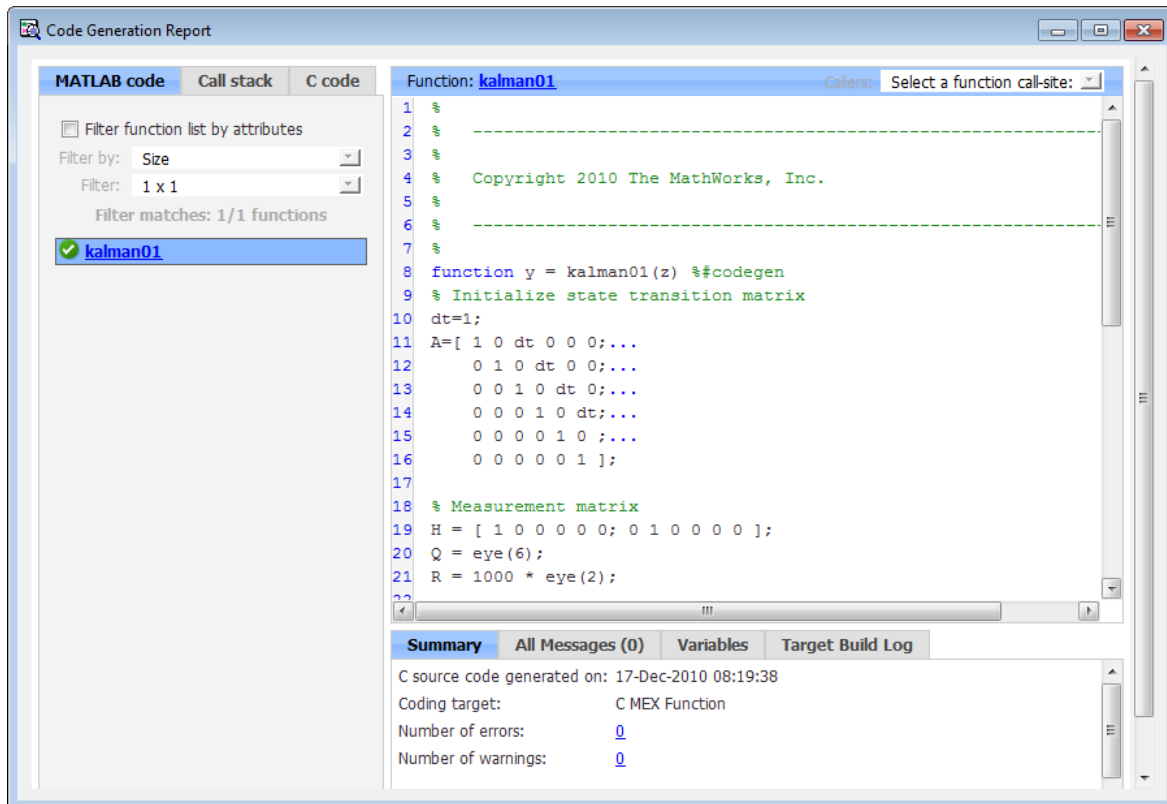
If you specify the `-launchreport` option, the code generation report opens automatically.

If MATLAB Coder did not detect build errors, to open the code generation report, in the MATLAB Command Window, click the **View report** link.

If MATLAB Coder detected build errors, to open the error report, in the MATLAB Command Window, click the **Open error report** link.

## Description of Code Generation Reports

When you generate code for MATLAB files from a MATLAB Coder project, or from the command line using the `codegen -report` option, MATLAB Coder generates a report. The following example shows a report for a completed build.



The report provides the following information, as applicable:

- MATLAB code information, including a list of functions and classes and their build status
- Call stack information, providing information on the nesting of function calls
- Links to generated C/C++ code files
- Summary of build results, including type of target and number of warnings or errors
- List of error and warning messages

- List of variables in your MATLAB code
- Target build log that records compilation and linking activities

## Enable Code Generation Reports

### How to Enable Code Generation Reports in the Project Settings Dialog Box

- 1 On the project **Build** tab, click the **More settings** link.
- 2 In the **Project Settings** dialog box, click the **Debugging** tab.
- 3 On the **Debugging** tab, select **Always create a code generation report**.

If you want the code generation or error report to automatically open when MATLAB Coder finishes building a project, select **Automatically launch a report if one is generated**.

### How to Enable Code Generation Reports at the Command Line

Use the `codegen` function `-report` option. To generate a standalone C/C++ static library and code generation report for a function `foo` that has no input parameters, at the MATLAB command line, enter:




```
codegen -config:lib -report foo
```

If you want the code generation or error report to automatically open, use the `-launchreport` option instead of the `-report` option.

## View Your MATLAB Code in a Report

To view your MATLAB code, click the **MATLAB code** tab. The code generation report displays the code for the function or class highlighted in the list on this tab.

The **MATLAB code** tab provides:

- A list of the MATLAB functions and classes that have been built. Depending on the build results, the report displays icons next to each function or class name:
  -  Errors in function or class.
  -  Warnings in function or class.
  -  Completed build, no errors or warnings.

- A filter control. You can use **Filter functions and methods** to sort your functions and methods by:
  - Size
  - Complexity
  - Class
- An optional highlight control to highlight potential data type issues in the generated C/C++ code. This option requires an Embedded Coder license. See “Highlight Potential Data Type Issues in a Report”.

### View Local Functions

The code generation report annotates the local function with the name of the parent function in the function list on the **MATLAB code** tab.

For example, if the MATLAB function `fcn1` contains the local function `local_fcn1`, and `fcn2` contains the local function `local_fcn2`, the report displays:

```
fcn1 > local_fcn1  
fcn2 > local_fcn2
```

### View Specializations

If your MATLAB function calls the same function with different types of inputs, the code generation report numbers each of these **specializations** in the function list on the **MATLAB code** tab.

For example, if the function `fcn` calls the function `subfcn` with different types of inputs:

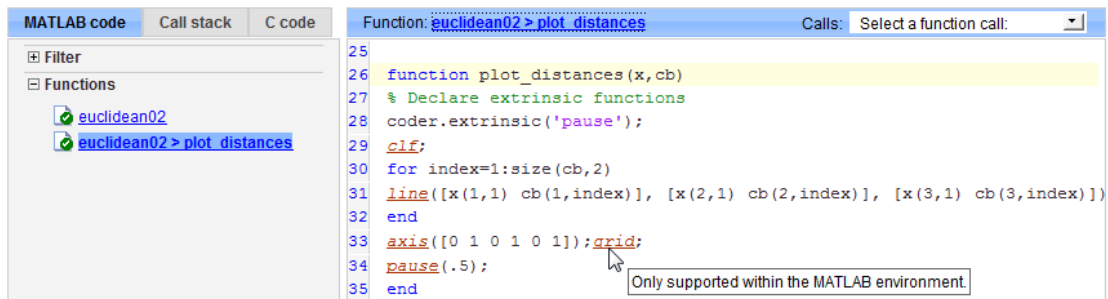
```
function y = fcn(u) %#codegen  
% Specializations  
y = y + subfcn(single(u));  
y = y + subfcn(double(u));
```

The code generation report numbers the specializations in the function list:

```
fcn > subfcn > 1  
fcn > subfcn > 2
```

### View Extrinsic Functions

The report highlights the extrinsic functions that are supported only within the MATLAB environment.



## Viewing Call Stack Information

The code generation report provides call stack information:

- On the **Call stack** tab.
- In the list of **Calls** at the top right of the report.

This list shows the calls from and to the function or method. If a function is called from more than one function, this list provides details of each call-site. Otherwise, the list is disabled.

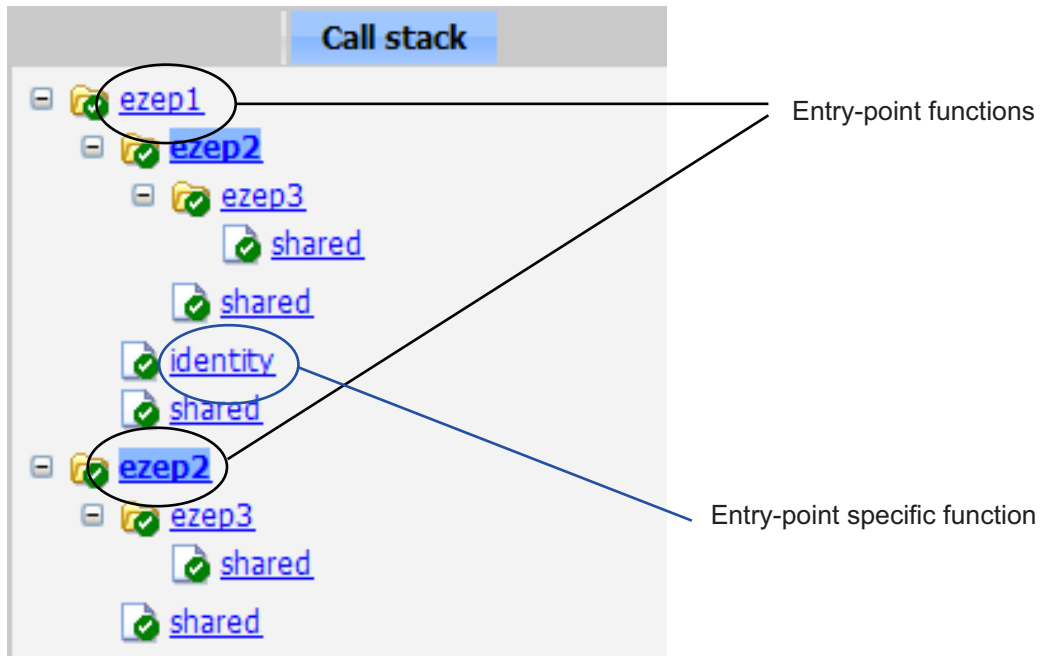
### Viewing Call Stack Information on the Call stack Tab

To view call stack information, click the **Call stack** tab.

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

For more than one entry-point function, the call stack displays a separate tree for each entry point. You can easily distinguish between shared and entry-point specific functions. If you click a shared function, the report highlights instances of this function. If you click an entry-point specific function, the report highlights only that instance.

For example, in the following call stack, `ezep1` and `ezep2` are entry-point functions. `identity` is an entry-point specific function, called only by `ezep1`. Functions `ezep3` and `shared` are shared functions.



### Viewing Call Sites in the Callers List

If a function or method is called from more than one function or method, or if the function or method calls other functions or methods, the **Calls** list provides details of each call site. To navigate between call sites, select a call site from the **Calls** list. If the function is not called more than once, this list is disabled.

If you select the entry-point function `ezep2` in the call stack, the **Calls** list displays the other call-site in `ezep1`.

Function: <code>ezep2 &gt; 1</code>	Callers: <input type="text" value="Select a function call-site:"/>
<pre> 1 function y = ezep2(u) %#codegen 2 y = shared(ezep3(u)); </pre>	<input type="text" value="Select a function call-site:"/> <input type="text" value="ezep1 at 2"/>

## View Generated C and C++ Code in a Report

To view a list of the generated C or C++ files, click the **C code** tab. The code generation report displays a list of the generated files in the **Target Source Files** pane. Click a file in the list to view the code in the code pane.

If you generate a MEX function, a list of support files that the code generation software uses appears in the **Interface Source Files** pane of the **C code** tab. By default, this list is collapsed.

### Trace Generated Code to MATLAB Source Code

You can configure `codegen` to generate C code that includes the MATLAB source code as comments. In these auto-generated comments, `codegen` precedes each line of source code with a traceability tag that provides details about the location of the source code. For more information, see “Generation of Traceable Code” on page 19-84.

For code generated with an Embedded Coder license, these traceability tags are hyperlinks. Click a tag to go the relevant line in the source code in the MATLAB editor.

### Navigate to C and C++ Code Source Files

When viewing C or C++ code in the code pane, at the top of the pane, click the blue link to the source file to open the associated source code file in the MATLAB editor.

### View Type Definitions

The code generation report provides links to the definitions of data types. When viewing C or C++ code in the code pane, click the blue link for a data type to see its definition.

### View Custom Code

The report displays custom code with color syntax highlighting. To learn what these colors mean and how to customize color settings, see “Colors in the MATLAB Editor”.

## View the Build Summary Information

To view a summary of the build results, including type of target and number of errors or warnings, click the **Summary** tab.



## View Errors and Warnings in a Report

MATLAB Coder automatically reports errors and warnings. If errors occur during the build, MATLAB Coder does not generate code. The report lists the messages in the order that MATLAB Coder detects them. It is a best practice to address the first message in the list, because often subsequent errors and warnings are related to the first message. If the build produces warnings, but no errors, MATLAB Coder does generate code.

The code generation report:

- Lists errors and warnings on the **All Messages** tab. The report lists these messages in chronological order.
- Highlights errors and warnings on the **MATLAB code** pane.
- Records compilation and linking errors and warnings on the **Target Build Log** tab. If the code generation software detects compilation warnings, you see a message on the **All Messages** tab. The code generation software detects compilation warnings only for MEX output or if you use a supported compiler for other types of output. For a list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/).

### View Errors and Warnings in the All Messages Tab

If errors or warnings occur during the build, click the **All Messages** tab to view a complete list of these messages. The code generation report marks messages:



Error



Warning

To locate the incorrect line of code for an error or warning in the list, click the message in the list. The code generation report highlights errors in the list and MATLAB code in red and highlights warnings in orange. Click the blue line number next to the incorrect line of code in the MATLAB code pane to go to the error in the source file.

---

**Note:** You can fix errors only in the source file.

---

### View Error and Warning Information in Your MATLAB Code

If errors or warnings occur during the build, the code generation report underlines them in your MATLAB code. The report underlines errors in red and warnings in orange. To

learn more about a particular error or warning, place your cursor over the underlined text.

### View Compilation and Linking Errors and Warnings

The code generation report highlights compilation and linking errors and warnings in red on the **Target Build Log** tab. For errors, the code generation report opens to the **Target Build Log** tab so that you can view the build log. For warnings, the report opens to the **All Messages** tab. A message instructs you to view the warnings on the **Target Build Log** tab.

### Viewing Variables in Your MATLAB Code

The report provides compile-time type information for the variables and expressions in your MATLAB code, including name, type, size, complexity, and class. It also provides type information for fixed-point data types, including word length and fraction length. You can use this type information to find sources of error messages and to understand type propagation rules.

You can view information about the variables in your MATLAB code:

- On the **Variables** tab, view the list.
- In your MATLAB code, place your cursor over the variable name.

In the MATLAB code, an orange variable name indicates a compile-time constant argument to an entry-point or a specialized function. The information for these variables includes the value. You can use this information to understand the function signature. You can also use this information to see when code generation created specializations of a function with different constant argument values.

### Viewing Variables in the Variables Tab

To view a list of the variables in your MATLAB function, click the **Variables** tab. The report displays a complete list of variables in the order that they appear in the function that you selected on the **MATLAB code** tab. Clicking a variable in the list highlights instances of that variable, and scrolls the MATLAB code pane so that you can view the first instance.

As applicable, the report provides the following information about each variable:

- Order

- Name
- Type
- Size
- Complexity
- Class
- DataTypeMode (DT mode) — for fixed-point data types only. For more information, see “Data Type and Scaling Properties”.
- Signed — sign information for built-in data types, signedness information for fixed-point data types.
- Word length (WL) — for fixed-point data types only.
- Fraction length (FL) — for fixed-point data types only.

---

**Note:** For more information on viewing fixed-point data types, see “Use Fixed-Point Code Generation Reports”.

---

It only displays a column if at least one variable in the code has information in that column. For example, if the code does not contain fixed-point data types, the report does not display the DT mode, WL or FL columns.

#### **Sorting Variables in the Variables Tab**

By default, the report lists the variables in the order that they appear in the selected function.

You can sort the variables by clicking the column headings on the **Variables** tab. To sort the variables by multiple columns, hold down the **Shift** key when clicking the column headings.

To restore the list to the original order, click the **Order** column heading.

#### **Viewing Structures on the Variables Tab**

You can expand structures listed on the **Variables** tab to display the field properties.

Summary		All Messages (0)	Variables				
Order	Variable	Type	Size	Complex	Class		
1	s	Output	1 x 1	-	struct		
1.1	s.a	Field	1 x 1	No	double		
1.2	s.b	Field	1 x 1	No	double		
2	a	Input	1 x 1	No	double		
3	b	Input	1 x 1	No	double		

If you sort the variables by type, size, complexity or class, a structure and its fields might not appear sequentially in the list. To restore the list to the original order, click the **Order** column heading.


#### Viewing Information About Variable-Size Arrays in the Variables Tab

For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon **:**.

In the following report, variable *A* is variable-size. Its maximum computed size is 1×100.

Summary		All Messages (0)	Variables				
Order	Variable	Type	Size	Complex	Class		
1	B	Output	1 x :100	No	double		
2	A	Input	1 x :100	No	double		
3	tol	Input	1 x 1	No	double		
4	k	Local	1 x 1	No	double		
5	i	Local	1 x 1	No	double		

If the code generation software cannot compute the maximum size of a variable-size array, the report displays the size as **:?**.

Summary		All Messages (1)	Variables				
Order	Type	Function	Line	Description			
1		emldemo_uniquetol	10	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [1 x :?]. This error may be reported due to a limitation of the underlying analysis.			

If you declare a variable-size array and then subsequently fix the dimensions of this array in the code, the report appends \* to the size of the variable. In the generated C code, this variable appears as a variable-size array, but the size of its dimensions do not change during execution.

Summary	All Messages (0)	Variables	Target Build Log		
Order	Variable	Type	Size	Complex	Class
1	y	Output	1 x 10 *	No	double

For more information on how to use the size information for variable-sized arrays, see “Variable-Size Data Definition for Code Generation”.

### Viewing Renamed Variables in the Variables Tab

If your MATLAB function reuses a variable with different size, type, or complexity, the code generation software attempts to create separate, uniquely named variables in the generated code. For more information, see “Reuse the Same Variable with Different Properties”. The report numbers the renamed variables in the list on the **Variables** tab. When you place your pointer over a renamed variable, the report highlights only the instances of this variable that share the same data type, size, and complexity.

For example, suppose your code uses the variable `t` in a for-loop to hold a scalar double, and reuses it outside the for-loop to hold a 5x5 matrix. The report displays two variables, `t>1` and `t>2` in the list on the **Variables** tab.

```

6 if all(all(u))
7     % First time t is used to hold a scalar double value
8     t = mean(mean(u)) / numel(u);
9     u = u - t;
10 end

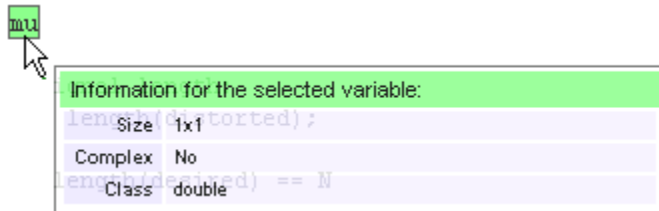
```

Summary	All Messages (0)	Variables			
Order	Variable	Type	Size	Complex	Class
1	u	Input	5 x 5	No	double
2	t > 1	Local	5 x 5	No	double
3	t > 2	Local	1 x 1	No	double

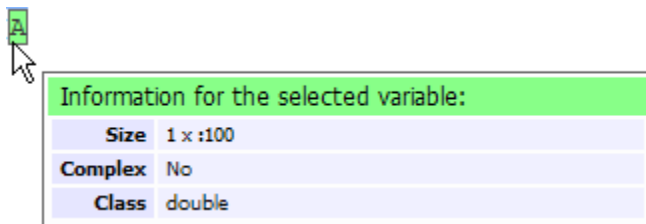
## Viewing Information About Variables and Expressions in Your MATLAB Function Code

To view information about a particular variable or expression in your MATLAB function code, on the MATLAB code pane, place your pointer over the variable name or expression. The report highlights variables and expressions in different colors:

**Green**, when the variable has data type information at this location in the code



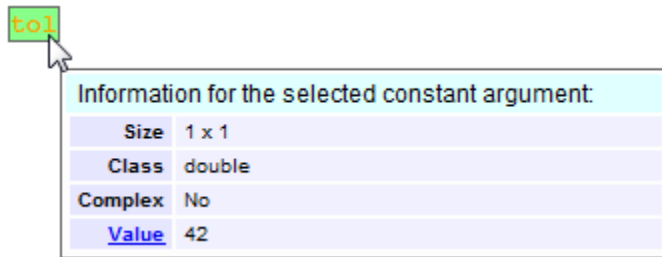
For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon `:`. Here the array `A` is variable-sized with a maximum computed size of `1 x 100`.



**Green with orange text**, when a constant argument has data type and value information

When the variable is a compile-time constant argument to an entry-point or a specialized function:

- The variable name is orange.
- The information for the variable includes the value.



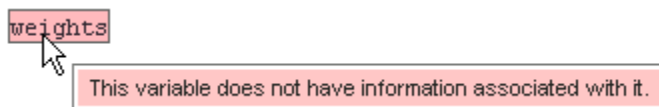
If you export the value as a variable to the base workspace, you can use the Workspace browser to view detailed information about the variable.

To export the value to the base workspace:

- 1 Click the **Value** link.
- 2 In the Export Constant Value dialog box, specify the **Variable name**.
- 3 Click **OK**.

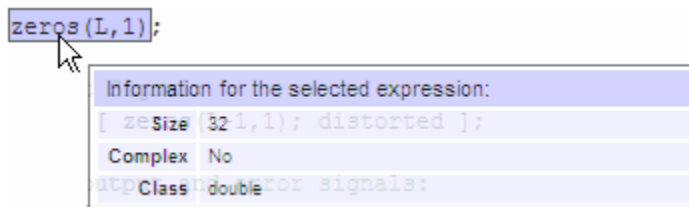
The variable and its value appear in the Workspace browser.

#### Pink, when the variable has no data type information



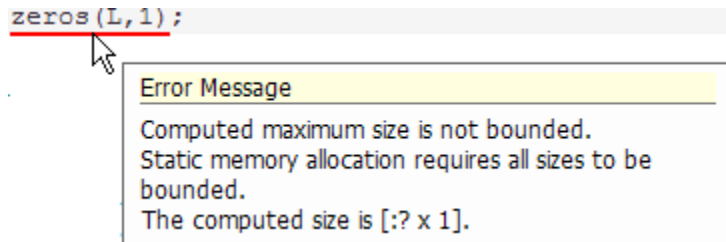
#### Purple, information about expressions

You can also view information about expressions in your MATLAB code. On the MATLAB code pane, place your pointer over an expression. The report highlights expressions in purple and provides more detailed information.



**Red, when there is error information for an expression**

If the code generation software cannot compute the maximum size of a variable-size array, the report underlines the variable name and provides error information.

**View Target Build Information**

If the build completes, MATLAB Coder provides target build information on the **Target Build Log** tab, including:

- Build folder

Clicking this link changes the MATLAB current folder to the build folder.

- Make wrapper

The batch file name that MATLAB Coder used for this build.

- Build log

If compilation or linking errors occur, the code generation report opens with the **Target Build Log** tab selected so that you can view the build log.



Summary	All Messages (12)	Variables	Target Build Log
Build Parameters			
Build directory	<a href="#">C:\Work\emcpri\mexfcn\warn</a>		
Make wrapper	warn_mex.bat		
Build Log			
<pre> 1  cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 2  warn_data.c 3  cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 4  warn.c 5  cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 6  warn_initialize.c 7  cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 8  warn_terminate.c 9  cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 10 morphfcn.c 11 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 12 warn_api.c 13 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 14 warn_mex.c 15 link /dll /export:mexFunction /LIBPATH:"\\.\mathworks\devel\JOBARC-1\Aslxtw\~\SNAPSH\2009_0-1\current\ma 16   Creating library warn.x and object warn.exp 17 mt -outputresource:"warn.mexw32;2" -manifest "warn.mexw32.manifest" 18 Microsoft (R) Manifest Tool version 5.2.3790.2014 19 Copyright (c) Microsoft Corporation 2005. 20 All rights reserved. </pre>			

## Keyboard Shortcuts for the Code Generation Report

You can use the following keyboard shortcuts to navigate between the different panes in the code generation report. Once you have selected a pane, use the **Tab** key to advance through data in that pane.

To select ...	Use...
<b>MATLAB code Tab</b>	Ctrl+m
<b>Call stack Tab</b>	Ctrl+k
<b>C code Tab</b>	Ctrl+c
<b>Code Pane</b>	Ctrl+w
<b>Summary Tab</b>	Ctrl+s
<b>All Messages Tab</b>	Ctrl+a
<b>Variables Tab</b>	Ctrl+v

To select ...	Use...
Target Build Log Tab	Ctrl+t

## Report Limitations

The report displays information about the variables and expressions in your MATLAB code with the following limitations:

## varargin and varargout

The report does not support `varargin` and `varargout` arrays.

## Loop Unrolling

The report does not display full information for unrolled loops. It displays data types of one arbitrary iteration.

## Dead Code

The report does not display information about dead code.

## Structures

The report does not provide complete information about structures.

- On the **MATLAB code** pane, the report does not provide information about all structure fields in the `struct()` constructor.
- On the **MATLAB code** pane, if a structure has a nonscalar field, and an expression accesses an element of this field, the report does not provide information for the field.

## Column Headings on Variables Tab

If you scroll through the list of variables, the report does not display the column headings on the **Variables** tab.

## Multiline Matrices

On the **MATLAB code** pane, the report does not support selection of multiline matrices. It supports only selection of individual lines at a time. For example, if you place your pointer over the following matrix, you cannot select the entire matrix.

```
out1 = [1 2 3;  
        4 5 6];
```

The report does support selection of single line matrices.

```
out1 = [1 2 3; 4 5 6];
```

## Troubleshooting

### Run-time Stack Overflow

If your C compiler reports a run-time stack overflow, set the value of the maximum stack usage parameter to be less than the available stack size. In a project, on the **Project Settings** dialog box **Memory** tab, set the **Stack usage max** parameter. For command-line configuration objects (`coder.MexCodeConfig`, `coder.CodeConfig`, `coder.EmbeddedCodeConfig`), set the `StackUsageMax` parameter.

## Package Code For Other Development Environments

### In this section...

“When to Package Code” on page 19-187

“Package Generated Code in a Project” on page 19-187

“Package Generated Code at the Command Line” on page 19-188

“Specify packNGo options” on page 19-189

### When to Package Code

If you need to relocate the generated code files to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB, use either the `packNGo` function at the command line or the `package` option in a project. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

See “Package Generated Code at the Command Line” on page 19-188 and “Package Generated Code in a Project” on page 19-187.

### Package Generated Code in a Project

This example shows how to package generated code into a zip file for relocation using the **Package** option in a MATLAB Coder project. By default, the zip file is created in the project folder.

- 1 In a local writable folder, for example `c:\work`, write a function `foo` that takes two double inputs.

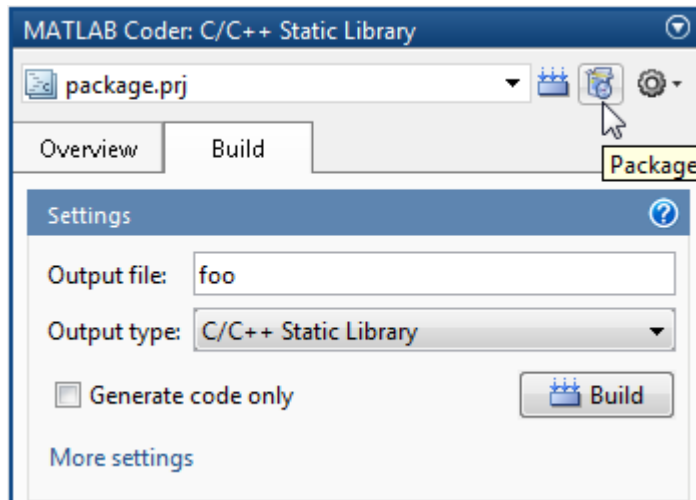
```
function y = foo(A,B)
    y = A + B;
end
```

- 2 In the same folder, create a new project.

```
coder -new package.prj
```

- 3 Add the file `foo` as an entry-point to the project.
- 4 Specify that inputs `A` and `B` are scalar doubles.
- 5 On the project **Build** tab, set **Output type** to build a static or dynamic library or executable. You cannot package the code generated for MEX targets.

- 6 At the top of the project, click **Package**.



Because you have not already built the project, MATLAB Coder builds the project.

- 7 When prompted, save the package file using the default path and file name. By default, MATLAB Coder derives the name of the package file from the project name and saves it in the current working folder. This zip file contains the C code and header files required for relocation. It does not contain compile flags, defines, or makefiles.
- 8 Inspect the contents of `package_pkg.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool you use you might be able to open and inspect the file without unpacking it.

You can now relocate the resulting zip file to the destination development environment and unpack the file.

## Package Generated Code at the Command Line

This example shows how to package generated code into a zip file for relocation using the `packNGO` function at the command line.

- 1 In a local writable folder, for example `c:\work`, write a function `foo` that takes two double inputs.

```
function y = foo(A,B)
    y = A + B;
end
```

- 2 Generate a static library for function `foo`. (`packNGo` does not package MEX function code.)

```
codegen -report -config:lib foo -args {0,0}
```

`codegen` generates code in the `c:\work\codegen\lib\foo` folder.

- 3 Load the `buildInfo` object.

```
load('c:\work\codegen\lib\foo\buildInfo.mat')
```

- 4 Create the zip file.

```
packNGo(buildInfo, 'fileName', 'foo.zip');
Alternatively, use the notation:
```

```
buildInfo.packNGo('fileName', 'foo.zip');
```

The `packNGo` function creates a zip file, `foo.zip`, in the current working folder. This zip file contains the C code and header files required for relocation. It does not contain compile flags, defines, or makefiles.

In this example, you specify only the file name. Optionally, you can specify additional packaging options. See “Specify `packNGo` options” on page 19-189.

- 5 Inspect the contents of `foo.zip` to verify that it is ready for relocation to the destination system. Depending on the zip tool you use you might be able to open and inspect the file without unpacking it. If you need to unpack the file and you packaged the generated code files as a hierarchical structure, you will need to unpack the primary and secondary zip files. When you unpack the secondary zip files, relative paths of the files are preserved.

You can now relocate the resulting zip file to the destination development environment and unpack the file.

## Specify `packNGo` options

You can specify options for the `packNGo` function.

To...	Specify...
Change the structure of the file packaging to hierarchical	<code>packNGo(buildInfo, {'packType' 'hierarchical'});</code>
Change the structure of the file packaging to hierarchical and rename the primary zip file	<code>packNGo(buildInfo, {'packType' 'hierarchical'... 'fileName' 'zippedsrcs'});</code>
Include all header files found on the include path (rather than the minimal header files required to build the code) in the zip file	<code>packNGo(buildInfo, {'minimalHeaders' false});</code>
Generate warnings for parse errors and missing files	<code>packNGo(buildInfo, {'ignoreParseError' true... 'ignoreFileMissing' true});</code>

For more information, see `packNGo` in “Build Information Methods”.

### Choose a Structure for the Zip File

Before you generate and package the files, decide whether you want the files to be packaged in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure. This approach is the simplest and might be the optimal choice.

If...	Then Use a...
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	Single, flat folder structure
The target development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code is dependent on the relative location of files	Hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of zip files. There is a primary zip file, which in turn contains the following secondary zip files:

- `mlrFiles.zip` — files in your *matlabroot* folder tree



- `sDirFiles.zip` — files in and under your build folder where you initiated code generation
- `otherFiles.zip` — required files not in the *matlabroot* or `start` folder trees

Paths for the secondary zip files are relative to the root folder of the primary zip file, maintaining the source development folder structure.



# Code Replacement for MATLAB Code

---

- “What Is Code Replacement?” on page 20-2
- “Code Replacement Libraries” on page 20-4
- “Code Replacement Terminology” on page 20-6
- “Code Replacement Limitations” on page 20-9
- “Replace Code Generated from MATLAB Code” on page 20-10
- “Choose a Code Replacement Library” on page 20-12

## What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU<sup>1</sup> gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel<sup>®</sup> Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)—GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

---

1. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

## **Related Examples**

- “Replace Code Generated from MATLAB Code”
- “Choose a Code Replacement Library”

## **More About**

- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

## Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

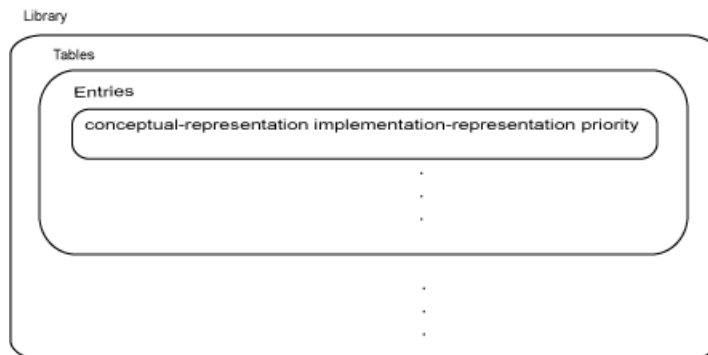


Table Entry Component	Description
Conceptual representation	<p>Identifies the table entry and contains match criteria for the code generator. Consists of:</p> <ul style="list-style-type: none"> <li>• Function name or a key. The function name identifies most functions. For operators and some functions, a string called a key identifies a function or operator. For example, function name 'COS' and operator key 'RTW_OP_ADD'.</li> <li>• Conceptual arguments that observe code generator naming ('y1', 'u1', 'u2', ...), with corresponding I/O types (output or input) and data types.</li> <li>• Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.</li> </ul>

Table Entry Component	Description
Implementation representation	Specifies replacement code. Consists of: <ul style="list-style-type: none"> <li>• Function name. For example, 'cos_dbl' or 'u8_add_u8_u8')</li> <li>• Implementation arguments, with corresponding I/O types (output or input) and data types.</li> <li>• Parameters that provide additional implementation details, such as header and source file names and paths of build resources.</li> </ul>
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

## Related Examples

- “Replace Code Generated from MATLAB Code”
- “Choose a Code Replacement Library”

## More About

- “What Is Code Replacement?”
- “Code Replacement Terminology”

## Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions ('y1',



Term	Definition
	'u1', 'u2', ...) and data types familiar to the code generator.
Conceptual representation	<p>Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of:</p> <ul style="list-style-type: none"> <li>• Function or operator name or key</li> <li>• Conceptual arguments with type, dimension, and complexity specification for inputs and output</li> <li>• Attributes, such as an algorithm and fixed-point saturation and rounding modes</li> </ul>
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.
Implementation representation	<p>Specifies C or C++ replacement function prototype. Consists of:</p> <ul style="list-style-type: none"> <li>• Function name (for example, 'cos_db1' or 'u8_add_u8_u8')</li> <li>• Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output</li> <li>• Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags</li> </ul>
Key	A string that identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key RTW_OP_ADD identifies the addition operator.

<b>Term</b>	<b>Definition</b>
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

### **More About**

- “What Is Code Replacement?”
- “Code Replacement Libraries”

## Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

### Related Examples

- “Replace Code Generated from MATLAB Code”

### More About

- “Code Replacement Libraries”

## Replace Code Generated from MATLAB Code

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

### Prepare for Code Replacement

- 1 Make sure required software is installed. You need MATLAB, MATLAB Coder, and a C compiler. Some code replacement libraries available in your development environment might also require Embedded Coder.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

- 2 Identify an existing or create a new MATLAB function for which you want the code generator to replace code.

### Choose a Code Replacement Library

If you are not sure which library to use, explore available libraries.

### Configure Code Generator To Use Code Replacement Library

- 1 Configure the code generator to apply a code replacement library during code generation for the MATLAB function.
  - In a project, on the **Hardware** tab, set the **Code Replacement Library** parameter.
  - In a code configuration object, set the `CodeReplacementLibrary` parameter.
- 2 Configure the code generator to produce code only so you can verify your code replacements before building an executable.
  - In a project, on the **Build** tab, set the **Generate code only** parameter.
  - In a code configuration object, set the `GenCodeOnly` parameter.

### Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can

help you verify code replacements. For more information, see “Review and Test Code Replacements” in the Embedded Coder documentation.

### **Generate Replacement Code**

Generate C/C++ code from the MATLAB code and, if you configured the code generator accordingly, a code generation report. For example, on the **Build** tab, click **Build**. Or, at the command prompt, enter:

```
codegen -report myFunction -args {5} -config cfg
```

The code generator produces the code and displays the report.

### **Verify Code Replacements**

Verify code replacements by examining the generated code. Code replacement might behave differently than you expect, for example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

### **Related Examples**

- “Choose a Code Replacement Library”
- “Build Setting Configuration”

### **More About**

- “What Is Code Replacement?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

### **External Web Sites**

- Supported Compilers

## Choose a Code Replacement Library

### In this section...

“About Choosing a Code Replacement Library” on page 20-12

“Explore Available Code Replacement Libraries” on page 20-12

“Explore Code Replacement Library Contents” on page 20-12

### About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

- 1 Explore available libraries. Identify one that best meets your application needs.
  - Consider the lists of application code replacement requirements and libraries that MathWorks provides in “What Is Code Replacement?”.
  - See “Explore Available Code Replacement Libraries”.
- 2 Explore the contents of the library. See “Explore Code Replacement Library Contents”.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

### Explore Available Code Replacement Libraries

You can select the code replacement library to use for code generation in a project, on the **Hardware** tab, by setting the **Code Replacement Library** parameter. Alternatively, in a code configuration object, set the `CodeReplacementLibrary` parameter.

### Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

- 1 At the command prompt, type `RTW.viewTf1`.

```
>> RTW.viewTf1
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> RTW.viewTf1('GNU C99 extensions')
```

- 2** In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
- 3** In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
- 4** In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.Tf1COperationEntryGenerator` or `RTW.Tf1COperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See [Code Replacement Viewer](#) for details on what the viewer displays.

## Related Examples

- [“Replace Code Generated from MATLAB Code”](#)

## More About

- [“Code Replacement Libraries”](#)
- [“Code Replacement Terminology”](#)
- [“Code Replacement Limitations”](#)





# Custom Toolchain Registration

---

- “Custom Toolchain Registration” on page 21-2
- “About `coder.make.ToolchainInfo`” on page 21-6
- “Create and Edit Toolchain Definition File” on page 21-8
- “Toolchain Definition File with Commentary” on page 21-10
- “Create and Validate `ToolchainInfo` Object” on page 21-16
- “Register the Custom Toolchain” on page 21-17
- “Use the Custom Toolchain” on page 21-19
- “Troubleshooting Custom Toolchain Validation” on page 21-20
- “Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers” on page 21-24

## Custom Toolchain Registration

In this section...
“What Is a Custom Toolchain?” on page 21-2
“What Is a Factory Toolchain?” on page 21-2
“What is a Toolchain Definition?” on page 21-3
“Key Terms” on page 21-4
“Typical Workflow” on page 21-4

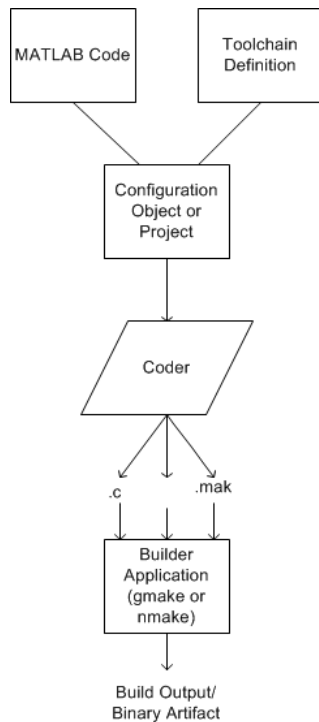
### What Is a Custom Toolchain?

You can add support for software build tools to MATLAB Coder software. For example, you can add support for a third-party compiler/linker/archiver (toolchain) to your MATLAB Coder software. This can be useful when a toolchain has support and optimizations for a specific type of processor or hardware. When you add support for toolchains, we call these *custom toolchains*.

### What Is a Factory Toolchain?

MATLAB Coder software includes support for specific toolchains. We call these *factory toolchains* to distinguish them from custom toolchains. If you install factory toolchains on your host computer, MATLAB Coder can automatically detect and use them. Support for factory toolchains depends on the host operating system. A complete list of supported toolchains is available at <http://www.mathworks.com/support/compilers/>.

## What is a Toolchain Definition?



A *toolchain definition* provides MATLAB Coder software with information about the software build tools, such as the compiler, linker, archiver. MATLAB Coder software uses this information, along with a configuration object or project, to build the generated code. This approach can be used when generating static libraries, dynamic libraries, and executables. MEX-file generation uses a different approach. To specify which compiler to use for MEX-function generation, see “Setting Up the C or C++ Compiler”.

MATLAB Coder software comes with a set of registered *factory toolchain* definitions. You can create and register *custom toolchain* definitions. You can customize and manage toolchain definitions. You can share custom toolchain definitions with others running MATLAB Coder software.

If you install toolchain software for one of the factory toolchains, MATLAB Coder can automatically detect and use the toolchain software. For more information about

factory toolchains in MATLAB Coder software, see <http://www.mathworks.com/support/compilers/>

## Key Terms

It is helpful to understand the following concepts:

- *Toolchain* — Software that can create a binary executable and libraries from source code. A toolchain can include:
  - *Prebuild tools* that set up the environment
  - *Build tools*, such as an Assembler, C compiler, C++ Compiler, Linker, Archiver, that build a binary executable from source code
  - *Postbuild tools* that download and run the executable on the hardware, and clean up the environment
- *Custom toolchain* — A toolchain that you define and register for use by MATLAB Coder software
- *Factory toolchains* — Toolchains that are predefined and registered in MATLAB Coder software
- *Registered toolchains* — The sum of custom and factory toolchain definitions registered in MATLAB Coder software
- *ToolchainInfo object* — An instance of the `coder.make.ToolchainInfo` class that contains a toolchain definition. You save the `ToolchainInfo` object as a MAT file, register the file with MATLAB Coder. Then you can configure MATLAB Coder to load the `ToolchainInfo` object during code generation.
- *Toolchain definition file* — A MATLAB file that defines the properties of a toolchain. You use this file to create a `ToolchainInfo` object.

---

**Note:** This documentation also refers to the `ToolchainInfo` object as a `coder.make.ToolchainInfo` object.

---

## Typical Workflow

The typical workflow for creating and using a custom toolchain definition is:

- 1 “Create and Edit Toolchain Definition File”



## About `coder.make.ToolchainInfo`

The following properties in `coder.make.ToolchainInfo` represent your custom toolchain:

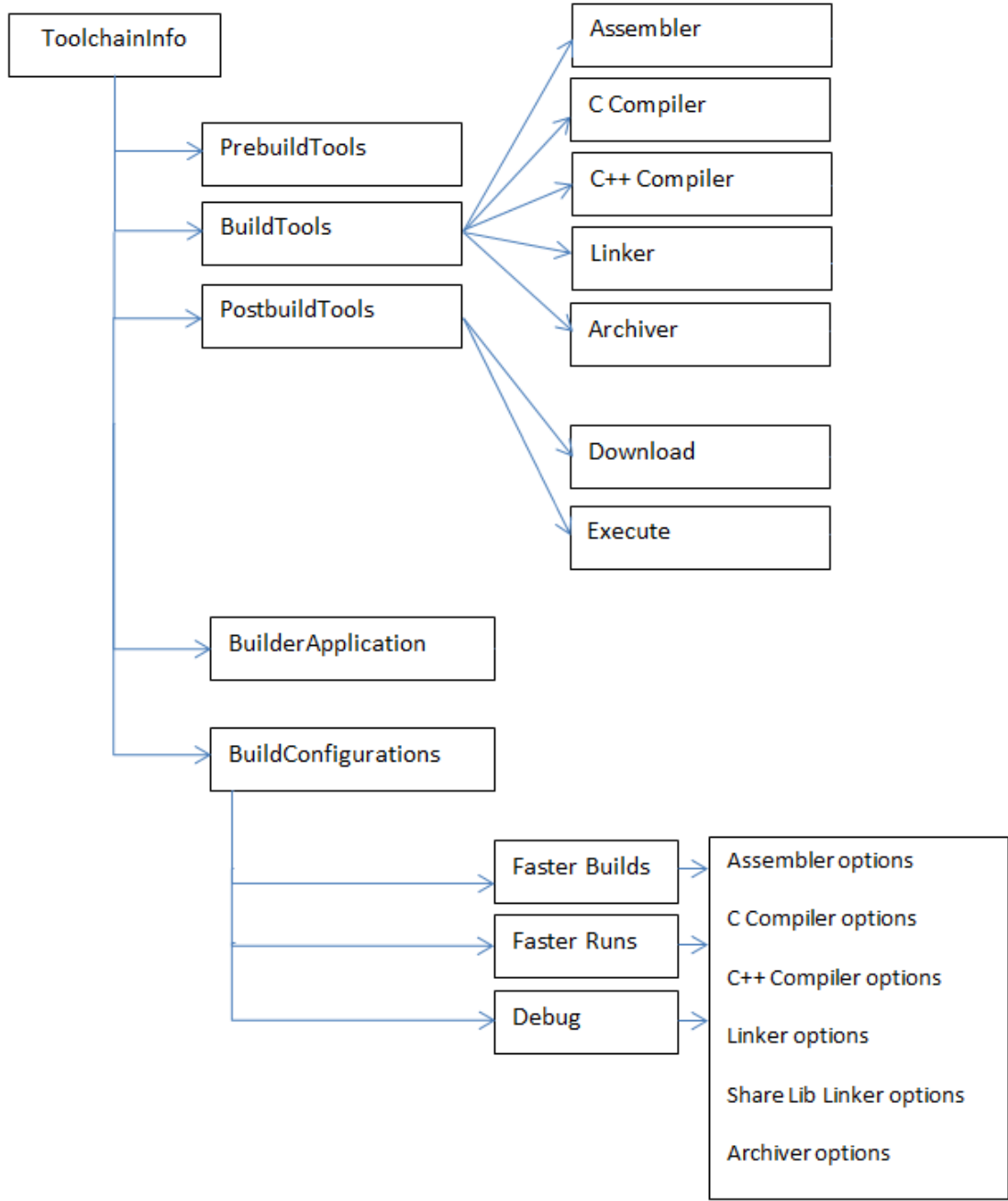
- `coder.make.ToolchainInfo.PrebuildTools` – Tools used before compiling the source files into object files.
- `coder.make.ToolchainInfo.BuildTools` – Tools used for compiling source files and linking/archiving them to form a binary.
- `coder.make.ToolchainInfo.PostbuildTools` – Tools used after the linker/archiver is invoked.
- `coder.make.ToolchainInfo.BuilderApplication` – Tools used to call the `PrebuildTools`, `BuildTools`, and `PostbuildTools`. For example: `gmake`, `nmake`.

Each configuration in `coder.make.ToolchainInfo.BuildConfigurations` applies a set of options to the build tools specified by `coder.make.ToolchainInfo.BuildTools`. By default, these configurations alter the way the assembler, compiler, linker, and archiver operate to produce faster builds, faster runs, and debug.

If you instantiate `coder.make.ToolchainInfo` to support building sources that involve assembler, C, or C++ files, the `coder.make.ToolchainInfo` object contains the default set of build tools shown here.

**ToolchainInfo class & key properties**

**Default build tools and options**



## Create and Edit Toolchain Definition File

This example shows how to create a toolchain definition file by copying and pasting an example file. You then update the relevant elements, and add or remove other elements as needed for your custom toolchain. This is the first step in the typical workflow for creating and using a custom toolchain definition. For more information about the workflow, see “Typical Workflow” on page 21-4.

- 1 Review the list of registered toolchains. In the MATLAB Command Window, enter:

```
coder.make.getToolchains
```

The resulting output includes the list of factory toolchains for your host computer environment, and previously-registered custom toolchains. For example, the following output shows the factory toolchains for a host computer running 64-bit Windows and no custom toolchains.

```
ans =
```

```
'Microsoft Visual C++ 2012 v11.0 | nmake (64-bit Windows)'  
'Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)'  
'Microsoft Visual C++ 2008 v9.0 | nmake (64-bit Windows)'  
'Microsoft Windows SDK v7.1 | nmake (64-bit Windows)'
```

- 2 Create the folder of example files from the “Adding a Custom Toolchain” example by entering the following command in the MATLAB Command Window:

```
coderdemo_setup('coderdemo_intel_compiler');
```

- 3 Copy the example toolchain definition file to another location and rename it. For example:

```
copyfile('intel_tc.m','../newtoolchn_tc.m')
```

- 4 Open the new toolchain definition file in the MATLAB Editor. For example:

```
cd ../  
edit newtoolchn_tc.m
```

- 5 Edit the contents of the new toolchain definition file, providing information for the custom toolchain.

For expanded commentary on an example toolchain definition file, see “Toolchain Definition File with Commentary” on page 21-10.



For reference information about the class attributes and methods you can use in the toolchain definition file, see `coder.make.ToolchainInfo`.

- 6 Save your changes to the toolchain definition file.

Next, create and validate a `coder.make.ToolchainInfo` object from the toolchain definition file, as described in “Create and Validate ToolchainInfo Object” on page 21-16

## Toolchain Definition File with Commentary

### In this section...

“Steps Involved in Writing a Toolchain Definition File” on page 21-10  
 “Write a Function That Creates a ToolchainInfo Object” on page 21-10  
 “Setup” on page 21-11  
 “Macros” on page 21-11  
 “C Compiler” on page 21-12  
 “C++ Compiler” on page 21-12  
 “Linker” on page 21-13  
 “Archiver” on page 21-13  
 “Builder” on page 21-14  
 “Build Configurations” on page 21-14

### Steps Involved in Writing a Toolchain Definition File

This example shows how to create a toolchain definition file and explains each of the steps involved. The example is based on the definition file used in “Adding a Custom Toolchain”. For more information about the workflow, see “Typical Workflow” on page 21-4.

### Write a Function That Creates a ToolchainInfo Object

```

function tc = intel_tc
% INTEL_TC Creates a Intel v12.1 ToolchainInfo object.
% This can be used as a template to add other toolchains on Windows.

% Copyright 2012 The MathWorks, Inc.

tc = coder.make.ToolchainInfo('BuildArtifact','nmake makefile');
tc.Name = 'Intel v12.1 | nmake makefile (64-bit Windows)';
tc.Platform = 'win64';
tc.SupportedVersion = '12.1';

tc.addAttribute('TransformPathsWithSpaces');
tc.addAttribute('RequiresCommandFile');
tc.addAttribute('RequiresBatchFile');

```

The preceding code:

- Defines a function, `intel_tc`, that creates a `coder.make.ToolchainInfo` object and assigns it to a handle, `tc`.

- Overrides the `BuildArtifact` property to create a makefile for `nmake` instead of `gmake`.
- Assigns values to the `Name`, `Platform`, and `SupportedVersion` properties for informational and display purposes.
- Adds three custom attributes to `Attributes` property that are required by this toolchain.
- `'TransformPathsWithSpaces'` converts paths that contain spaces to short Windows paths.
- `'RequiresCommandFile'` generates a linker command file that calls the linker. This avoids problems with calls that exceed the command line limit of 256 characters.
- `'RequiresBatchFile'` creates a `.bat` file that calls the builder application.

## Setup

```
% -----
% Setup
% -----
% Below we are using %ICPP_COMPILER12% as root folder where Intel Compiler is
% installed. You can either set an environment variable or give full path to the
% compilervars.bat file
tc.ShellSetup{1} = 'call %ICPP_COMPILER12%\bin\compilervars.bat intel164';
```

The preceding code:

- Assigns a system call to the `ShellSetup` property.
- The coder `.make.ToolchainInfo.setup` method runs these system calls before it runs tools specified by `PrebuildTools` property.
- Calls `compilervars.bat`, which is shipped with the Intel compilers, to get the set of environment variables for Intel compiler and linkers.

## Macros

```
% -----
% Macros
% -----
tc.addMacro('MW_EXTERNLIB_DIR', ['$(MATLAB_ROOT)\extern\lib\' tc.Platform '\microsoft']);
tc.addMacro('MW_LIB_DIR', ['$(MATLAB_ROOT)\lib\' tc.Platform]);
tc.addMacro('CFLAGS_ADDITIONAL', '-D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('CPPFLAGS_ADDITIONAL', '-Ehs -D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('LIBS_TOOLCHAIN', '$(conlibs)');
tc.addMacro('CVARSFLAG', '');

tc.addIntrinsicMacros({'ldebug', 'conflags', 'cflags'});
```

The preceding code:

- Uses `coder.make.ToolchainInfo.addMacro` method to define macros and assign values to them.
- Uses `coder.make.ToolchainInfo.addIntrinsicMacros` to define macros whose values are specified by the toolchain, outside the scope of your MathWorks software.

## C Compiler

```
% -----  
% C Compiler  
% -----  
  
tool = tc.getBuildTool('C Compiler');  
  
tool.setName('Intel C Compiler');  
tool.setCommand('icl');  
tool.setPath('');  
  
tool.setDirective('IncludeSearchPath', '-I');  
tool.setDirective('PreprocessorDefine', '-D');  
tool.setDirective('OutputFlag', '-Fo');  
tool.setDirective('Debug', '-Zi');  
  
tool.setFileExtension('Source', '.c');  
tool.setFileExtension('Header', '.h');  
tool.setFileExtension('Object', '.obj');  
  
tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

The preceding code:

- Creates a build tool object for the C compiler
- Assigns values to the build tool object properties
- Creates directives and file extensions using name-value pairs
- Sets a command pattern.
- You can use `setCommandPattern` method to control the use of space characters in commands. For example, the two bars in `OUTPUT_FLAG<||>OUTPUT` do not permit a space character between the output flag and the output.

## C++ Compiler

```
% -----  
% C++ Compiler  
% -----  
  
tool = tc.getBuildTool('C++ Compiler');
```

```

tool.setName('Intel C++ Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath', '-I');
tool.setDirective('PreprocessorDefine', '-D');
tool.setDirective('OutputFlag', '-Fo');
tool.setDirective('Debug', '-Zi');

tool.setFileExtension('Source', '.cpp');
tool.setFileExtension('Header', '.hpp');
tool.setFileExtension('Object', '.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

```

The preceding code:

- Creates a build tool object for the C++ compiler
- Is very similar to the build tool object for the C compiler

## Linker

```

% -----
% Linker
% -----

tool = tc.getBuildTool('Linker');

tool.setName('Intel C/C++ Linker');
tool.setCommand('xilink');
tool.setPath('');

tool.setDirective('Library', '-L');
tool.setDirective('LibrarySearchPath', '-I');
tool.setDirective('OutputFlag', '-out:');
tool.setDirective('Debug', '');

tool.setFileExtension('Executable', '.exe');
tool.setFileExtension('Shared Library', '.dll');

tool.DerivedFileExtensions = horzcat(tool.DerivedFileExtensions, { ...
    ['_' tc.Platform '.lib'], ...
    ['_' tc.Platform '.exp']});

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

```

The preceding code:

- Creates a build tool object for the linker
- Assigns values to the `coder.make.BuildTool.DerivedFileExtensions`

## Archiver

```

% -----

```

```

% Archiver
% -----

tool = tc.getBuildTool('Archiver');

tool.setName('Intel C/C++ Archiver');
tool.setCommand('xilib');
tool.setPath('');

tool.setDirective('OutputFlag','-out:');

tool.setFileExtension('Static Library','.lib');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');

```

The preceding code:

- Creates a build tool object for the archiver.

## Builder

```

% -----
% Builder
% -----

tc.setBuilderApplication(tc.Platform);

```

The preceding code:

- Gives the value of `coder.make.ToolchainInfo.Platform` as the argument for setting the value of `BuilderApplication`. This sets the default values of the builder application based on the platform. For example, when `Platform` is `win64`, this line sets the delete command to `'del'`; the display command to `'echo'`, the file separator to `'\'`, and the include directive to `'!include'`.

## Build Configurations

```

% -----
% BUILD CONFIGURATIONS
% -----

optimsOff0pts = {'/c /Od'};
optimsOn0pts = {'/c /O2'};
cCompilerOpts = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL)';
cppCompilerOpts = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL)';
linkerOpts = {'$(ldebug) $(conflags) $(LIBS_TOOLCHAIN)'};
sharedLinkerOpts = horzcat(linkerOpts,'-dll -def:$(DEF_FILE)');
archiverOpts = {'/nologo'};

% Get the debug flag per build tool
debugFlag.CCompiler = '$(CDEBUG)';
debugFlag.CppCompiler = '$(CPPDEBUG)';
debugFlag.Linker = '$(LDDEBUG)';

```

```
debugFlag.Archiver      = '$(ARDEBUG)';

cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler', horzcat(cCompilerOpts, optimsOff0pts));
cfg.setOption('C++ Compiler', horzcat(cppCompilerOpts, optimsOff0pts));
cfg.setOption('Linker', linkerOpts);
cfg.setOption('Shared Library Linker', sharedLinkerOpts);
cfg.setOption('Archiver', archiverOpts);

cfg = tc.getBuildConfiguration('Faster Runs');
cfg.setOption('C Compiler', horzcat(cCompilerOpts, optimsOn0pts));
cfg.setOption('C++ Compiler', horzcat(cppCompilerOpts, optimsOn0pts));
cfg.setOption('Linker', linkerOpts);
cfg.setOption('Shared Library Linker', sharedLinkerOpts);
cfg.setOption('Archiver', archiverOpts);

cfg = tc.getBuildConfiguration('Debug');
cfg.setOption('C Compiler', horzcat(cCompilerOpts, optimsOff0pts, debugFlag.CCompiler));
cfg.setOption('C++ Compiler', horzcat(cppCompilerOpts, optimsOff0pts, debugFlag.CppCompiler));
cfg.setOption('Linker', horzcat(linkerOpts, debugFlag.Linker));
cfg.setOption('Shared Library Linker', horzcat(sharedLinkerOpts, debugFlag.Linker));
cfg.setOption('Archiver', horzcat(archiverOpts, debugFlag.Archiver));

tc.setBuildConfigurationOption('all', 'Download', '');
tc.setBuildConfigurationOption('all', 'Execute', '');
tc.setBuildConfigurationOption('all', 'Make Tool', '-f $(MAKEFILE)');
```

The preceding code:

- Creates each build configuration object.
- Sets the value of each option for a given build configuration object.

## Create and Validate ToolchainInfo Object

This example shows how to create and validate a `coder.make.ToolchainInfo` object from the toolchain definition file.

Before you create and validate a `ToolchainInfo` object, create and edit a toolchain definition file, as described in “Create and Edit Toolchain Definition File” on page 21-8.

- 1 Use the function defined by the toolchain definition file to create a `coder.make.ToolchainInfo` object and assign the object to a handle. For example, the MATLAB Command Window, enter:

```
tc = newtoolchn_tc
```

- 2 Use the `coder.make.ToolchainInfo.validate` method with the `coder.make.ToolchainInfo` object. For example, enter:

```
tc.validate
```

If the `coder.make.ToolchainInfo` object contains errors, the validation method displays error messages in the MATLAB Command Window.

- 3 Search the toolchain definition file for items named in the error message (without quotes) and update the values.
- 4 Repeat the process of creating and validating the `ToolchainInfo` object until there are no more errors.

Next, register the custom toolchain, as described in “Register the Custom Toolchain” on page 21-17.

For more information, see “Troubleshooting Custom Toolchain Validation” on page 21-20.



## Register the Custom Toolchain

Before you register the custom toolchain, create and validate the `ToolchainInfo` object, as described in “Create and Validate ToolchainInfo Object” on page 21-16.

- 1 Use the `save` function to create a MATLAB-formatted binary file (MAT-file) from the `coder.make.ToolchainInfo` object in the MATLAB workspace variables. For example, enter:

```
save newtoolchn_tc tc
```

The new `.mat` file appears in the Current Folder.

- 2 Create a new MATLAB function called `rtwTargetInfo.m`.
- 3 Copy and paste the following text into `rtwTargetInfo.m`:

```
function rtwTargetInfo(tr)
% RTWTARGETINFO Target info callback

tr.registerTargetInfo(@loc_createToolchain);

end

% -----
% Create the ToolchainInfoRegistry entries
% -----
function config = loc_createToolchain

    config(1)          = coder.make.ToolchainInfoRegistry;
    config(1).Name     = '<mytoolchain v#.##> | <buildartifact (platform)>';
    config(1).FileName = fullfile('<yourdir>', '<mytoolchain_tc.mat>');
    config(1).TargetHWDeviceType = {'<devicetype>'};
    config(1).Platform = {'<win64>'};

% To register more custom toolchains:
% 1) Copy and paste the five preceding 'config' lines.
% 2) Increment the index of config().
% 3) Replace the values between angle brackets.
% 4) Remove the angle brackets.

end
```

- 4 Replace the items between angle brackets with real values, and remove the angle brackets:

- **Name** — Provide a unique name for the toolchain definition file using the recommended format: name, version number, build artifact, and platform.
- **FileName** — The full path and name of the MAT-file.
- **TargetHWDeviceType** — The platform or platforms supported by the custom toolchain.

- **Platform** — The host operating system supported by the custom toolchain. For all platforms, use the following wildcard: '\*'

For more information, refer to the corresponding `ToolchainInfo` properties in “Properties”.

Here are some example entries for an Intel toolchain that uses `nmake`, based on “Adding a Custom Toolchain”:

```
config(1)          = coder.make.ToolchainInfoRegistry;  
config(1).Name     = 'Intel v12.1 | nmake makefile (64-bit Windows)';  
config(1).FileName = fullfile(fileparts(mfilename('fullpath')), 'intel_tc.mat');  
config(1).TargetHWDeviceType = {'ARM9', 'ARM10', 'ARM11'};  
config(1).Platform = {computer('arch')};
```

- 5 Save the new `rtwTargetInfo.m` file to a folder that is on the MATLAB path.
- 6 List all of the `rtwTargetInfo.m` files on the MATLAB path. Using the MATLAB Command Window, enter:

```
which -all rtwTargetInfo
```

- 7 Verify that the `rtwTargetInfo.m` file you just created appears in the list of files.
- 8 Reset `TargetRegistry` so it picks up the custom toolchain from the `rtwTargetInfo.m` file:

```
RTW.TargetRegistry.getInstance('reset');
```

Next, use the custom toolchain, as described in “Use the Custom Toolchain” on page 21-19.

## Use the Custom Toolchain

You can use a custom toolchain when generating a static or dynamic library or an executable. You cannot use one to generate MEX functions. To specify which compiler to use for MEX-function generation, see “Setting Up the C or C++ Compiler”.

Before using the custom toolchain, register the custom toolchain, as described in “Register the Custom Toolchain” on page 21-17.

- 1 Use `coder.config` to create a configuration object. For example:

```
cfg = coder.config('exe');
```

- 2 Get the value of `config(end).Name` from the `rtwTargetInfo.m` file. Then assign that value to the `cfg.Toolchain` property:

```
cfg.Toolchain = 'mytoolchain v#.#' | 'buildartifact (platform)'
```

With the “Adding a Custom Toolchain” example, this would look like:

```
cfg.Toolchain = 'Intel v12.1 | nmake makefile (64-bit Windows)';
```

- 3 Perform other steps required to generate code, as described in “Deployment”. For example, specify the path and file name of the source code:

```
cfg.CustomSource = 'filename_main.c';  
cfg.CustomInclude = pwd;
```

- 4 When you generate code using the `codegen` function, specify the configuration object that uses the custom toolchain. For example:

```
codegen -config cfg filename
```

You have completed the full workflow of creating and using a custom toolchain described in “Custom Toolchain Registration” on page 21-2.

## Troubleshooting Custom Toolchain Validation

### In this section...

“Build Tool Command Path Incorrect” on page 21-20

“Build Tool Not in System Path” on page 21-20

“Tool Path Does Not Exist” on page 21-21

“Unsupported Platform” on page 21-21

“Toolchain is Not installed” on page 21-22

“Project or Configuration is Using the Template Makefile” on page 21-22

“Skipped Validation of Build Tool “Download” or “Execute”” on page 21-23

### Build Tool Command Path Incorrect

If the path or command file name are not correct, validation displays:

```
Cannot find file 'path+command'. The file does not exist.
```

Consider the following two lines from an example toolchain definition file:

```
tool.setCommand('abc');  
tool.setPath('/toolchain/');
```

To correct this issue:

- Check that the build tool is installed.
- Review the arguments given for the `tool.setCommand` and `tool.setPath` lines in toolchain definition file.

### Build Tool Not in System Path

When the build tool’s path is not provided and the command file is not in the system path, validation displays:

```
Cannot find 'command'. It is not in the system path.
```

Consider the following two lines from an example toolchain definition file:

```
tool.setCommand('icl');  
tool.setPath('');
```

Because the argument for `setPath()` is `' '` instead of an absolute path, the build tool must be on the system path.

To correct this issue:

- Use `coders.make.ToolchainInfo.ShellSetup` property to add the path to the toolchain installation.
- Use your system setup to add the toolchain installation directory to system environment path.

Otherwise, replace `' '` with the absolute path of the command file.

## Tool Path Does Not Exist

If the path of the build tool path is provided, but does not exist, validation displays:

```
Path 'toolpath' does not exist.
```

To correct this issue:

- Check the actual path of the build tool. Then, update the value of `coders.make.BuildTool.setPath` in the toolchain definition file.
- Use your system setup to add the toolchain installation directory to system environment path. Then, set the value of `coders.make.BuildTool.setPath` to `' '`.

## Unsupported Platform

If the toolchain is not supported on the host computer platform, validation displays:

```
Toolchain 'tlchn' is supported on a 'pltfрма' platform. However, you are running on a 'pltfрmb' platform.
```

To correct this issue:

- Check the `coders.make.ToolchainInfo.Platform` property in your toolchain definition file for errors.
- Update or replace the toolchain definition file with one that supports your host computer platform.
- Change host computer platforms.

## Toolchain is Not installed

If the toolchain is not installed, validation displays:

Toolchain is not installed.

To correct this issue, install the expected toolchain, or verify that you selected the correct toolchain, as described in “Use the Custom Toolchain” on page 21-19.

## Project or Configuration is Using the Template Makefile

By default, MATLAB Coder tries to use the selected build toolchain to build the generated code. However, if the makefile configuration options detailed in the following sections are **not** set to their default value, MATLAB Coder cannot use the toolchain and reverts to using the template makefile approach for building the generated code.

### MATLAB Coder Project Settings

Project Settings Dialog Box All Settings Parameter Name	Default Setting
Generate makefile	Yes
Make command	make_rtw
Template makefile	default_tmf
Compiler optimization level	Off

### Command-line Configuration Parameters for the codegen function

coder.CodeConfig or coder.EmbeddedCodeConfig Parameter Name	Default Value
GenerateMakefile	'true'
MakeCommand	'make_rtw'
TemplateMakefile	'default_tmf'
CCompilerOptimization	'Off'

To use the toolchain approach, reset your configuration options to these default values manually or:

- To reset settings for project `project_name`, at the MATLAB command line, enter:

```
coder.make.upgradeMATLABCoderProject(project_name)
```

- To reset command-line settings for configuration object `config`, create an updated configuration object `new_config` and then use `new_config` with the `codegen` function in subsequent builds. At the MATLAB command line, enter:

```
new_config = coder.make.upgradeCoderConfigObject(config);
```

## Skipped Validation of Build Tool “Download” or “Execute”

Even though the Validation Report states “Toolchain Validation Result: Passed” it includes one or both of the following notes:

```
### Validation of build tool "Download"  
Skipped. No "Download" build tool is specified.  
### Validation of build tool "Execute"  
Skipped. "Execute" build tool "${PRODUCT}" cannot be validated.
```

To correct this issue, update the toolchain definition file and re-register the updated toolchain. For more information, see:

- “Create and Edit Toolchain Definition File” on page 21-8
- “Create and Validate ToolchainInfo Object” on page 21-16
- “Register the Custom Toolchain” on page 21-17

## Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers

Symptom: During a software build, a build error occurs; variables don't resolve correctly.

If your toolchain uses a one-pass or single-pass linker, prevent circular data dependencies by adding the `StartLibraryGroup` and `EndLibraryGroup` linker directives to the toolchain definition file.

For example, if the linker is like GNU `gcc`, then the directives are `'-Wl,--start-group'` and `'-Wl,--end-group'`, as shown here:

```
% -----  
% Linker  
% -----  
  
tool = tc.getBuildTool('Linker');  
  
tool.setName(          'GNU Linker');  
tool.setCommand(      'gcc');  
tool.setPath(         '');  
  
tool.setDirective(    'Library',          '-l');  
tool.setDirective(    'LibrarySearchPath', '-L');  
tool.setDirective(    'OutputFlag',       '-o');  
tool.setDirective(    'Debug',            '-g');  
tool.addDirective(    'StartLibraryGroup', {'-Wl,--start-group'});  
tool.addDirective(    'EndLibraryGroup',   {'-Wl,--end-group'});  
  
tool.setFileExtension( 'Executable',      '');  
tool.setFileExtension( 'Shared Library',   '.so');
```



# Deploying Generated Code

---

- “Call a C Static Library Function from C Code” on page 22-2
- “Call a C/C++ Static Library Function from MATLAB Code” on page 22-4
- “Call Generated C/C++ Functions” on page 22-6
- “Use a MATLAB Coder Dynamic Library in a Simple Microsoft Visual Studio Project” on page 22-9
- “Specify External File Locations” on page 22-12

## Call a C Static Library Function from C Code

This example shows how to call a generated C library function from C code. It uses the C static library function `absval` described in “Call a C/C++ Static Library Function from MATLAB Code” on page 22-4.

- 1 Write a `main` function in C that does the following:
  - Includes the generated header file, which contains the function prototypes for the library function.
  - Calls the `initialize` function before calling the library function for the first time.
  - Calls the `terminate` function after calling the library function for the last time.

Here is an example of a C `main` function that calls the library function `absval`:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "absval.h"

int main(int argc, char *argv[])
{
    absval_initialize();

    printf("absval(-2.75)=%g\n", absval(-2.75));

    absval_terminate();

    return 0;
}
```

- 2 Configure your target to integrate this custom C `main` function with your generated code, as described in “Specify External File Locations” on page 22-12.

For example, you can define a configuration object that points to the custom C code:

- a Create a configuration object. At the MATLAB prompt, enter:

```
cfg = coder.config('exe');
```
- b Set custom code properties on the configuration object, as in these example commands:

```
cfg.CustomSource = 'main.c';  
cfg.CustomInclude = 'c:\myfiles';
```

- 3** Generate the C executable. Use the `-args` option to specify that the input is a real, scalar double. At the MATLAB prompt, enter:

```
codegen -config cfg absval -args {0}
```

- 4** Call the executable. For example:

```
absval(-2.75)
```

## Call a C/C++ Static Library Function from MATLAB Code

This example shows how to call a C/C++ library function from MATLAB code that is suitable for code generation.

Suppose you have a MATLAB file `absval.m` that contains the following function:

```
function y = absval(u) %#codegen
    y = abs(u);
end
```

To generate a C static library function and call it from MATLAB code:

- 1 Generate the C library for `absval.m`.

```
codegen -config:lib absval -args {0.0}
```

Here are key points about this command:

- The `-config:lib` option instructs MATLAB Coder to generate `absval` as a C static library function.

The default target language is C. To change the target language to C++, see “Specify a Language for Code Generation” on page 19-21.

- MATLAB Coder creates the library `absval.lib` (or `absval.a` on Linux Torvalds' Linux) and header file `absval.h` in the folder `/emcprj/rtwlib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the C library.
- The `-args` option specifies the class, size, and complexity of the primary function input `u` by example, as described in “Define Input Properties by Example at the Command Line” on page 19-44.

- 2 Write a MATLAB function to call the generated library:

```
%#codegen
function y = callabsval

% Call the initialize function before
% calling the C function for the first time
coder.ceval('absval_initialize');

y = -2.75;
y = coder.ceval('absval',y);
```

```
% Call the terminate function after  
% calling the C function for the last time  
coder.ceval('absval_terminate');
```

The MATLAB function `callabsval` uses the interface `coder.ceval` to call the generated C functions `absval_initialize`, `absval`, and `absval_terminate`. You must use this function to call C functions from generated code. For more information, see “Call Generated C/C++ Functions” on page 22-6.

- 3** Convert the code in `callabsval.m` to a MEX function so that you can call the C library function `absval` directly from the MATLAB prompt.

- a** Generate the MEX function using `codegen` as follows:

- Create a code generation configuration object for a MEX function:

```
cfg = coder.config
```

- On Microsoft Windows platforms, use this command:

```
codegen -config cfg callabsval codegen/lib/absval/absval.lib  
        codegen/lib/absval/absval.h
```

By default, this command creates, in the current folder, a MEX function named `callabsval_mex`

On the Linux Torvalds' Linux platform, use this command:

```
codegen -config cfg callabsval codegen/lib/absval/absval.a  
        codegen/lib/absval/absval.h
```

- b** At the MATLAB prompt, call the C library by running the MEX function. For example, on Windows:

```
callabsval_mex
```

## Call Generated C/C++ Functions

### In this section...

“Conventions for Calling Functions in Generated Code” on page 22-6

“How to Call C/C++ Functions from MATLAB Code” on page 22-6

“Calling Initialize and Terminate Functions” on page 22-7

“Calling C/C++ Functions with Multiple Outputs” on page 22-8

“Calling C/C++ Functions that Return Arrays” on page 22-8

### Conventions for Calling Functions in Generated Code

When generating code, MATLAB Coder uses the following calling conventions:

- Passes arrays by reference as inputs.
- Returns arrays by reference as outputs.
- Unless you optimize your code by using the same variable as both input and output, passes scalars by value as inputs. In that case, MATLAB Coder passes the scalar by reference.
- Returns scalars by value for single-output functions.
- Returns scalars by reference:
  - For functions with multiple outputs.
  - When you use the same variable as both input and output.

For more information about optimizing your code by using the same variable as both input and output, see “Eliminate Redundant Copies of Function Inputs”.

### How to Call C/C++ Functions from MATLAB Code

You can call the C/C++ functions generated for libraries as custom C/C++ code from MATLAB functions that are suitable for code generation. For static libraries, you must use the `coder.ceval` function to wrap the function calls, as in this example:

```
function y = callmyCFunction %#codegen
    y = 1.5;
```

```
y = coder.ceval('myCFunction',y);
end
```

Here, the MATLAB function `callmyCFunction` calls the custom C function `myCFunction`, which takes one input argument.

For dynamically-linked libraries, you can also use `coder.ceval`.

There are additional requirements for calling C/C++ functions from the MATLAB code in the following situations:

- You want to call generated C/C++ libraries or executables from a MATLAB function. Call housekeeping functions generated by MATLAB Coder, as described in “Calling Initialize and Terminate Functions” on page 22-7.
- You want to call C/C++ functions that are generated from MATLAB functions that have more than one output, as described in “Calling C/C++ Functions with Multiple Outputs” on page 22-8.
- You want to call C/C++ functions that are generated from MATLAB functions that return arrays, as described in “Calling C/C++ Functions that Return Arrays” on page 22-8.

## Calling Initialize and Terminate Functions

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder automatically generates two housekeeping functions that you must call along with the C/C++ function.

Housekeeping Function	When to Call
<code>primary_function_name_initialize</code>	Before you call your C/C++ executable or library function for the first time
<code>primary_function_name_terminate</code>	After you call your C/C++ executable or library function for the last time

From C/C++ code, you can call these functions directly. However, to call them from MATLAB code that is suitable for code generation, you must use the `coder.ceval` function. `coder.ceval` is a MATLAB Coder function, but is not supported by the native MATLAB language. Therefore, if your MATLAB code uses this function, use `coder.target` to disable these calls in MATLAB and replace them with equivalent functions.

## Calling C/C++ Functions with Multiple Outputs

Although MATLAB Coder can generate C/C++ code from MATLAB functions that have multiple outputs, the generated C/C++ code cannot return multiple outputs directly because the C/C++ language does not support multiple return values. Instead, you can achieve the effect of returning multiple outputs from your C/C++ function by using `coder.wref` with `coder.ceval`.

## Calling C/C++ Functions that Return Arrays

Although MATLAB Coder can generate C/C++ code from MATLAB functions that return values as arrays, the generated code cannot return arrays *by value* because the C/C++ language is limited to returning single, scalar values. Instead, you can return arrays from your C/C++ function *by reference* as pointers by using `coder.wref` with `coder.ceval`.



## Use a MATLAB Coder Dynamic Library in a Simple Microsoft Visual Studio Project

These steps outline how to create and configure a simple Microsoft Visual Studio® Win32 Console Application project to call a dynamic library (DLL) that was generated by MATLAB Coder. This procedure provides information on how to do this in Microsoft Visual Studio 2008, the steps might differ in other versions of Microsoft Visual Studio.

- 1 Create a MATLAB function `foo` and save it as `foo.m` in a local writable folder, for example, `c:\dll_test`.

```
function c = foo(a) %#codegen
    c = sqrt(a);
end
```

- 2 Generate a DLL for the MATLAB function `foo`, using the `-args` option to specify that the input `a` is a real double.

```
codegen -report -config:dll foo -args {0}
```

On Microsoft Windows systems, `codegen` generates a C dynamic library, `foo.dll`, and supporting files, in the default folder, `codegen/dll/foo`.

- 3 In Microsoft Visual Studio, create an empty Win32 Console Application project.
- 4 Verify that the project configuration specifies architecture that matches your computer. By default, MATLAB Coder builds a DLL for the platform that you are working on, but Microsoft Visual Studio builds for Win32.

In Microsoft Visual Studio 2008:

- a Select **Build > Configuration Manager**.
  - b In the **Configuration Manager**, set **Active solution platform** to match your platform.
- 5 Configure the project to use the release version of the C run-time library. By default, the Microsoft Visual Studio project uses the debug version of the C run-time library, but the DLL generated by MATLAB Coder uses the release version. For example, in Microsoft Visual Studio 2008:
    - a Select **Build > Configuration Manager**.
    - b In the **Configuration Manager**, set **Active solution configuration** to **Release**.
  - 6 Create a main file that calls `foo.dll`. The main function **must**:

- Include the generated header file, which contains the function prototypes for the library function.
- Call the initialize function before calling the library function for the first time.
- Call the terminate function after calling the library function for the last time.

For example:

```
#include "foo.h"
#include "foo_initialize.h"
#include "foo_terminate.h"
#include <stdio.h>

int main()
{
    foo_initialize();
    printf("%f\n", foo(25));
    foo_terminate();
    getchar();
    return 0;
}
```

- 7 Add the `main` file to the project.
- 8 In the project, add the folder containing the generated header file to the list of additional include directories. For example, in Microsoft Visual Studio 2008:
  - a Right-click the project name and select **Properties**.
  - b Under **C/C++ > General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Include Directories**.
- 9 Add the folder containing the `.lib` file (by default, this is the folder containing the `.dll`) to the list of additional library directories. For example, in Microsoft Visual Studio 2008:
  - a Right-click the project name and select **Properties**.
  - b Under **Linker > General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Library Directories**.
- 10 Add the `.lib` file name to the list of additional libraries. For example, in Microsoft Visual Studio 2008:
  - a Right-click the project name and select **Properties**.

- b** Under **Linker > Input**, add `foo.lib` to **Additional Dependencies**.

You are now ready to build your project.

---

**Note:** To run the application, you must either add the folder containing the generated DLL to your path or run from the folder that contains the DLL.

---

## Specify External File Locations

### In this section...

“External File Locations for External Code Integration” on page 22-12

“Specify External Files in a Class Derived from `coder.ExternalDependency`” on page 22-12

“Specify External Files in MATLAB Code Using `coder.updateBuildInfo`” on page 22-12

“Specify External Files in the Project Settings Dialog Box” on page 22-13

“Specify External Files at the Command Line” on page 22-13

“Specify External Files with Configuration Objects” on page 22-14

### External File Locations for External Code Integration

To integrate external code with generated C/C++ code, you must specify the locations of your external source files, header files, and libraries to MATLAB Coder.

You can specify the file locations:

- In a class definition file, when you derive a class from `coder.ExternalDependency`
- In your MATLAB code using the `coder.updateBuildInfo` function
- In the project settings dialog box
- From the command line
- In the configuration object

### Specify External Files in a Class Derived from `coder.ExternalDependency`

When you derive a class from `coder.ExternalDependency`, you write a method `updateBuildInfo` that specifies the locations of the external files required for the build. See `coder.ExternalDependency`.

### Specify External Files in MATLAB Code Using `coder.updateBuildInfo`

In your MATLAB code, you can call `coder.updateBuildInfo` to specify the locations of external files. See `coder.updateBuildInfo`.

## Specify External Files in the Project Settings Dialog Box

- 1 On the project **Build** tab, click the **More settings** link to open the Project Settings dialog box.
- 2 On the **Custom Code** tab, under **Custom C-code to include in generated files**, specify **Source file** and **Header file**. **Source file** specifies that the code appear at the top of generated C/C++ source files. **Header file** specifies that the code appear at the top of generated header files.

Custom Code Property	Description
Under <b>Additional files and directories to be built</b> , provide an absolute path or a path relative to the project folder.	
<b>Include directories</b>	Specifies a list of folders that contain custom header, source, object, or library files. Separate list items with a semicolon.
<b>Source files</b>	Specifies additional custom C/C++ files to be compiled with the MATLAB file. Separate list items with a semicolon.
<b>Libraries</b>	Specifies the names of object or library files to be linked with the generated code. Separate list items with a semicolon.
Under <b>Custom C-code to include in generated files</b>	
<b>Source file</b>	Specifies code to appear at the top of generated C/C++ source files.
<b>Header file</b>	Specifies custom code to appear at the top of generated header files

## Specify External Files at the Command Line

When you compile MATLAB function with MATLAB Coder, you can specify custom C/C++ files — such as source, header, and library files — on the command line along with your MATLAB file. For example, suppose you want to generate an embeddable C code executable that integrates a custom C function `myCfcn` with a MATLAB function `myMfcn` that has no input parameters. The custom source and header files for `myCfcn` reside in the folder `C:\custom`. You can use the following command to generate the code:

```
codegen C:\custom\myCfcn.c C:\custom\myCfcn.h myMfcn
```

## Specify External Files with Configuration Objects

You can specify custom C/C++ files by setting custom code properties on configuration objects.

- 1 Define a configuration object, as described in “Creating Configuration Objects” on page 19-28.

For example:

```
cc = coder.config('lib');
```

- 2 Set one or more of the custom code properties.

Custom Code Property	Description
CustomInclude	<p>Specifies a list of folders that contain custom header, source, object, or library files.</p> <hr/> <p><b>Note:</b> If your folder path name contains spaces, you must enclose it in double quotes:</p> <pre>cc.CustomInclude = 'C:\Program Files\MATLAB\work'</pre>
CustomSource	Specifies additional custom C/C++ files to be compiled with the MATLAB file.
CustomLibrary	Specifies the names of object or library files to be linked with the generated code.
CustomSourceCode	Specifies code to insert at the top of each generated C/C++ source file.
CustomHeaderCode	Specifies custom code to insert at the top of each generated C/C++ header file.

For example:

```
cc.CustomInclude = 'C:\custom\src C:\custom\lib';
cc.CustomSource = 'cfunction.c';
cc.CustomLibrary = 'chelper.obj clibrary.lib';
cc.CustomSourceCode = '#include "cgfunction.h"';
```

- 3 Compile the MATLAB code specifying the code generation configuration object.

---

**Note:** If you generate code for a function that has input parameters, you must specify the inputs. “Primary Function Input Specification”

---

```
codegen -config cc myFunc
```

**4** Call custom C/C++ functions.

From...	Call...
C/C++ source code	Custom C/C++ functions directly
MATLAB code, compiled on the MATLAB Coder path	Custom C/C++ functions using <code>coder.ceval</code> .

For example, from MATLAB code:

```
...  
y = 2.5;  
y = coder.ceval('myFunc',y);  
...
```



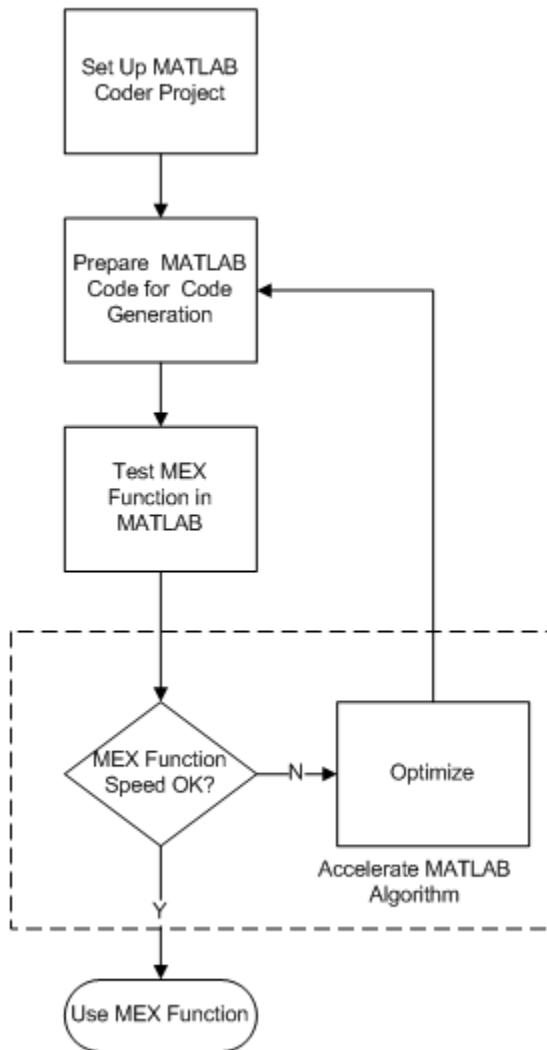


# Accelerating MATLAB Algorithms

---

- “Workflow for Accelerating MATLAB Algorithms” on page 23-2
- “Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms” on page 23-4
- “Edge Detection on Images” on page 23-7
- “Accelerate MATLAB Algorithms” on page 23-14
- “Modifying MATLAB Code for Acceleration” on page 23-15
- “Control Run-Time Checks” on page 23-16
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 23-18
- “Control Compilation of parfor-Loops” on page 23-24
- “Reduction Assignments in parfor-Loops” on page 23-25
- “Classification of Variables in parfor-Loops” on page 23-26
- “Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)” on page 23-35
- “Specify Maximum Number of Threads in parfor-Loops” on page 23-36
- “Troubleshooting parfor-Loops” on page 23-37
- “Accelerating Simulation of Bouncing Balls” on page 23-38

## Workflow for Accelerating MATLAB Algorithms



## See Also

- “MATLAB Coder Project Set Up Workflow”
- “Workflow for Preparing MATLAB Code for Code Generation”
- “Workflow for Testing MEX Functions in MATLAB”
- “Modifying MATLAB Code for Acceleration” on page 23-15

## Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms

### In this section...

“Accelerate Code That Dominates Execution Time” on page 23-4

“Include Loops Inside MEX Function” on page 23-4

“Avoid Generating MEX Functions from Unsupported Functions” on page 23-5

“Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time” on page 23-6

“Minimize MEX Function Calls” on page 23-6

When you choose a section of MATLAB code to accelerate, the following practices are recommended.

### Accelerate Code That Dominates Execution Time

Find the section of MATLAB code that dominates run time. Accelerate this section of the code using a MEX function as follows:

- 1 Place this section of the code inside a separate MATLAB function.
- 2 From this MATLAB function, generate a MEX function.
- 3 From your original MATLAB code, call the MEX function.

To find the execution time of each MATLAB instruction, use MATLAB Profiler.

- To open the Profiler from the command line, type `profile viewer`.
- To open Profiler from the MATLAB Editor window, under the **Editor** tab, click **Run and Time**.

For more information about using the Profiler to measure run time of a MATLAB code, see “Running the Profiler”.

### Include Loops Inside MEX Function

Instead of calling a MEX function inside a loop in the MATLAB code, include the loop inside the MEX function. Including the loop eliminates the overheads in calling the MEX function for every run of the loop.

For example, the following code finds the greatest element in every row of a 1000-by-1000 matrix, `mat`. You can accelerate sections 1,2, and 3 using a MEX function.:

```
% Section 1 begins
for i = 1:10000

    % Section 2 begins
    max = mat(i,0); % Initialize max
    for j = 1:10000

        % Section 3 begins
        if (mat(i,j) > max)
            max = mat(i,j) % Store the current maximum
        end
        % Section 3 ends

    end
    % Section 2 ends

end
% Section 1 ends
```

Accelerate section 1 using a MEX function. Accelerate section 1 first so that the MEX function is called only once.. If you cannot accelerate section 1 first, then accelerate sections 2 or 3, in that order. If section 2 (or 3) is accelerated using a MEX function, the function is called 10000 (or  $10000 \times 10000$ ) times.

## Avoid Generating MEX Functions from Unsupported Functions

Check that the section of MATLAB code that you accelerate does not contain many functions and language features that are unsupported by MATLAB Coder. For a list of supported functions, see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List”.

---

**Note:** In certain situations, you might have to accelerate sections of code even though they contain a few unsupported functions. Declare an unsupported function as extrinsic to invoke the original MATLAB function instead of the code generated for the function. You can declare a function as extrinsic by using `coder.extrinsic` or wrapping it in an `feval` statement. See “Call MATLAB Functions”.

---

## Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time

Use MEX functions to accelerate MATLAB code only if user-generated code dominates the run time.

Avoid generating MEX functions if computationally intensive, built-in MATLAB functions dominate the run time. These functions are pre-compiled and optimized, so the MATLAB code is not accelerated significantly using a MEX function. Examples of such functions include `svd`, `eig`, `fft`, `qr`, `lu`.

---

**Tip** You can invoke computationally intensive, built-in MATLAB functions from your MEX function. Declare the MATLAB function as extrinsic using `coder.extrinsic` or wrap it in an `feval` statement. For more information, see “Call MATLAB Functions”.

---

## Minimize MEX Function Calls

Accelerate as much of the MATLAB code as possible using one MEX function instead of several MEX functions called at lower levels. This minimizes the overheads in calling the MEX functions.

For example, consider the function, `testfunc`, which calls two functions, `testfunc_1` and `testfunc_2`:

```
function [y1,y2] = testfunc(x1,x2)
    y1 = testfunc_1(x1,x2);
    y2 = testfunc_2(x1,x2);
end
```

Instead of generating MEX functions individually for `testfunc_1` and `testfunc_2`, and then calling the MEX functions in `testfunc`, generate a MEX function for `testfunc` itself.

## Edge Detection on Images

This example shows how to generate a standalone C library from MATLAB code that implements a simple Sobel filter that performs edge detection on images. The example also shows how to generate and test a MEX function in MATLAB prior to generating C code to verify that the MATLAB code is suitable for code generation.

### Prerequisites

There are no prerequisites for this example.

### Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will only contain the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), you should change your working folder.

### Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_edge_detection');
```

### About the 'sobel' Function

The `sobel.m` function takes an image (represented as a double matrix) and a threshold value and returns an image with the edges detected (based on the threshold value).

```
type sobel
```

```
% edgeImage = sobel(originalImage, threshold)
% Sobel edge detection. Given a normalized image (with double values)
% return an image where the edges are detected w.r.t. threshold value.
function edgeImage = sobel(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = [1 2 1; 0 0 0; -1 -2 -1];
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k', 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

### **Generate the MEX Function**

Generate a MEX function using the 'codegen' command.

```
codegen sobel
```

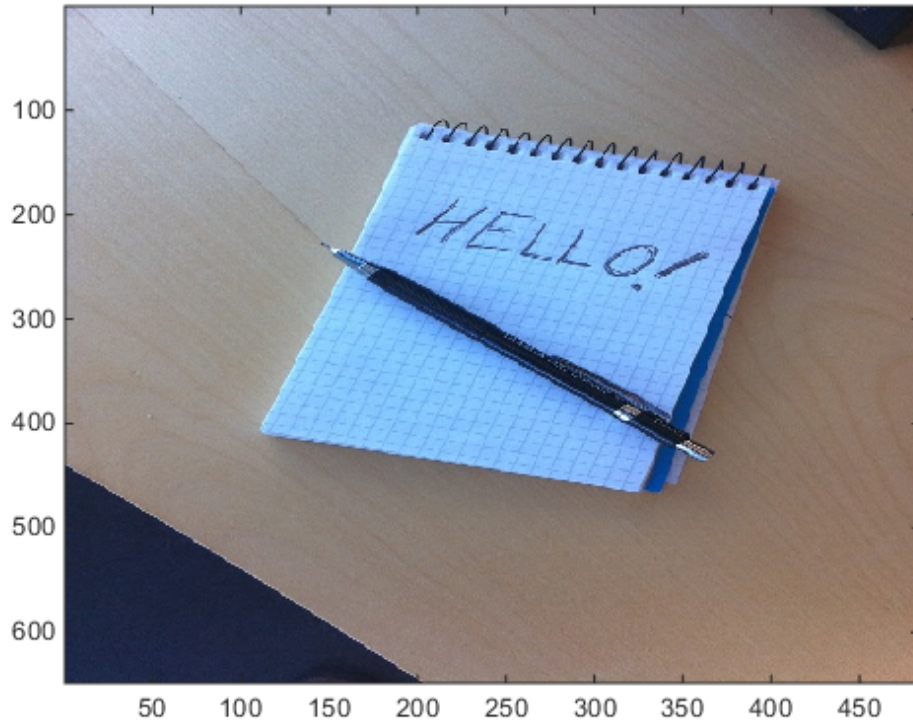
Before generating C code, you should first test the MEX function in MATLAB to ensure that it is functionally equivalent to the original MATLAB code and that no run-time errors occur. By default, 'codegen' generates a MEX function named 'sobel\_mex' in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

### **Read in the Original Image**

Use the standard 'imread' command.

```
im = imread('hello.jpg');  
image(im);
```





### Convert Image to a Grayscale Version

Convert the color image (shown above) to an equivalent grayscale image with normalized values (0.0 for black, 1.0 for white).

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)));
```

### Run the MEX Function (The Sobel Filter)

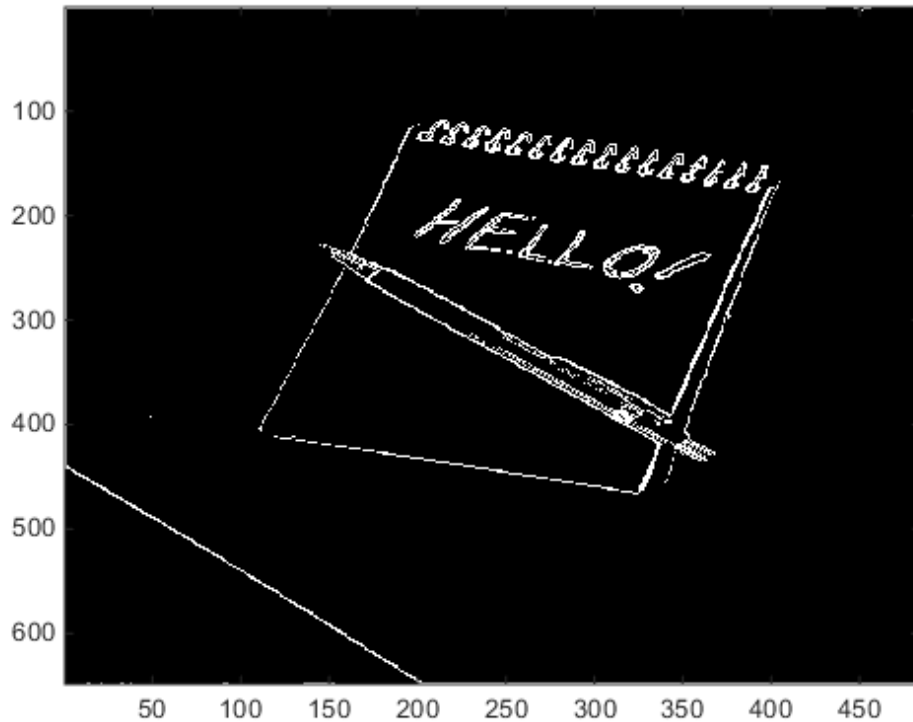
Pass the normalized image and a threshold value.

```
edgeIm = sobel_mex(gray, 0.7);
```

### Display the Result

```
im3 = repmat(edgeIm, [1 1 3]);
```

```
image(im3);
```



### Generate Standalone C Code

```
codegen -config coder.config('lib') sobel
```

Using 'codegen' with the '-config coder.config('lib')' option produces a standalone C library. By default, the code generated for the library is in the folder `codegen/lib/sobel/`

### Inspect the Generated Function

```
type codegen/lib/sobel/sobel.c
```

```
/*  
 * File: sobel.c
```

```
*
* MATLAB Coder version      : 2.7
* C/C++ source code generated on : 04-Sep-2014 08:59:35
*/

/* Include Files */
#include "rt_nonfinite.h"
#include "sobel.h"
#include "sobel_emxutil.h"
#include "sqrt.h"
#include "conv2.h"

/* Function Declarations */
static double rt_roundd_snf(double u);

/* Function Definitions */

/*
 * Arguments      : double u
 * Return Type   : double
 */
static double rt_roundd_snf(double u)
{
    double y;
    if (fabs(u) < 4.503599627370496E+15) {
        if (u >= 0.5) {
            y = floor(u + 0.5);
        } else if (u > -0.5) {
            y = u * 0.0;
        } else {
            y = ceil(u - 0.5);
        }
    } else {
        y = u;
    }

    return y;
}

/*
 * Arguments      : const mxArray_real_T *originalImage
 *                 double threshold
 *                 mxArray_uint8_T *edgeImage
 * Return Type   : void
 */
```

```

*/
void sobel(const emxArray_real_T *originalImage, double threshold,
           emxArray_uint8_T *edgeImage)
{
    emxArray_real_T *H;
    emxArray_real_T *V;
    int b_H;
    int c_H;
    emxInit_real_T(&H, 2);
    emxInit_real_T(&V, 2);

    /* edgeImage = sobel(originalImage, threshold) */
    /* Sobel edge detection. Given a normalized image (with double values) */
    /* return an image where the edges are detected w.r.t. threshold value. */
    conv2(originalImage, H);
    b_conv2(originalImage, V);
    b_H = H->size[0] * H->size[1];
    emxEnsureCapacity((emxArray__common *)H, b_H, (int)sizeof(double));
    b_H = H->size[0];
    c_H = H->size[1];
    c_H *= b_H;
    for (b_H = 0; b_H < c_H; b_H++) {
        H->data[b_H] = H->data[b_H] * H->data[b_H] + V->data[b_H] * V->data[b_H];
    }

    emxFree_real_T(&V);
    b_sqrt(H);
    b_H = edgeImage->size[0] * edgeImage->size[1];
    edgeImage->size[0] = H->size[0];
    edgeImage->size[1] = H->size[1];
    emxEnsureCapacity((emxArray__common *)edgeImage, b_H, (int)sizeof(unsigned
    char));
    c_H = H->size[0] * H->size[1];
    for (b_H = 0; b_H < c_H; b_H++) {
        edgeImage->data[b_H] = (unsigned char)rt_roundd_snf((double)(H->data[b_H] >
        threshold) * 255.0);
    }

    emxFree_real_T(&H);
}

/*
* File trailer for sobel.c
*

```

```
* [EOF]  
*/
```

### **Cleanup**

Remove files and return to original folder

### **Run Command: Cleanup**

```
cleanup
```

## Accelerate MATLAB Algorithms

For many applications, you can generate MEX functions to accelerate MATLAB algorithms. If you have a Fixed-Point Designer license, you can generate MEX functions to accelerate fixed-point MATLAB algorithms. After generating a MEX function, test it in MATLAB to verify that its operation is functionally equivalent to the original MATLAB algorithm. Then compare the speed of execution of the MEX function with that of the MATLAB algorithm. If the MEX function speed is not sufficiently fast, you might improve it using one of the following methods:

- Choosing a different C/C++ compiler.

It is important that you use a C/C++ compiler that is designed to generate high performance code.

---

**Note:** The default MATLAB compiler for Windows 32-bit platforms, `lcc`, is designed to generate code quickly. It is not designed to generate high performance code.

---

- “Modifying MATLAB Code for Acceleration” on page 23-15
- “Control Run-Time Checks” on page 23-16

# Modifying MATLAB Code for Acceleration

## How to Modify Your MATLAB Code for Acceleration

You might improve the efficiency of the generated code using one of the following optimizations:

- “Unroll for-Loops” on page 26-37
- “Inline Code”
- “Eliminate Redundant Copies of Function Inputs”

## Control Run-Time Checks

**In this section...**

“Types of Run-Time Checks” on page 23-16

“When to Disable Run-Time Checks” on page 23-16

“How to Disable Run-Time Checks” on page 23-17

### Types of Run-Time Checks

The code generated for your MATLAB functions includes the following run-time checks and external calls to MATLAB functions.

- Memory integrity checks

These checks detect violations of memory integrity in code generated for MATLAB functions and stop execution with a diagnostic message.

---

**Caution** These checks are enabled by default. Without memory integrity checks, violations result in unpredictable behavior.

---

- Responsiveness checks in code generated for MATLAB functions

These checks enable periodic checks for Ctrl+C breaks in code generated for MATLAB functions. Enabling responsiveness checks also enables graphics refreshing.

---

**Caution** These checks are enabled by default. Without these checks, the only way to end a long-running execution might be to terminate MATLAB.

---

- Extrinsic calls to MATLAB functions

Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information about extrinsic functions, see “Declaring MATLAB Functions as Extrinsic Functions”.

### When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more generated code and slower MEX function execution than generating code with the checks disabled.



Similarly, extrinsic calls are time consuming and increase memory usage and execution time. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster MEX function execution. The following table lists issues to consider when disabling run-time checks and extrinsic calls.

Consider disabling...	Only if...
Memory integrity checks	You have already verified that array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C.
Extrinsic calls	You are using extrinsic calls only for functions that do not affect application results.

## How to Disable Run-Time Checks

You can disable run-time checks explicitly from the project settings dialog box, the command line, or a MEX configuration dialog box.

### Disabling Run-Time Checks in the Project Settings Dialog Box

- 1 On the MATLAB Coder project **Build** tab, click **More settings**.
- 2 On the **Project Settings** dialog box **Speed** tab, clear **Ensure memory integrity**, **Enable responsiveness to CTRL+C** and **graphics refreshing** or **Extrinsic calls**, as applicable.

### Disabling Run-Time Checks From the Command Line

- 1 In the MATLAB workspace, define the MEX configuration object:
 

```
mexcfg = coder.config('mex');
```
- 2 At the command line, set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties to `false`, as applicable:
 

```
mexcfg.IntegrityChecks = false;
mexcfg.ExtrinsicCalls = false;
mexcfg.ResponsivenessChecks = false;
```

## Algorithm Acceleration Using Parallel for-Loops (parfor)

### In this section...

“Parallel for-Loops (parfor) in Generated Code” on page 23-18

“How parfor-Loops Improve Execution Speed” on page 23-19

“When to Use parfor-Loops” on page 23-19

“When Not to Use parfor-Loops” on page 23-19

“parfor-Loop Syntax” on page 23-20

“parfor Restrictions” on page 23-20

### Parallel for-Loops (parfor) in Generated Code

To potentially accelerate execution, you can generate MEX functions or C/C++ code from MATLAB code that contains parallel for-loops (parfor-loops).

A parfor-loop, like the standard MATLAB for-loop, executes a series of statements (the loop body) over a range of values. Unlike the for-loop, however, the iterations of the parfor-loop can run in parallel on multiple cores on the target hardware.

Running the iterations in parallel might significantly improve execution speed of the generated code. For more information, see “How parfor-Loops Improve Execution Speed” on page 23-19.

---

**Note:** The parallel execution occurs only in generated MEX functions or C/C++ code; not the original MATLAB code. To accelerate your MATLAB code, generate a MEX function from the parfor-loop. Then, call the MEX function from your code. For more information, see “Workflow for Accelerating MATLAB Algorithms”.

---

MATLAB Coder software uses the Open Multiprocessing (OpenMP) application interface to support shared-memory, multicore code generation. If you want distributed parallelism, use the Parallel Computing Toolbox™ product. By default, MATLAB Coder uses up to as many cores as it finds available. If you specify the number of threads to use, MATLAB Coder uses at most that number of cores for the threads, even if additional cores are available. For more information, see parfor.

Because the loop body can execute in parallel on multiple threads, it must conform to certain restrictions. If MATLAB Coder software detects loops that do not conform

to `parfor` specifications, it produces an error. For more information, see “`parfor` Restrictions”.

## How `parfor`-Loops Improve Execution Speed

A `parfor`-loop might provide better execution speed than its analogous `for`-loop because several threads can compute concurrently on the same loop.

Each execution of the body of a `parfor`-loop is called an iteration. The threads evaluate iterations in arbitrary order and independently of each other. Because each iteration is independent, they do not have to be synchronized. If the number of threads is equal to the number of loop iterations, each thread performs one iteration of the loop. If there are more iterations than threads, some threads perform more than one loop iteration.

For example, when a loop of 100 iterations runs on 20 threads, each thread executes five iterations of the loop simultaneously. If your loop takes a long time to run because of the large number of iterations or individual iterations being lengthy, you can reduce the run time significantly using multiple threads. In this example, you might not, however, get 20 times improvement in speed because of parallelization overheads, such as thread creation and deletion.

## When to Use `parfor`-Loops

Use `parfor` when you have:

- Many iterations of a simple calculation. `parfor` divides the loop iterations into groups so that each thread executes one group of iterations.
- A loop iteration that takes a long time to execute. `parfor` executes the iterations simultaneously on different threads. Although this simultaneous execution does not reduce the time spent on an individual iteration, it might significantly reduce overall time spent on the loop.

## When Not to Use `parfor`-Loops

Do not use `parfor` when:

- An iteration of your loop depends on other iterations. Running the iterations in parallel can lead to erroneous results.

To help you avoid using `parfor` when an iteration of your loop depends on other iterations, MATLAB Coder specifies a rigid classification of variables. For more information, see “Classification of Variables in `parfor`-Loops” on page 23-26. If MATLAB Coder detects loops that do not conform to the `parfor` specifications, it does not generate code and produces an error.

Reductions are an exception to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order. For more information, see “Reduction Variables” on page 23-28.

- There are only a few iterations that perform some simple calculations.

---

**Note:** For small number of loop iterations, you might not accelerate execution due to parallelization overheads. Such overheads include time taken for thread creation, data synchronization between threads and thread deletion.

---

## parfor-Loop Syntax

- For a `parfor`-loop, use this syntax:

```
parfor i = InitVal:EndVal  
parfor (i = InitVal:EndVal)
```

- To specify the maximum number of threads, use this syntax:

```
parfor (i = InitVal:EndVal, NumThreads)
```

For more information, see `parfor`.

## parfor Restrictions

- The `parfor` loop does not support the syntax:

```
parfor (i=InitVal:EndVal:Step)  
parfor i=Initval:Endval:Step
```

- You must use a compiler that supports the Open Multiprocessing (OpenMP) application interface. See [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). If you use a compiler that does not support OpenMP, MATLAB Coder treats the `parfor`-loops as `for`-loops. In the generated MEX function or C/C++ code, the loop iterations run on a single thread.

- The type of the loop index must be representable by an integer type on the target hardware. Use a type that does not require a multiword type in the generated code.
- `parfor` for standalone code generation requires the toolchain approach for building executables or libraries. Do not change settings that cause the code generation software to use the template makefile approach. See “Project or Configuration is Using the Template Makefile”.
- Do not use the following constructs in the body of a `parfor` loop:

- **Nested parfor-Loops**

You can have a `parfor` loop inside another `parfor`-loop. However, the inner `parfor` loop will be executed on a single thread as an ordinary `for`-loop.

Inside a `parfor` loop, you can call a function that contains another `parfor`-loop.

- **Break and Return statements**

You cannot use `break` or `return` statements inside a `parfor`-loop.

- **Global and persistent variables**

You cannot use persistent variables in the body of a `parfor` loop. However, inside the loop, you can call a function that uses persistent variables. In the generated MEX function or C/C++ code, each thread maintains its own copy of the persistent variables. The copies of the persistent variables for the serial thread and each of the parallel threads are independent of each other.

- **Reductions on MATLAB classes**

You cannot use reductions on MATLAB classes inside a `parfor`-loop.

- **Reductions on char variables**

You cannot use reductions on `char` variables inside a `parfor`-loop.

For example, you cannot generate C code for the following MATLAB code:

```
c = char(0);
parfor i=1:10
    c = c + char(1);
end
```

In the `parfor`-loop, MATLAB makes `c` a double. For code generation, `c` cannot change type.

- **Reductions using external C code**

You cannot use `coder.ceval` in reductions inside a `parfor`-loop. For example, you cannot generate code for the following `parfor`-loop:

```
parfor i=1:4
    y=coder.ceval('myCFcn',y,i);
end
```

Instead, write a local function that calls the C code using `coder.ceval` and call this function in the `parfor`-loop. For example:

```
parfor i=1:4
    y = callMyCFcn(y,i);
end
...
function y = callMyCFcn(y,i)
    y = coder.ceval('mCyFcn', y , i);
end
```

- **Extrinsic function calls**

You cannot call extrinsic functions using `coder.extrinsic` inside a `parfor`-loop. Calls to functions that contain extrinsic calls result in a run-time error.

- **Inlining functions**

MATLAB Coder does not inline functions into `parfor`-loops, including functions that use `coder.inline('always')`.

- **Unrolling loops**

You cannot use `coder.unroll` inside a `parfor`-loop.

If a loop is unrolled inside a `parfor`-loop, MATLAB Coder cannot classify the variable. For example:

```
for j=coder.unroll(3:6)
    y(i,j)=y(i,j)+i+j;
end
```

This code is unrolled to:

```
y(i,3)=y(i,3)+i+3;
```

```
...
```

```
y(i,6)=y(i,6)+i+6;
```

In the unrolled code, MATLAB Coder cannot classify the variable `y` because `y` is indexed in different ways inside the `parfor`-loop.

MATLAB Coder does not support variables that it cannot classify. For more information, see “Classification of Variables in `parfor`-Loops” on page 23-26.

- **varargin/varargout**

You cannot use `varargin` or `varargout` inside a `parfor`-loop.

## Control Compilation of parfor-Loops

By default, MATLAB Coder generates code that can run the `parfor`-loop on multiple threads. To treat the `parfor`-loops as for-loops that run on a single thread, disable `parfor`:

- By using the `codegen` function with `-O disable:openmp` option at the command line.
- By setting **Enable OpenMP library if possible** to **No** under **All Settings** tab in the **Project Settings** dialog box.

### When to Disable parfor

Disable `parfor` if you want to:

- Compare the execution times of the serial and parallel versions of the generated code.
- Investigate failures. If the parallel version of the generated code fails, disable `parfor` and generate a serial version to facilitate debugging.
- Use C compilers that do not support OpenMP.



## Reduction Assignments in parfor-Loops

### What are Reduction Assignments?

Reduction assignments, or *reductions*, are an exception to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the loop iterations together, but is independent of the iteration order. For a list of supported reduction variables see “Reduction Variables” on page 23-28.

### Multiple Reductions in a parfor-Loop

You can perform the same reduction assignment multiple times within a `parfor`-loop provided that you use the same data type each time.

For example, in the following `parfor`-loop, `u(i)` and `v(i)` must be the same type.

```
parfor i = 1:10;
    X = X + u(i);
    X = X + v(i);
end
```

Similarly, the following example is valid provided that `u(i)` and `v(i)` are the same type.

```
parfor i=1:10
    r = foo(r,u(i));
    r = foo(r,v(i));
end
```

## Classification of Variables in parfor-Loops

In this section...
“Overview” on page 23-26
“Sliced Variables” on page 23-27
“Broadcast Variables” on page 23-28
“Reduction Variables” on page 23-28
“Temporary Variables” on page 23-33

### Overview

MATLAB Coder classifies variables inside a `parfor`-loop into one of the categories in the following table. It does not support variables that it cannot classify. If a `parfor`-loop contains variables that cannot be uniquely categorized or if a variable violates its category restrictions, the `parfor`-loop generates an error.

Classification	Description
Loop	Serves as a loop index for arrays
Sliced	An array whose segments are operated on by different iterations of the loop
Broadcast	A variable defined before the loop whose value is used inside the loop, but not assigned inside the loop
Reduction	Accumulates a value across iterations of the loop, regardless of iteration order
Temporary	A variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop

Each of these variable classifications appears in this code fragment:

```
a=0;
c=pi;
z=0;
r=rand(1,10);
parfor i=1:10
    a=i;    % 'a' is a temporary variable
    z=z+i; % 'z' is a reduction variable
    b(i)=r(i); % 'b' is a sliced output variable;
```

```

                                % 'r' a sliced input variable
if i<=c % 'c' is a broadcast variable
    d=2*a; % 'd' is a temporary variable
end
end
end

```

## Sliced Variables

A *sliced variable* is one whose value can be broken up into segments, or *slices*, which are then operated on separately by different threads. Each iteration of the loop works on a different slice of the array.

In the next example, a slice of **A** consists of a single element of that array:

```

parfor i = 1:length(A)
    B(i) = f(A(i));
end

```

### Characteristics of a Sliced Variable

A variable in a `parfor`-loop is sliced if it has the following characteristics:

- **Type of First-Level Indexing** — The first level of indexing is parentheses, `()`.
- **Fixed Index Listing** — Within the first-level parenthesis, the list of indices is the same for all occurrences of a given variable.
- **Form of Indexing** — Within the list of indices for the variable, exactly one index involves the loop variable.
- **Shape of Array** — In assigning to a sliced variable, the right-hand side of the assignment is not `[]` or `''` (these operators indicate deletion of elements).

*Type of First-Level Indexing.* For a sliced variable, the first level of indexing is enclosed in parentheses, `()`. For example, `A(...)`. If you reference a variable using dot notation, `A.x`, the variable is not sliced.

Variable **A** on the left is not sliced; variable **A** on the right is sliced:

```

A.q(i,12)                A(i,12).q

```

*Fixed Index Listing.* Within the first-level parentheses of a sliced variable's indexing, the list of indices is the same for all occurrences of a given variable.

Variable **B** on the left is not sliced because **B** is indexed by `i` and `i+1` in different places. Variable **B** on the right is sliced.

```
parfor i = 1:10
    B(i) = B(i+1) + 1;
end
```

```
parfor i = 1:10
    B(i+1) = B(i+1) + 1;
end
```

*Form of Indexing.* Within the list of indices for a sliced variable, one index is of the form  $i$ ,  $i+k$ ,  $i-k$ ,  $k+i$ , or  $k-i$ .

- $i$  is the loop variable.
- $k$  is a constant or a simple (nonindexed) variable.
- Every other index is a constant, a simple variable, colon, or `end`.

When you use other variables along with the loop variable to index an array, you cannot set these variables inside the loop. These variables are constant over the execution of the entire `parfor` statement. You cannot combine the loop variable with itself to form an index expression.

In the following examples,  $i$  is the loop variable,  $j$  and  $k$  are nonindexed variables.

Variable A Is Not Sliced	Variable A Is Sliced
<code>A(i+f(k),j,:,3)</code>	<code>A(i+k,j,:,3)</code>
<code>A(i,20:30,end)</code>	<code>A(i,:,end)</code>
<code>A(i,:,s.field1)</code>	<code>A(i,:,k)</code>

*Shape of Array.* A sliced variable must maintain a constant shape. In the following examples, the variable `A` is not sliced:

```
A(i,:) = [];
A(end + 1) = i;
```

## Broadcast Variables

A *broadcast variable* is a variable other than the loop variable or a sliced variable that is not modified inside the loop.

## Reduction Variables

A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order.

This example shows a `parfor`-loop that uses a scalar reduction assignment. It uses the reduction variable `x` to accumulate a sum across 10 iterations of the loop. The execution order of the iterations on the threads does not matter.

```
x = 0;
parfor i = 1:10
    x = x + i;
end
x
```

Where `expr` is a MATLAB expression, reduction variables appear on both sides of an assignment statement.

<code>X = X + expr</code>	<code>X = expr + X</code>
<code>X = X - expr</code>	See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 23-32
<code>X = X .* expr</code>	<code>X = expr .* X</code>
<code>X = X * expr</code>	<code>X = expr * X</code>
<code>X = X &amp; expr</code>	<code>X = expr &amp; X</code>
<code>X = X   expr</code>	<code>X = expr   X</code>
<code>X = min(X, expr)</code>	<code>X = min(expr, X)</code>
<code>X = max(X, expr)</code>	<code>X = max(expr, X)</code>
<code>X=f(X, expr)</code> Function <code>f</code> must be a user-defined function.	<code>X = f(expr, X)</code> See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 23-32

Each of the allowed statements is referred to as a *reduction assignment*. A reduction variable can appear only in assignments of this type.

The following example shows a typical usage of a reduction variable `X`:

```
X = ...;           % Do some initialization of X
parfor i = 1:n
    X = X + d(i);
end
```

This loop is equivalent to the following, where each `d(i)` is calculated by a different iteration:

$$X = X + d(1) + \dots + d(n)$$

If the loop were a regular `for`-loop, the variable `X` in each iteration would get its value either before entering the loop or from the previous iteration of the loop. However, this concept does not apply to `parfor`-loops.

In a `parfor`-loop, the value of `X` is not updated directly inside each thread. Rather, additions of `d(i)` are done in each thread, with `i` ranging over the subset of `1:n` being performed on that thread. The software then accumulates the results into `X`.

Similarly, the reduction:

```
r=r<op> x(i)
```

is equivalent to:

```
r=r<op>x(1) ] <op>x(2) . . . <op>x(n)
```

The operation `<op>` is first applied to `x(1) . . . x(n)`, then the partial result is applied to `r`.

If operation `<op>` takes two inputs, it should meet one of the following criteria:

- Take two arguments of `typeof(x(i))` and return `typeof(x(i))`
- Take one argument of `typeof(r)` and one of `typeof(x(i))` and return `typeof(r)`

### Rules for Reduction Variables

#### Use the same reduction function or operation in all reduction assignments

For a reduction variable, you must use the same reduction function or operation in all reduction assignments for that variable. In the following example, the `parfor`-loop on the left is not valid because the reduction assignment uses `+` in one instance, and `*` in another.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
<pre>parfor i = 1:n     if A &gt; 5*k         A = A + 1;     else         A = A * 2;     end</pre>	<pre>parfor i = 1:n     if A &gt; 5*k         A = A * 3;     else         A = A * 2;     end</pre>

#### Restrictions on reduction function parameter and return types

A reduction `r=r<op> x(i)`, should take arguments of `typeof(x(i))` and return `typeof(x(i))` or take arguments of `typeof(r)` and `typeof(x(i))` and return `typeof(r)`.

In the following example, in the invalid loop, `r` is a fixed-point type and `2` is not. To fix this issue, cast `2` to be the same type as `r`.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
<pre>function r = fiops(in) r=fi(in,'WordLength',20,... 'FractionLength',14,... 'SumMode','SpecifyPrecision',... 'SumWordLength',20,... 'SumFractionLength',14,... 'ProductMode','SpecifyPrecision',... 'ProductWordLength',20,... 'ProductFractionLength',14); parfor i = 1:10     r = r*2; end</pre>	<pre>r=fi(in,'WordLength',20,... 'FractionLength',14,... 'SumMode','SpecifyPrecision',... 'SumWordLength',20,... 'SumFractionLength',14,... 'ProductMode','SpecifyPrecision',... 'ProductWordLength',20,... 'ProductFractionLength',14); T = r.numerictype; F = r.fimath; parfor i = 1:10     r = r*fi(2,T,F); end</pre>

In the following example, the reduction function `fCN` is invalid because it does not handle the case when input `u` is fixed point. (The `+` and `*` operations are automatically polymorphic.) You must write a polymorphic version of `fCN` to handle the expected input types.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
<pre>function [y0, y1, y2] = pfuserfcn(u)     y0 = 0;     y1 = 1;     [F, N] = fiprops();     y2 = fi(1,N,F);     parfor (i=1:numel(u),12)         y0 = y0 + u(i);         y1 = y1 * u(i);         y2 = fcn(y2, u(i));     end end  function y = fcn(u, v)     y = u * v; end</pre>	<pre>function [y0, y1, y2] = pfuserfcn(u)     y0 = 0;     y1 = 1;     [F, N] = fiprops();     y2 = fi(1,N,F);     parfor (i=1:numel(u),12)         y0 = y0 + u(i);         y1 = y1 * u(i);         y2 = fcn(y2, u(i));     end end % fcn handles inputs of type double % and fi function y = fcn(u, v)     if isa(u,'double')         y = u * v;     else         [F, N] = fiprops();</pre>

Invalid Use of Reduction Variable	Valid Use of Reduction Variable
	<pre> y = u * fi(v,N,F); end end  function [F, N] = fiprops()     N = numerictype(1,96,30);     F = fimath('ProductMode',...               'SpecifyPrecision',...               'ProductWordLength',96); end </pre>

### Reduction Assignments, Associativity, and Commutativity of Reduction Functions

*Reduction Assignments.* MATLAB Coder does not allow reduction variables to be read anywhere in the `parfor`-loop except in reduction statements. In the following example, the call `foo(r)` after the reduction statement `r=r+i` causes the loop to be invalid.

```

function r = temp %#codegen
    r = 0;
    parfor i=1:10
        r = r + i;
        foo(r);
    end
end

```

*Associativity in Reduction Assignments.* If you use a user-defined function `f` in the definition of a reduction variable, to get deterministic behavior of `parfor`-loops, the reduction function `f` must be associative.

---

**Note:** If `f` is not associative, MATLAB Coder does not generate an error. You must write code that meets this recommendation.

---

To be associative, the function `f` must satisfy the following for all `a`, `b`, and `c`:

$$f(a, f(b, c)) = f(f(a, b), c)$$

*Commutativity in Reduction Assignments.* Some associative functions, including `+`, `.`, `min`, and `max`, are also commutative. That is, they satisfy the following for all `a` and `b`:

$$f(a, b) = f(b, a)$$



The function  $f$  of a reduction assignment must be commutative. If  $f$  is not commutative, different executions of the loop might result in different answers.

Unless  $f$  is a known noncommutative built-in, the software assumes that it is commutative.

## Temporary Variables

A *temporary variable* is a variable that is the target of a direct, nonindexed assignment, but is not a reduction variable. In the following `parfor`-loop, `a` and `d` are temporary variables:

```
a = 0;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;           % Variable a is temporary
    z = z + i;
    if i <= 5
        d = 2*a;    % Variable d is temporary
    end
end
```

In contrast to the behavior of a `for`-loop, before each iteration of a `parfor`-loop, MATLAB Coder effectively clears temporary variables. Because the iterations must be independent, the values of temporary variables cannot be passed from one iteration of the loop to another. Therefore, temporary variables must be set inside the body of a `parfor`-loop, so that their values are defined separately for each iteration.

A temporary variable in the context of the `parfor` statement is different from a variable with the same name that exists outside the loop.

## Uninitialized Temporaries

Because temporary variables are cleared at the beginning of every iteration, MATLAB Coder can detect certain cases in which an iteration through the loop uses the temporary variable before it is set in that iteration. In this case, MATLAB Coder issues a static error rather than a run-time error, because there is little point in allowing execution to proceed if a run-time error will occur. For example, suppose you write:

```
b = true;
parfor i = 1:n
```

```
    if b && some_condition(i)
        do_something(i);
        b = false;
    end
    ...
end
```

This loop is acceptable as an ordinary `for`-loop, but as a `parfor`-loop, `b` is a temporary variable because it occurs directly as the target of an assignment inside the loop. Therefore, it is cleared at the start of each iteration, so its use in the condition of the `if` is uninitialized. (If you change `parfor` to `for`, the value of `b` assumes sequential execution of the loop, so that `do_something(i)` is executed for only the lower values of `i` until `b` is set `false`.)

## Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)

This example shows how to generate a MEX function for a MATLAB algorithm that contains a `parfor`-loop.

- 1 Write a MATLAB function that contains a `parfor`-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
    a(i,:)=real(fft(r(i,:)));
end
```

- 2 Generate a MEX function for `test_parfor`. At the MATLAB command line, enter:

```
codegen test_parfor
codegen generates a MEX function, test_parfor_mex, in the current folder.
```

- 3 Run the MEX function. At the MATLAB command line, enter:

```
test_parfor_mex
```

Because you did not specify the maximum number of threads to use, the generated MEX function executes the loop iterations in parallel on the maximum number of available cores.

## Specify Maximum Number of Threads in parfor-Loops

This example shows how to specify the maximum number of threads to use for a `parfor`-loop. Because you specify the maximum number of threads to use, the generated MEX function executes the loop iterations in parallel on as many cores as available, up to the maximum number that you specify. If you specify more threads than there are cores available, the MEX function uses the available cores.

- 1 Write a MATLAB function, `specify_num_threads`, that uses one input to specify the maximum number of threads to execute a `parfor`-loop in the generated MEX function. For example:

```
function y = specify_num_threads(u) %#codegen
    y = ones(1,100);
    % u specifies maximum number of threads
    parfor (i = 1:100,u)
        y(i) = i;
    end
end
```

- 2 Generate a MEX function for `specify_num_threads`. Use `-args {0}` to specify that input `u` is a scalar double. Use `-report` to generate a code generation report. At the MATLAB command line, enter:

```
codegen -report specify_num_threads -args {0}
codegen generates a MEX function, specify_num_threads_mex, in the current folder.
```

- 3 Run the MEX function, specifying that it try to run in parallel on four threads. At the MATLAB command line, enter:

```
specify_num_threads_mex(4)
```

The generated MEX function runs on up to four cores. If less than four cores are available, the MEX function runs on the maximum number of cores available at the time of the call.

## Troubleshooting parfor-Loops

### What Causes Errors With Global Structures in Parallel Regions?

- The body of the `parfor`-loop contains `global` or `persistent` variable declarations. `parfor` does not support such declarations.
- Local variables use more memory than the specified stack size. When this situation occurs, MATLAB Coder moves the local variables to a static area. It accesses them using a pointer in a global structure. MATLAB Coder does not support global structures in parallel regions. If possible, increase the stack size.

If you use	Action	For More Information
A MATLAB Coder project	In the <b>Project Settings</b> dialog box <b>All Settings</b> tab, under <b>Advanced</b> , set the <b>Inline stack limit</b> to the new limit.	“Specifying Build Configuration Parameters in the Project Settings Dialog Box” on page 19-26
<code>codegen</code> at the command line with a configuration object	Create a <code>coder.CodeConfig</code> or <code>coder.EmbeddedCodeConfig</code> object, as applicable. Set the <code>InlineStackLimit</code> parameter to the new limit.	“Specifying Build Configuration Parameters at the Command Line Using Configuration Objects” on page 19-26

### Compiler Does Not Support OpenMP

The MATLAB Coder software uses the Open Multiprocessing (OpenMP) application interface to support shared-memory, multicore code generation. To generate a loop that runs in parallel on shared-memory, multicore platforms, you must have a compiler that supports OpenMP. OpenMP is enabled by default. If your compiler does not support OpenMP, MATLAB Coder generates a warning.

Install a compiler that supports OpenMP. See [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/).

## Accelerating Simulation of Bouncing Balls

This example shows how to accelerate MATLAB algorithm execution using a generated MEX function. It uses the 'codegen' command to generate a MEX function for a complicated application that uses multiple MATLAB files. You can use 'codegen' to check that your MATLAB code is suitable for code generation and, in many cases, to accelerate your MATLAB algorithm. You can run the MEX function to check for run-time errors.

### Prerequisites

There are no prerequisites for this example.

### Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will contain only the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), change your working folder.

### Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_bouncing_balls');
```

### About the 'run\_balls' Function

The run\_balls.m function takes a single input to specify the number of bouncing balls to simulate. The simulation runs and plots the balls bouncing until there is no energy left and returns the state (positions) of all the balls.

```
type run_balls
```

```
% balls = run_balls(n)
% Given 'n' number of balls, run a simulation until the balls come to a
% complete halt (or when the system has no more kinetic energy).
function balls = run_balls(n) %#codegen

coder.extrinsic('fprintf');

% Copyright 2010-2013 The MathWorks, Inc.

% Seeding the random number generator will guarantee that we get
% precisely the same simulation every time we call this function.
old_settings = rng(1283,'V4');
```

```

% The 'cdata' variable is a matrix representing the colordata bitmap which
% will be rendered at every time step.
cdata = zeros(400,600,'uint8');

% Setup figure windows
im = setup_figure_window(cdata);

% Get the initial configuration for 'n' balls.
balls = initialize_balls(cdata, n);

energy = 2; % Something greater than 1
iteration = 1;
while energy > 1
    % Clear the bitmap
    cdata(:, :) = 0;
    % Apply one iteration of movement
    [cdata, balls, energy] = step_function(cdata, balls);
    % Render the current state
    cdata = draw_balls(cdata, balls);
    iteration = iteration + 1;
    if mod(iteration, 10) == 0
        fprintf(1, 'Iteration %d\n', iteration);
    end
    refresh_image(im, cdata);
end
fprintf(1, 'Completed iterations: %d\n', iteration);

% Restore RNG settings.
rng(old_settings);

```

### Generate the MEX Function

First, generate a MEX function using the command `codegen` followed by the name of the MATLAB file to compile. Pass an example input (`-args 0`) to indicate that the generated MEX function will be called with an input of type double.

```
codegen run_balls -args 0
```

The `'run_balls'` function calls other MATLAB functions, but you need to specify only the entry-point function when calling `'codegen'`.

By default, `'codegen'` generates a MEX function named `'run_balls_mex'` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

### Compare Results

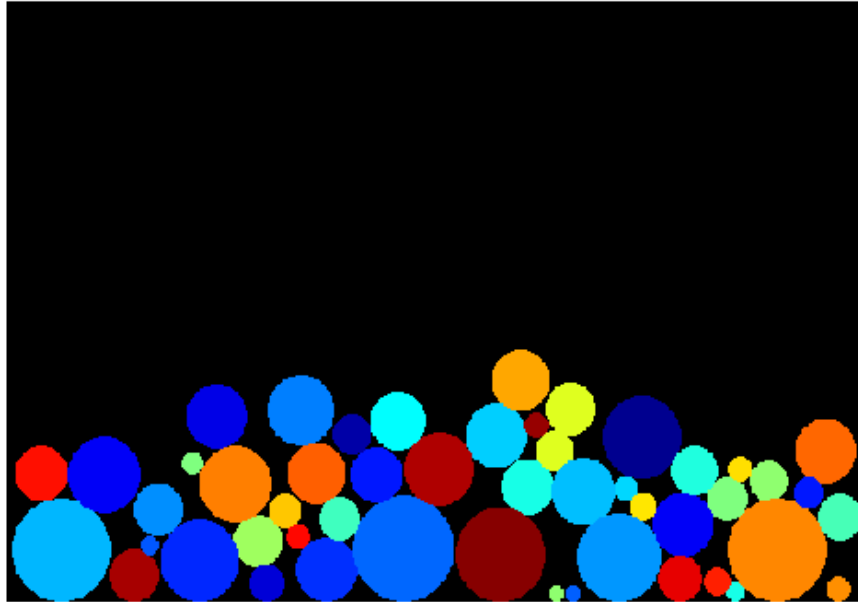
Run and time the original 'run\_balls' function followed by the generated MEX function.

```
tic, run_balls(50); t1 = toc;  
tic, run_balls_mex(50); t2 = toc;
```

```
Iteration 10  
Iteration 20  
Iteration 30  
Iteration 40  
Iteration 50  
Iteration 60  
Iteration 70  
Iteration 80  
Iteration 90  
Iteration 100  
Iteration 110  
Iteration 120  
Iteration 130  
Iteration 140  
Iteration 150  
Iteration 160  
Iteration 170  
Iteration 180  
Iteration 190  
Iteration 200  
Iteration 210  
Iteration 220  
Iteration 230  
Iteration 240  
Iteration 250  
Iteration 260  
Iteration 270  
Iteration 280  
Completed iterations: 281  
Iteration 10  
Iteration 20  
Iteration 30  
Iteration 40  
Iteration 50  
Iteration 60  
Iteration 70  
Iteration 80  
Iteration 90
```



```
Iteration 100  
Iteration 110  
Iteration 120  
Iteration 130  
Iteration 140  
Iteration 150  
Iteration 160  
Iteration 170  
Iteration 180  
Iteration 190  
Iteration 200  
Iteration 210  
Iteration 220  
Iteration 230  
Iteration 240  
Iteration 250  
Iteration 260  
Iteration 270  
Iteration 280  
Completed iterations: 281
```



Estimated speed up is:

```
fprintf(1, 'Speed up: x ~%2.1f\n', t1/t2);
```

```
Speed up: x ~5.6
```

### **Clean Up**

Remove files and return to original folder

### **Run Command: Cleanup**

```
cleanup
```

# Calling C/C++ Functions from Generated Code

---

- “External Function Calls from Generated Code” on page 24-2
- “Call External Functions Using `coder.ceval`” on page 24-6
- “Return Multiple Values from C Functions” on page 24-8
- “How MATLAB Coder Infers C/C++ Data Types” on page 24-9

## External Function Calls from Generated Code

In this section...
“Calling External Functions from Generated Code” on page 24-2
“Why Call External Functions from Generated Code?” on page 24-2
“How To Call External Functions” on page 24-2
“Pass Arguments by Reference to External Functions” on page 24-3
“Manipulate C Data” on page 24-4

### Calling External Functions from Generated Code

You can call external functions from generated code. The external functions must have a C programming interface. The code generation software provides functions for:

- Calling external functions from generated code.
- Passing arguments by reference to external code.
- Manipulating C/C++ data.

By using these functions, you gain unrestricted access to external code. Misuse of these functions or errors in your code can destabilize MATLAB when generating MEX functions.

### Why Call External Functions from Generated Code?

Call external functions from generated code when you want to:

- Use legacy code.
- Use your own optimized functions instead of generated code.
- Interface your libraries and hardware with MATLAB functions.

### How To Call External Functions

To call external functions, use one of the following methods:

- The `coder.ceval` function in your MATLAB code. `coder.ceval` passes function input and output arguments to C/C++ functions by value or by reference.

- The `coder.ExternalDependency` class to define methods that call the functions. These methods use the `coder.ceval` function. In your MATLAB code, use these methods to call external functions.

Define the called functions in external C/C++ source files, object files, or libraries. You must then include C/C++ source files, libraries, object files, and header files in the build configuration. See “Specify External File Locations”.

## Pass Arguments by Reference to External Functions

By default, `coder.ceval` passes arguments by value to the C/C++ function whenever C/C++ supports passing arguments by value. You can pass MATLAB variables as arguments by reference to external C/C++ functions with the following constructs:

- `coder.ref` — pass value by reference.
- `coder.rref` — pass read-only value by reference.
- `coder.wref` — pass write-only value by reference.

These constructs offer the following benefits:

- Passing values by reference optimizes memory use.

When you pass arguments by value, MATLAB Coder passes a copy of the value of each argument to the C/C++ function to preserve the original values. When you pass arguments by reference, MATLAB Coder does not copy values. If you need to pass large matrices to the C/C++ function, the memory savings can be significant.

Passing write-only values by reference allows you to return multiple outputs.

Use `coder.wref` to return multiple outputs from your C/C++ function, including arrays and matrices. Otherwise, the C/C++ function can return only a single scalar value through its `return` statement.

Do not store pointers that you pass to C/C++ functions because MATLAB Coder optimizes the code based on the assumption that you do not store the addresses of these variables. Storing the addresses might invalidate our optimizations leading to incorrect behavior. For example, if a MATLAB function passes a pointer to an array using `coder.ref`, `coder.rref`, or `coder.wref`, then the C/C++ function can modify the data in the array—but you should not store the pointer for future use.

When you pass arguments by reference using `coder.rref`, `coder.wref`, and `coder.ref`, the corresponding C/C++ function signature must declare these variables

as pointers of the same data type. Otherwise, the C/C++ compiler generates a type mismatch error.

For example, suppose your MATLAB function calls an external C function `ctest`:

```
function y = fcn()
u = pi;

y = 0;
y = coder.ceval('ctest',u);
```

Now suppose the C function signature is:

```
double ctest(double *a)
```

When you compile the code, you get a type mismatch error because `coder.ceval` calls `ctest` with an argument of type `double` when `ctest` expects a pointer to a double-precision, floating-point value.

Match the types of arguments in `coder.ceval` with their counterparts in the C function. For instance, you can fix the error in the previous example by passing the argument by reference:

```
y = coder.ceval('ctest', coder.rref(u));
```

You can pass a reference to an element of a matrix. For example, to pass the second element of the matrix `v`, you can use the following code:

```
y = coder.ceval('ctest', coder.ref(v(1,2)));
```

## Manipulate C Data

The construct `coder.opaque` allows you to manipulate C/C++ data that a MATLAB function does not recognize. You can store the opaque data in a variable or structure field and pass it to, or return it from, a C/C++ function using `coder.ceval`.

### Declaring Opaque Data

The following example uses `coder.opaque` to declare a variable `f` as a `FILE *` type.

```
% This example returns its own source code by using
% fopen/fread/fclose.
function buffer = filetest
%#codegen
```

```
% Declare 'f' as an opaque type 'FILE *'
f = coder.opaque('FILE *', 'NULL');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));

% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
    % By default, MATLAB converts constant values
    % to doubles in generated code
    % so explicit type conversion to in32 is inserted.
    n = coder.ceval('fread', coder.ref(buffer(i)), int32(1), ...
        int32(numel(buffer)), f);
    i = i + n;
end
coder.ceval('fclose',f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB string
function y = cstring(x)
    y = [x char(0)];

% Remove character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
    j = 1;
    for i = 1:numel(buffer)
        if buffer(i) ~= char(13)
            buffer(j) = buffer(i);
            j = j + 1;
        end
    end
    buffer(i) = 0;
```

## Call External Functions Using `coder.ceval`

**In this section...**

“Workflow for Calling External Functions” on page 24-6

“Best Practices for Calling External Code from Generated Code” on page 24-7

### Workflow for Calling External Functions

To call external C/C++ functions from generated code:

- 1 Write your C/C++ functions in external source files or libraries.
- 2 Create header files, if required.

The header file defines the data types used by the C/C++ functions that MATLAB Coder generates in code, as described in “Mapping MATLAB Types to C/C++ Types” on page 24-9.

---

**Tip** One way to add these type definitions is to include the header file `tmwtypes.h`, which defines general data types supported by MATLAB. This header file is in `matlabroot/extern/include`. Check the definitions in `tmwtypes.h` to determine if they are compatible with your target. If not, define these types in your own header files.

---

- 3 In your MATLAB function, add calls to `coder.ceval` to invoke your external C/C++ functions.

You need one `coder.ceval` statement for each call to a C/C++ function. In your `coder.ceval` statements, use `coder.ref`, `coder.rref`, and `coder.wref` constructs as required (see “Pass Arguments by Reference to External Functions” on page 24-3).

- 4 Include the custom C/C++ functions in the build. See “Specify External File Locations”.
- 5 Check for compilation warnings about data type mismatches.

Perform this check so that you catch type mismatches between C/C++ and MATLAB (see “How MATLAB Coder Infers C/C++ Data Types” on page 24-9).

- 6 Generate code and fix errors.



7 Run your application.

## Best Practices for Calling External Code from Generated Code

The following are recommended practices when calling C/C++ code from generated code.

- **Start small.** — Create a test function and learn how `coder.ceval` and its related constructs work.
- **Use separate files.** — Create a file for each C/C++ function that you call. Make sure that you call the C/C++ functions with suitable types.
- In a header file, declare a function prototype for each function that you call, and include this header file in the generated code. For more information, see “Specify External File Locations”.

## Return Multiple Values from C Functions

The C language restricts functions from returning multiple outputs; instead, they return only a single, scalar value. The constructs `coder.ref` and `coder.wref` allow MATLAB functions to exchange multiple outputs with the external C functions that they call.

For example, suppose you write a MATLAB function `foo` that takes two inputs `x` and `y` and returns three outputs `a`, `b`, and `c`. In MATLAB, you call this function as follows:

```
[a, b, c] = foo (x, y)
```

If you rewrite `foo` as a C function, you cannot return `a`, `b`, and `c` through the `return` statement. You can create a C function with multiple pointer type input arguments, and pass the output parameters by reference. For example:

```
foo(double x, double y, double *a, double *b, double *c)
```

Then you can call the C function with multiple outputs from a MATLAB function using `coder.wref` constructs:

```
coder.ceval ('foo', x, y, ...  
            coder.wref(a), coder.wref(b), coder.wref(c));
```

Similarly, suppose that one of the outputs `a` is also an input argument. In this case, create a C function with multiple pointer type input arguments, and pass the output parameters by reference. For example:

```
foo(double *a, double *b, double *c)
```

Then call the C function from a MATLAB function using `coder.wref` and `coder.rref` constructs:

```
coder.ceval ('foo', coder.rref(a), coder.wref(b), coder.wref(c));
```

## How MATLAB Coder Infers C/C++ Data Types

### In this section...

- “Mapping MATLAB Types to C/C++ Types” on page 24-9
- “Mapping 64-Bit Integer Types to C/C++” on page 24-10
- “Mapping Fixed-Point Types to C/C++” on page 24-11
- “Mapping Arrays to C/C++” on page 24-11
- “Mapping Complex Values to C/C++” on page 24-12
- “Mapping Structures to C/C++ Structures” on page 24-13
- “Mapping Strings to C/C++” on page 24-13
- “Mapping Multiword Types to C/C++” on page 24-14

### Mapping MATLAB Types to C/C++ Types

The C/C++ type associated with a MATLAB variable or expression is based on the following properties:

- Class
- Size
- Complexity

By default, the MATLAB Coder software tries to use built-in C/C++ types in the generated code. If the target hardware supports the built-in C type, the software generates a built-in C type for these MATLAB types.

<code>int8</code>	<code>uint8</code>	<code>double</code>
<code>int16</code>	<code>uint16</code>	<code>single</code>
<code>int32</code>	<code>uint32</code>	<code>char</code>
<code>int64</code>	<code>uint64</code>	

The built-in C/C++ type that the code generation software uses depends on the target hardware. You have the option to use MathWorks C/C++ data types instead of built-in C/C++ types. For information about setting this option, see “Specify Data Type Used in Generated Code”.

The following translation table shows how the MATLAB Coder software maps MATLAB types to MathWorks C/C++ data types.

MATLAB Type	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
int8	int8_T	int8_T *
int16	int16_T	int16_T *
int32	int32_T	int32_T *
int64	See “Mapping 64-Bit Integer Types to C/C++”.	
uint8	uint8_T	uint8_T *
uint16	uint16_T	uint16_T *
uint32	uint32_T	uint32_T *
uint64	See “Mapping 64-Bit Integer Types to C/C++”.	
double	real_T	real_T *
single	real32_T	real32_T *
char	char_T	char *
logical	boolean_T	boolean_T *
fi	numeric_type also influences the C/C++ type. Integer type varies according to the MATLAB fixed-point type, as described in “Mapping Fixed-Point Types to C/C++”.	
struct	The MATLAB Coder software translates structures to C/C++ types field-by-field. See “Mapping Structures to C/C++ Structures” on page 24-13 .	
complex	See “Mapping Complex Values to C/C++” on page 24-12.	
Multiword types	See “Mapping Multiword Types to C/C++” on page 24-14.	

## Mapping 64-Bit Integer Types to C/C++

The C/C++ data type associated with a 64-bit integer MATLAB type depends on the sizes of the integer types on the target hardware. If a type wide enough for a 64-bit type does not exist, then a 64-bit type maps to a multiword type.

By default, MATLAB Coder software tries to map `int64` and `uint64` types to built-in C types. For a multiword type, the software uses a built-in C type for the array in the struct that represents the multiword type. You have the option to use MATLAB C/C++ data types instead of built-in types. The following table shows how 64 bit integer types map to MATLAB C/C++ data types.

MATLAB Type	MATLAB C/C++ Type	Multiword MATLAB C/C++ Type
<code>int64</code>	<code>int64_T</code>	<code>int64m_T</code>
<code>uint64</code>	<code>uint64_T</code>	<code>uint64m_T</code>
<code>complex int64</code>	<code>cint64_T</code>	<code>cint64m_T</code>
<code>complex uint64</code>	<code>cuint64_T</code>	<code>cuint64m_T</code>

See “Mapping Multiword Types to C/C++”.

## Mapping Fixed-Point Types to C/C++

The `numericType` properties of a `fi` object determine the C/C++ data type. By default, the code generation software tries to use built-in C/C++ types. However, you can choose to use MATLAB C/C++ data types instead. The following table shows how the `Signedness`, `WordLength`, and `FractionLength` properties determine the MATLAB C/C++ data type. The MATLAB C/C++ data type is the next larger target word size that can store the fixed-point value, based on its word length. The sign of the integer type matches the sign of the fixed-point type.

Signedness	Word Length	Fraction Length	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
1	16	15	<code>int16_T</code>	<code>int16_T *</code>
1	13	10	<code>int16_T</code>	<code>int16_T *</code>
0	19	15	<code>uint32_T</code>	<code>uint32_T *</code>
1	8	7	<code>int8_T</code>	<code>int8_T *</code>

## Mapping Arrays to C/C++

By default, the code generation software tries to use built-in C/C++ types for arrays in the generated code. However, you can choose to use MATLAB C/C++ data types instead.

The following translation table shows how MATLAB Coder software maps arrays to MATLAB C/C++ data types. In the first column, the arrays are specified by the MATLAB function `zeros`:

`zeros(number of rows, number of columns, data type)`  
 MATLAB array data is laid out in column major order.

Array	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
<code>zeros(10, 5, 'int8')</code>	<code>int8_T</code>	<code>int8_T *</code>
<code>zeros(5, 10, 'int8')</code>	<code>int8_T</code>	<code>int8_T *</code>
<code>zeros(3, 7)</code>	<code>real_T</code>	<code>real_T *</code>
<code>zeros(10, 1, 'single')</code>	<code>real32_T</code>	<code>real32_T *</code>

## Mapping Complex Values to C/C++

The following translation table shows how the MATLAB Coder software infers complex values in generated code.

Complex	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
<code>complex int8</code>	<code>cint8_T</code>	<code>cint8_T *</code>
<code>complex int16</code>	<code>cint16_T</code>	<code>cint16_T *</code>
<code>complex int32</code>	<code>cint32_T</code>	<code>cint32_T *</code>
<code>complex int64</code>	See “Mapping 64-Bit Integer Types to C/C++”.	
<code>complex uint8</code>	<code>cuint8_T</code>	<code>cuint8_T *</code>
<code>complex uint16</code>	<code>cuint16_T</code>	<code>cuint16_T *</code>
<code>complex uint32</code>	<code>cuint32_T</code>	<code>cuint32_T *</code>
<code>complex uint64</code>	See “Mapping 64-Bit Integer Types to C/C++”.	
<code>complex double</code>	<code>creal_T</code>	<code>creal_T *</code>
<code>complex single</code>	<code>creal32_T</code>	<code>creal32_T *</code>

The MATLAB Coder software defines each complex value as a structure with a real component `re` and an imaginary component `im`, as in this example from `tmwtypes.h`:

```
typedef struct {  
    real32_T re; /* Real component*/  
    real32_T im; /* Imaginary component*/  
} creal32_T;
```

MATLAB Coder uses the names `re` and `im` in generated code to represent the components of complex numbers. For example, suppose you define a variable `x` of type `creal32_T`. The generated code references the real component as `x.re` and the imaginary component as `x.im`.

If your C/C++ library requires a different representation, you can define your own versions of MATLAB Coder complex types. However, you *must* use the names `re` for the real components and `im` for the imaginary components in your definitions.

The MATLAB Coder software represents a matrix of complex numbers as an array of structures.

## Mapping Structures to C/C++ Structures

The MATLAB Coder software translates structures to C/C++ types field-by-field. The order of the field items is preserved as the order in MATLAB. To control the name of the generated C/C++ structure type, or provide a definition, use the `coder.cstructname` function.

---

**Note:** If you are not using dynamic memory allocation, arrays in structures translate into single-dimension arrays, not pointers.

---

## Mapping Strings to C/C++

The MATLAB Coder software translates MATLAB strings to C/C++ character matrices. You cannot use character matrices as substitutes for C/C++ strings because they are not null terminated. You can terminate a MATLAB string with a null character by appending a zero to the end of the string: `['sample string' 0]`. A single character translates to a C/C++ `char` type, not a C/C++ string.

---

**Caution** Failing to null-terminate your MATLAB strings might cause C/C++ code to crash without compiler errors or warnings.

---

## Mapping Multiword Types to C/C++

The MATLAB Coder software translates multiword types to structure types that contain an array of integers. The array dimensions depend on the size of the widest integer type on the target hardware. For example, for a 128-bit fixed-point type, if the widest integer type on the target hardware is 32-bits, the software generates a structure with an array of four 32-bit integers.

```
typedef struct
{
    unsigned int  chunks[4];
} uint128m_T;
```

If the widest integer type on the target hardware is `long` with a size of 64-bits, MATLAB Coder generates a structure with an array of two 64-bit long integers.

```
typedef struct
{
    unsigned long chunks[2];
} uint128m_T;
```



# External Code Integration

---

- “External Code Integration for Code Generation” on page 25-2
- “Encapsulating the Interface to External Code” on page 25-3
- “Best Practices for Using `coder.ExternalDependency`” on page 25-4
- “Encapsulate Interface to an External C Library” on page 25-6
- “Update Build Information from MATLAB code” on page 25-9
- “Call External Functions Encapsulated by `coder.ExternalDependency`” on page 25-10

## External Code Integration for Code Generation

You can integrate external code with MATLAB code intended for code generation. The external code can be external libraries, object files, or C/C++ source code.

The basic workflow is:

- 1 Create the external code.
- 2 Call the external code from MATLAB code.
- 3 Specify the external file locations.
- 4 Generate code from the MATLAB code.

Call the external code and specify the file locations in one of the following ways:

- Use `coder.ExternalDependency` to encapsulate the interface to the external code. The `updateBuildInfo` method specifies file locations and other build information. Write methods that define the programming interface to the external functions. In your MATLAB code, use these methods to call the external functions.
- Use `coder.ceval` to call external functions from your MATLAB code. When you generate code, define the locations of external files.
- Use `coder.ceval` to call external functions from your MATLAB code. Use `coder.updateBuildInfo` to specify external file locations and update build information.

### See Also

`coder.ceval` | `coder.ExternalDependency` | `coder.updateBuildInfo`

### More About

- “Encapsulating the Interface to External Code”
- “Specify External File Locations”
- “External Function Calls from Generated Code”

## Encapsulating the Interface to External Code

Use the `coder.ExternalDependency` class to encapsulate the interface between external code and MATLAB code intended for code generation. With the encapsulation, you can separate the details of the interface from your MATLAB code. The methods of `coder.ExternalDependency`:

- specify the location of external files
- update build information
- define the programming interface for external functions

In your MATLAB code, you can call the external code without providing build information.

The workflow is:

- 1 Write a class definition file for a class that derives from `coder.ExternalDependency`.
- 2 Store the class definition file in a folder on the MATLAB path.
- 3 In your MATLAB code, use a method of the class to call an external function.
- 4 Generate code from your MATLAB code.

### See Also

`coder.ExternalDependency`

### Related Examples

- “Encapsulate Interface to an External C Library”

### More About

- “Best Practices for Using `coder.ExternalDependency`”

## Best Practices for Using `coder.ExternalDependency`

### In this section...

“Terminate Code Generation for Unsupported External Dependency” on page 25-4

“Parameterize Methods for MATLAB and Generated Code” on page 25-4

“Parameterize `updateBuildInfo` for Multiple Platforms” on page 25-5

### Terminate Code Generation for Unsupported External Dependency

The `isSupportedContext` method returns true if the external code interface is supported in the build context. If the external code interface is not supported, do not return false. Instead, use `error` to terminate code generation with an error message. For example:

```
function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('MyLibrary is not available for this target');
    end
end
```

### Parameterize Methods for MATLAB and Generated Code

Parameterize methods that call external functions so that the methods run in MATLAB. For example:

```
...
if coder.target('MATLAB')
    % running in MATLAB, use built-in addition
    c = a + b;
else
    % running in generated code, call library function
    coder.ceval('adder_initialize');
end
...
```

## Parameterize `updateBuildInfo` for Multiple Platforms

Parameterize the `updateBuildInfo` method to support multiple platforms. For example, use `coder.BuildConfig.getStdLibInfo` to get the platform-specific library file extensions.

```
...
    [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo()
% Link files
linkFiles = strcat('adder', linkLibExt);
buildInfo.addLinkObjects(linkFiles, linkPath, linkPriority, ...
    linkPrecompiled, linkLinkOnly, group);
...
```

### See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `error`

### Related Examples

- “Encapsulate Interface to an External C Library”

## Encapsulate Interface to an External C Library

This example shows how to encapsulate the interface to an external C dynamic linked library using `coder.ExternalDependency`.

Write a function `adder` that returns the sum of its inputs.

```
function c = adder(a,b)
    %#codegen
    c = a + b;
end
```

Generate a library that contains `adder`.

```
codegen('adder', '-args', {-2,5}, '-config:dll', '-report');
```

Write the class definition file `AdderAPI.m` to encapsulate the library interface.

```
%=====
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=====

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, ctx)
            [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

            % Header files
        end
    end
end
```

```
hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
buildInfo.addIncludePaths(hdrFilePath);

% Link files
linkFiles = strcat('adder', linkLibExt);
linkPath = hdrFilePath;
linkPriority = '';
linkPrecompiled = true;
linkLinkOnly = true;
group = '';
buildInfo.addLinkObjects(linkFiles, linkPath, ...
    linkPriority, linkPrecompiled, linkLinkOnly, group);

% Non-build files
nbFiles = 'adder';
nbFiles = strcat(nbFiles, execLibExt);
buildInfo.addNonBuildFiles(nbFiles, '', '');
end

%API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % running in generated code, call library function
        coder.cinclude('adder.h');

        % Because MATLAB Coder generated adder, use the
        % housekeeping functions before and after calling
        % adder with coder.ceval.
        % Call initialize function before calling adder for the
        % first time.

        coder.ceval('adder_initialize');
        c = 0;
        c = coder.ceval('adder', a, b);

        % Call the terminate function after
        % calling adder for the last time.

        coder.ceval('adder_terminate');
    end
end
```

```
        end
    end
end
```

Write a function `adder_main` that calls the external library function `adder`.

```
function y = adder_main(x1, x2)
%#codegen
    y = AdderAPI.adder(x1, x2);
end
```

Generate a MEX function for `adder_main`. The MEX Function exercises the `coder.ExternalDependency` methods.

```
codegen('adder_main', '-args', {7,9}, '-report')
```

Copy the library to the current folder using the file extension for your platform.

For Windows, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dll'));
```

For Linux, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.so'));
```

Run the MEX function and verify the result.

```
adder_main_mex(2,3)
```

## See Also

[coder.BuildConfig](#) | [coder.ExternalDependency](#) | [error](#)

## More About

- “Encapsulating the Interface to External Code”
- “Build Information Object”
- “Build Information Methods”



## Update Build Information from MATLAB code

You can choose to control aspects of the build process that occur after code generation but before compilation. For example, you can specify compiler or linker options.

To customize the build from your MATLAB code:

- 1 In your MATLAB code, call `coder.updateBuildInfo` to update the build information object. You specify a build information object method and the input arguments for the method.
- 2 Generate code from your MATLAB code.

### See Also

`coder.updateBuildInfo`

## Call External Functions Encapsulated by `coder.ExternalDependency`

When a method of a class derived from `coder.ExternalDependency` defines the interface to an external function, you call the external function by calling the method.

Suppose you define the following method for a class named `AdderAPI`:

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

This method defines the interface to a function `adder` which has two inputs, `a` and `b`. In your MATLAB code, call `adder` this way:

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

### See Also

`coder.ExternalDependency`

### Related Examples

- “Encapsulate Interface to an External C Library”

### More About

- “Encapsulating the Interface to External Code”

# Generate Efficient and Reusable Code

---

- “Optimization Strategies” on page 26-2
- “Modularize MATLAB Code” on page 26-5
- “Eliminate Redundant Copies of Function Inputs” on page 26-6
- “Inline Code” on page 26-8
- “Control Inlining Using Configuration Object” on page 26-10
- “Fold Function Calls into Constants” on page 26-13
- “Control Stack Space Usage” on page 26-15
- “Stack Allocation and Performance” on page 26-16
- “Rewrite Logical Array Indexing as a Loop” on page 26-17
- “Dynamic Memory Allocation and Performance” on page 26-18
- “Minimize Dynamic Memory Allocation” on page 26-19
- “Provide Maximum Size for Variable-Size Arrays” on page 26-20
- “Disable Dynamic Memory Allocation During Code Generation” on page 26-26
- “Set Dynamic Memory Allocation Threshold” on page 26-27
- “Excluding Unused Paths from Generated Code” on page 26-30
- “Prevent Code Generation for Unused Execution Paths” on page 26-31
- “Generate Code with Parallel for-Loops (parfor)” on page 26-33
- “Minimize Redundant Operations in Loops” on page 26-35
- “Unroll for-Loops” on page 26-37
- “Support for Integer Overflow and Non-Finites” on page 26-40
- “Integrate Custom Code” on page 26-42
- “MATLAB Coder Optimizations in Generated Code” on page 26-48
- “Generate Reusable Code” on page 26-51

## Optimization Strategies

MATLAB Coder introduces certain optimizations when generating C/C++ code or MEX functions from your MATLAB code. For more information, see “MATLAB Coder Optimizations in Generated Code”.

To optimize your generated code further, you can:

- Adapt your MATLAB code.
- Control code generation using the configuration object from the command-line or the Project Settings dialog box.

To optimize the execution speed of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have <code>for</code> -loops whose iterations are independent of each other.	“Generate Code with Parallel for-Loops ( <code>parfor</code> )”
You have variable-size arrays in your MATLAB code.	“Minimize Dynamic Memory Allocation”
You have multiple variable-size arrays in your MATLAB code. You want dynamical memory allocation for larger arrays and static allocation for smaller ones.	“Set Dynamic Memory Allocation Threshold”
You want your generated function to be called by reference.	“Eliminate Redundant Copies of Function Inputs”
You are calling small functions in your MATLAB code.	“Inline Code”
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	“Control Inlining Using Configuration Object”
You do not want to generate code for expressions that contain constants only.	“Fold Function Calls into Constants”
You have loop operations in your MATLAB code that do not depend on the loop index.	“Minimize Redundant Operations in Loops”

Condition	Action
You have integer operations in your MATLAB code. You know beforehand that integer overflow will not occur during execution of your generated code.	“Disable Support for Integer Overflow”
You know beforehand that Inf-s and NaN-s will not occur during execution of your generated code.	“Disable Support for Non-Finites”
You have for-loops with few iterations.	“Unroll for-Loops”
You already have legacy C/C++ code optimized for your target environment.	“Integrate Custom Code”

To optimize the memory usage of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have if/else/elseif statements or switch/case/otherwise statements in your MATLAB code. You do not require some branches of the statements in your generated code.	“Prevent Code Generation for Unused Execution Paths”
You have logical array indexing in your MATLAB code. For more information, see “Using Logicals in Array Indexing”.	“Rewrite Logical Array Indexing as a Loop”
You want your generated function to be called by reference.	“Eliminate Redundant Copies of Function Inputs”
You have limited stack space for your generated code.	“Control Stack Space Usage”
You are calling small functions in your MATLAB code.	“Inline Code”
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	“Control Inlining Using Configuration Object”
You do not want to generate code for expressions that contain constants only.	“Fold Function Calls into Constants”

<b>Condition</b>	<b>Action</b>
You have loop operations in your MATLAB code that do not depend on the loop index.	“Minimize Redundant Operations in Loops”
You have integer operations in your MATLAB code. You know beforehand that integer overflow will not occur during execution of your generated code.	“Disable Support for Integer Overflow”
You know beforehand that Inf-s and NaN-s will not occur during execution of your generated code.	“Disable Support for Non-Finites”

## Modularize MATLAB Code

For large MATLAB code, streamline code generation by modularizing the code:

- 1 Break up your MATLAB code into smaller, self-contained sections.
- 2 Save each section in a MATLAB function.
- 3 Generate C/C++ code for each function.
- 4 Call the generated C/C++ functions in sequence from a wrapper MATLAB function using `coder.ceval`.
- 5 Generate C/C++ code for the wrapper function.

Besides streamlining code generation for the original MATLAB code, this approach also supplies you with C/C++ codes for the individual sections. You can reuse these codes later by integrating them with other generated C/C++ code using `coder.ceval`.

## Eliminate Redundant Copies of Function Inputs

You can reduce the number of copies in your generated code by writing functions that use the same variable as both an input and an output. For example:

```
function A = foo( A, B ) %#codegen
A = A * B;
end
```

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, MATLAB passes the variable by reference in the generated code instead of redundantly copying the input to a temporary variable. In the preceding example, input `A` is passed by reference in the generated code because it also acts as an output for function `foo`:

```
...
/* Function Definitions */
void foo(double *A, double B)
{
    *A *= B;
}
...
```

The reference parameter optimization reduces memory usage and execution time, especially when the variable passed by reference is a large data structure. To achieve these benefits at the call site, call the function with the same variable as both input and output.

By contrast, suppose you rewrite function `foo` without the optimization:

```
function y = foo2( A, B ) %#codegen
y = A * B;
end
```

MATLAB generates code that passes the inputs by value and returns the value of the output:

```
...
/* Function Definitions */
double foo2(double A, double B)
{
    return A * B;
}
```



...

In some cases, the output of the function cannot be a modified version of its inputs. If you do not use the inputs later in the function, you can modify your code to operate on the inputs instead of on a copy of the inputs. One method is to create additional return values for the function. For example, consider the code:

```
function y1=foo(u1) %#codegen
    x1=u1+1;
    y1=bar(x1);
end

function y2=bar(u2)
    % Since foo does not use x1 later in the function,
    % it would be optimal to do this operation in place
    x2=u2.*2;
    % The change in dimensions in the following code
    % means that it cannot be done in place
    y2=[x2,x2];
end
```

You can modify this code to eliminate redundant copies.

```
function y1=foo(u1) %#codegen
    u1=u1+1;
    [y1, u1]=bar(u1);
end

function [y2, u2]=bar(u2)
    u2=u2.*2;
    % The change in dimensions in the following code
    % still means that it cannot be done in place
    y2=[u2,u2];
end
```

## Related Examples

- “Pass Structure Arguments by Reference or by Value”

## Inline Code

MATLAB uses internal heuristics to determine whether or not to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. For more information, see `coder.inline`.

In this section...
“Prevent Function Inlining” on page 26-8
“Use Inlining in Control Flow Statements” on page 26-8

### Prevent Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
    coder.inline('never');
    y = x;
end
```

### Use Inlining in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
    coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
    coder.inline('never');
```

```
end

if any(divisor == 0)
    error('Can not divide by 0');
end

y = dividend / divisor;
```

## Related Examples

- [“Control Inlining Using Configuration Object”](#)

## Control Inlining Using Configuration Object

This example shows how to control inlining behavior using the `codegen` configuration object. Restrict inlining when:

- The size of generated code exceeds desired limits due to excessive inlining of functions. Suppose you include the statement, `coder.inline('always')`, inside a certain function. You then call that function at a large number of different sites in your code. The generated code can be large due to the function being inlined every time it is called.

The call sites must be different. For instance, inlining does not lead to large code if the function to be inlined is called several times inside a loop.

- You have limited RAM or stack space.

### In this section...

“Control Size of Functions Inlined” on page 26-10

“Control Size of Functions After Inlining” on page 26-11

“Control Stack Size Limit on Inlined Functions” on page 26-11

## Control Size of Functions Inlined

You can control the maximum size of functions that can be inlined from the Project Settings dialog box or the command line. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThreshold`, to the maximum size that you want.

```
cfg = coder.config('lib');
cfg.InlineThreshold = 100;
```

Generate code using this configuration object.

## Control Size of Functions After Inlining

You can control the maximum size of functions after inlining from the Project Settings dialog box or the command line. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold max**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineThresholdMax = 100;
```

Generate code using this configuration object.

## Control Stack Size Limit on Inlined Functions

Specifying a limit on the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even after the function is executed. The value of the property, `InlineStackLimit`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that can be accommodated by a certain value of `InlineStackLimit`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

You can control the stack size limit on inlined functions from the Project Settings dialog box or the command line.

- In the Project Settings dialog box, on the **All Settings** tab, set the value of the field, **Inline stack limit**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

```
cfg = coder.config('lib');  
cfg.InlineStackLimit = 2000;
```

Generate code using this configuration object.

## **Related Examples**

- “Inline Code”

## Fold Function Calls into Constants

This example shows how to specify constants in generated code using `coder.const`. The code generation software folds an expression or a function call in a `coder.const` statement into a constant in generated code. Because the generated code does not have to evaluate the expression or call the function every time, this optimization reduces the execution time of the generated code.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generation software generates code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int k;
    for (k = 0; k < 10; k++) {
        y[k] = (double)((1 + k) * (1 + k)) + Shift;
    }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generation software creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
    int i0;
    static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                         49, 64, 81, 100 };

    for (i0 = 0; i0 < 10; i0++) {
        y[i0] = (double)iv0[i0] + Shift;
    }
}
```

### **See Also**

`coder.const`



## Control Stack Space Usage

This example shows how to set the maximum stack space used by the generated code. Set the maximum stack usage when:

- You have limited stack space, for instance, in case of embedded targets.
- Your C compiler reports a run-time stack overflow.

The value of the property, `InlineStackLimit`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that can be accommodated by a certain value of `InlineStackLimit`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

### Control Stack Space Usage Using Project Interface

- 1 On the **Build** tab **Settings** pane, set the **Output type** to `C/C++ Static Library`, `C/C++ Dynamic Library`, or `C/C++ Executable` (depending on your requirements).
- 2 Click the **More settings** link to open the **Project Settings** dialog box.
- 3 On the **Memory** tab, set the field, **Stack usage max**, to the value that you want.

### Control Stack Space Usage from Command Line

- 1 Create a configuration object for code generation.

Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

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

- 2 Set the property, `StackUsageMax`, to the value that you want.

```
cfg.StackUsageMax=400000;
```

### More About

- “Stack Allocation and Performance”

## Stack Allocation and Performance

By default, local variables are allocated on the stack. Large variables that do not fit on the stack are statically allocated in memory.

Stack allocation typically uses memory more efficiently than static allocation. However, stack space is sometimes limited, typically in embedded processors. MATLAB Coder allows you to manually set a limit on the stack space usage to make your generated code suitable for your target hardware. You can choose this limit based on the target hardware configurations. For more information, see “Control Stack Space Usage”.

## Rewrite Logical Array Indexing as a Loop

Rewriting logical array indexing as a loop can optimize the generated code for both speed and readability. For more information on logical array indexing, see “Using Logicals in Array Indexing”.

For example, the MATLAB function, `foo`, uses logical array indexing.

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

The generated C code for this function is not very efficient. Rewrite the MATLAB code to use a loop instead of logical indexing:

```
function x = foo_rewrite(x,N) %#codegen
assert(all(size(x) == [1 100]))
for ii=1:numel(x)
    if x(ii) > N
        x(ii) = N;
    end
end
```

## Dynamic Memory Allocation and Performance

To achieve faster execution of generated code, minimize dynamic (or run-time) memory allocation of arrays.

MATLAB Coder does not provide a size for unbounded arrays in generated code. Instead, such arrays are referenced indirectly through pointers. For such arrays, memory cannot be allocated during compilation of generated code. Based on storage requirements for the arrays, memory is allocated and freed at run time as required. This run-time allocation and freeing of memory leads to slower execution of the generated code. For more information on dynamic memory allocation, see “Bounded Versus Unbounded Variable-Size Data”.

### When Dynamic Memory Allocation Occurs

Dynamic memory allocation occurs when the code generation software cannot find upper bounds for variable-size arrays. The software cannot find upper bounds when you specify the size of an array using a variable that is not a compile-time constant. An example of such a variable is an input variable (or a variable computed from an input variable).

Instances in the MATLAB code that might lead to dynamic memory allocation are:

- Array initialization: You specify array size using a variable whose value is known only at run time.
- After initialization of an array:
  - You declare the array as variable-size using `coder.varsize` without explicit upper bounds. After this declaration, you expand the array by concatenation inside a loop. The number of loop runs is known only at run time.
  - You use a `reshape` function on the array. At least one of the size arguments to the `reshape` function is known only at run time.

If you know the maximum possible size of the array, you can avoid dynamic memory allocation. You can then provide an upper bound for the array and prevent dynamic memory allocation in generated code. For more information, see “Minimize Dynamic Memory Allocation” on page 26-19.

## Minimize Dynamic Memory Allocation

When possible, you should minimize dynamic memory allocation since it leads to slower execution of generated code. Dynamic memory allocation occurs when the code generation software cannot find upper bounds for variable-size arrays.

You can avoid dynamic memory allocation of a variable-size array if you know its maximum possible size. To do so, follow these steps:

- 1 “Provide Maximum Size for Variable-Size Arrays” on page 26-20.
- 2 Depending on your requirements, do one of the following:
  - “Disable Dynamic Memory Allocation During Code Generation” on page 26-26.
  - “Set Dynamic Memory Allocation Threshold”

---

**Caution** If a variable-size array in the MATLAB code does not have a maximum size, disabling dynamic memory allocation leads to a code generation error. Before disabling dynamic memory allocation, you must provide a maximum size for variable-size arrays in your MATLAB code.

---

### More About

- “Dynamic Memory Allocation and Performance”

## Provide Maximum Size for Variable-Size Arrays

To constrain array size for variable-size arrays, do one of the following:

- **Constrain Array Size Using `assert` Statements**

If the variable specifying array size is not a compile-time constant, use an `assert` statement with relational operators to constrain the variable. Doing so helps the code generation software to determine a maximum size for the array.

The following examples constrain array size using `assert` statements:

- **When Array Size Is Specified by Input Variables**

Define a function `array_init` which initializes an array `y` with input variable `N`:

```
function y = array_init (N)
    assert(N <= 25); % Generates exception if N > 25
    y = zeros(1,N);
```

The `assert` statement constrains input `N` to a maximum size of 25. In the absence of the `assert` statement, `y` is assigned a pointer to an array in the generated code, thus allowing dynamic memory allocation.

- **When Array Size Is Obtained from Computation Using Input Variables**

Define a function, `array_init_from_prod`, which takes two input variables, `M` and `N`, and uses their product to specify the maximum size of an array, `y`.

```
function y = array_init_from_prod (M,N)
    size=M*N;
    assert(size <= 25); % Generates exception if size > 25
    y=zeros(1,size);
```

The `assert` statement constrains the product of `M` and `N` to a maximum of 25.

Alternatively, if you restrict `M` and `N` individually, it leads to dynamic memory allocation:

```
function y = array_init_from_prod (M,N)
```

```
assert(M <= 5);  
assert(N <= 5);  
size=M*N;  
y=zeros(1,size);
```

This code causes dynamic memory allocation because `M` and `N` can both have unbounded negative values. Therefore, their product can be unbounded and positive even though, individually, their positive values are bounded.

---

**Tip** Place the `assert` statement on a variable immediately before it is used to specify array size.

---

---

**Tip** You can use `assert` statements to restrict array sizes in most cases. When expanding an array inside a loop, this strategy does not work if the number of loop runs is known only at run time.

---

## • Restrict Concatenations in a Loop Using `coder.varsize` with Upper Bounds

You can expand arrays beyond their initial size by concatenation. When you concatenate additional elements inside a loop, there are two syntax rules for expanding arrays.

### 1 **Array size during initialization is not a compile-time constant**

If the size of an array during initialization is not a compile-time constant, you can expand it by concatenating additional elements:

```
function out=ExpandArray(in) % Expand an array by five elements  
    out = zeros(1,in);  
    for i=1:5  
        out = [out 0];  
    end
```

## 2 Array size during initialization is a compile-time constant

Before concatenating elements, you have to declare the array as variable-size using `coder.versize`:

```
function out=ExpandArray() % Expand an array by five elements
    out = zeros(1,5);
    coder.versize('out');
    for i=1:5
        out = [out 0];
    end
```

Either case leads to dynamic memory allocation. To prevent dynamic memory allocation in such cases, use `coder.versize` with explicit upper bounds. This example shows how to use `coder.versize` with explicit upper bounds:

### Restrict Concatenations Using `coder.versize` with Upper Bounds

- 1 Define a function, `RunningAverage`, that calculates the running average of an `N`-element subset of an array:

```
function avg=RunningAverage(N)

% Array whose elements are to be averaged
NumArray=[1 6 8 2 5 3];

% Initialize average:
% These will also be the first two elements of the function output
avg=[0 0];

% Place a bound on the argument
coder.versize('avg',[1 8]);

% Loop to calculate running average
for i=1:N
    s=0;
    s=s+sum(NumArray(1:i));
    avg=[avg s/i];
% Increase the size of avg as required by concatenation
end
```



The output, `avg`, is an array that you can expand as required to accommodate the running averages. As a new running average is calculated, it is added to the array `avg` through concatenation, thereby expanding the array.

Because the maximum possible number of running averages is equal to the number of elements in `NumArray`, you can supply an explicit upper bound for `avg` in the `coder.varsizes` statement. In this example, the upper bound is 8 (the two initial elements plus the six elements of `NumArray`).

- 2 Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, `avg` is assigned an array of size 8 (static memory allocation). The function definition for `RunningAverage` appears as follows (using built-in C types):

```
void RunningAverage (double N, double avg_data[8], int avg_size[2])
```

- 3 By contrast, if you remove the explicit upper bound, the generated code dynamically allocates `avg`.

Replace the statement

```
coder.varsizes('avg',[1 8]);
```

with:

```
coder.varsizes('avg');
```

- 4 Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, `avg` is assigned a pointer to an array, thereby allowing dynamic memory allocation. The function definition for `RunningAverage` appears as follows (using built-in C types):

```
void Test(double N, mxArray_real_T *avg)
```

---

**Note:** Dynamic memory allocation also occurs if you precede `coder.varsizes('avg')` with the following assert statement:

```
assert(N < 6);
```

The `assert` statement does not restrict the number of concatenations within the loop.

---

## • Constrain Array Size When Rearranging a Matrix

The statement `out = reshape(in,m,n,...)` takes an array, `in`, as an argument and returns array, `out`, having the same elements as `in`, but reshaped as an `m`-by-`n`-by-... matrix. If one of the size variables `m,n,...` is not a compile-time constant, then dynamic memory allocation of `out` takes place.

To avoid dynamic memory allocation, use an `assert` statement before the `reshape` statement to restrict the size variables `m,n,...` to `numel(in)`. This example shows how to use an `assert` statement before a `reshape` statement:

### Rearrange a Matrix into Given Number of Rows

- 1 Define a function, `ReshapeMatrix`, which takes an input variable, `N`, and reshapes a matrix, `mat`, to have `N` rows:

```
function [out1,out2] = ReshapeMatrix(N)

    mat = [1 2 3 4 5; 4 5 6 7 8]
    % Since mat has 10 elements, N must be a factor of 10
    % to pass as argument to reshape

    out1 = reshape(mat,N,[]);
    % N is not restricted

    assert(N < numel(mat));
    % N is restricted to number of elements in mat
    out2 = reshape(mat,N,[]);
```

- 2 Generate code for `ReshapeArray` using the `codegen` command (the input argument does not have to be a factor of 10):

```
codegen -config:lib -report ReshapeArray -args 3
```

While `out1` is dynamically allocated, `out2` is assigned an array with size 100 (=10 X 10) in the generated code.

---

**Tip** If your system has limited memory, do not use the `assert` statement in this way. For an  $n$ -element matrix, the `assert` statement creates an  $n$ -by- $n$  matrix, which might be large.

---

### Related Examples

- “Minimize Dynamic Memory Allocation”
- “Disable Dynamic Memory Allocation During Code Generation”
- “Set Dynamic Memory Allocation Threshold”

### More About

- “Dynamic Memory Allocation and Performance”

## Disable Dynamic Memory Allocation During Code Generation

Disabling dynamic memory allocation during code generation leads to faster execution of generated code. You can disable dynamic memory allocation explicitly from the project settings dialog box or the command line.

To disable dynamic memory allocation in the Project Settings box :

- 1 On the MATLAB Coder project **Build** tab, click **More settings**.
- 2 In the **Project Settings** dialog box **Memory** tab, under **Enable variable-sizing**, set **Dynamic memory allocation** to **Never**.

To disable dynamic memory allocation from the command line:

- 1 In the MATLAB workspace, define the configuration object:

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

- 2 Set the `DynamicMemoryAllocation` property of the configuration object to `Off`:

```
cfg.DynamicMemoryAllocation = 'Off';
```

Disabling dynamic memory allocation leads to a code generation error if a variable-size array in the MATLAB code does not have a maximum upper bound. Therefore, you can also use this feature to identify variable-size arrays in your MATLAB code that do not have a maximum upper bound. These arrays are the ones that are dynamically allocated in the generated code.

### Related Examples

- “Minimize Dynamic Memory Allocation”
- “Provide Maximum Size for Variable-Size Arrays”
- “Set Dynamic Memory Allocation Threshold”

### More About

- “Dynamic Memory Allocation and Performance”

## Set Dynamic Memory Allocation Threshold

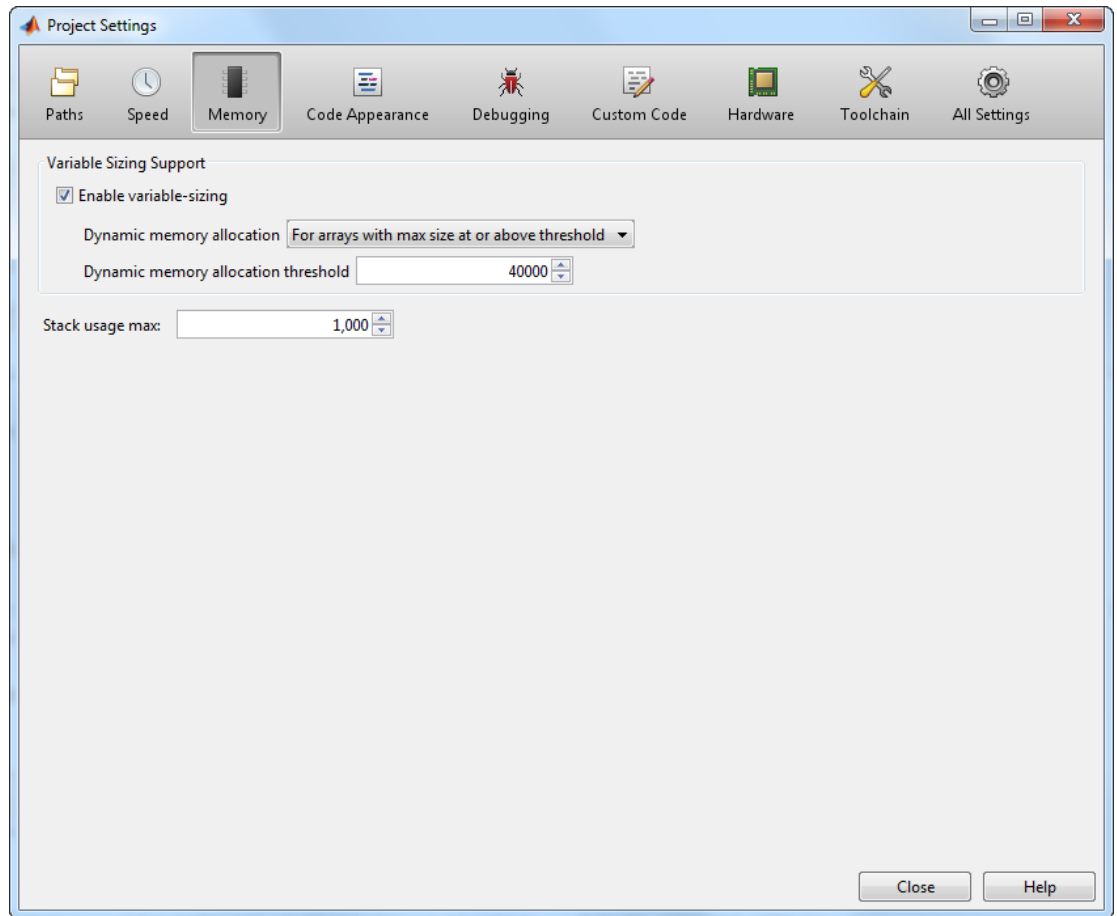
This example shows how to specify a dynamic memory allocation threshold for variable-size arrays. Dynamic memory allocation optimizes storage requirements for variable-size arrays but causes slower execution of generated code. Instead of disabling dynamic memory allocation for all variable-size arrays, you can disable it only for arrays below a certain size. Set a dynamic memory allocation threshold to disable dynamic memory allocation for array size below the threshold and enable it for array size at or above the threshold.

Use this strategy when you want to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, it can be more efficient to speed up generated code by allocating memory statically. Though static memory allocation can lead to unused storage space, it might not be a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, you can reduce storage requirements significantly using dynamic memory allocation.

### Set Dynamic Memory Allocation Threshold Using Project Interface

- 1 On the **Build** tab **Settings** pane, click the **More settings** link to open the **Project Settings** dialog box.
- 2 On the **Memory** tab, select **Enable variable-sizing**.
- 3 On the same tab, select the **For arrays with max size at or above threshold** option in the **Dynamic memory allocation** list.
- 4 Set the **Dynamic memory allocation threshold** to the value that you want.



The **Dynamic memory allocation threshold** value is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that can be accommodated by a certain value of `DynamicMemoryAllocationThreshold`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

## Set Dynamic Memory Allocation Threshold from Command Line

- 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

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

- 2 Set the property, `DynamicMemoryAllocation`, to `'Threshold'`.

```
cfg.DynamicMemoryAllocation='Threshold';
```

- 3 Set the property, `DynamicMemoryAllocationThreshold`, to the value that you want.

```
cfg.DynamicMemoryAllocationThreshold = 40000;
```

The value stored in `DynamicMemoryAllocationThreshold` is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that can be accommodated by a certain value of `DynamicMemoryAllocationThreshold`. This estimate excludes possible C compiler optimizations such as putting variables in registers.

### Related Examples

- “Minimize Dynamic Memory Allocation”
- “Provide Maximum Size for Variable-Size Arrays”
- “Disable Dynamic Memory Allocation During Code Generation”

### More About

- “Dynamic Memory Allocation and Performance”

## Excluding Unused Paths from Generated Code

In certain situations, you do not need some branches of an `if`, `elseif`, `else` statement or a `switch`, `case`, `otherwise` statement in your generated code. For instance :

- You have a MATLAB function that performs multiple tasks determined by a control-flow variable. You might not need some of the tasks in the code generated from this function.
- You have an `if/elseif/else` statement in a MATLAB function performing different tasks based on the nature (type/value) of the input. In some cases, you know the nature of the input beforehand. If so, you do not need some branches of the `if` statement.

You can prevent code generation for the unused branches of an `if/elseif/else` statement or a `switch/case/otherwise` statement. Declare the control-flow variable as a constant. The code generation software generates code only for the branch that is chosen by the control-flow variable.

### Related Examples

- “Prevent Code Generation for Unused Execution Paths”



## Prevent Code Generation for Unused Execution Paths

### In this section...

“Prevent Code Generation When Local Variable Controls Flow” on page 26-31

“Prevent Code Generation When Input Variable Controls Flow” on page 26-32

If a variable controls the flow of an `if`, `elseif`, `else` statement or a `switch`, `case`, `otherwise` statement, declare it as constant so that code generation takes place for one branch of the statement only.

Depending on the nature of the control-flow variable, you can declare it as constant in two ways:

- If the variable is local to the MATLAB function, assign it to a constant value in the MATLAB code. For an example, see “Prevent Code Generation When Local Variable Controls Flow” on page 26-31.
- If the variable is an input to the MATLAB function, you can declare it as constant using `coder.Constant`. For an example, see “Prevent Code Generation When Input Variable Controls Flow” on page 26-32.

### Prevent Code Generation When Local Variable Controls Flow

- 1 Define a function `SquareOrCube` which takes an input variable, `in`, and squares or cubes its elements based on whether the choice variable, `ch`, is set to `s` or `c`:

```
function out = SquareOrCube(ch,in) %#codegen
    if ch=='s'
        out = in.^2;
    elseif ch=='c'
        out = in.^3;
    else
        out = 0;
    end
```

- 2 Generate code for `SquareOrCube` using the `codegen` command:

```
codegen -config:lib SquareOrCube -args {'s',zeros(2,2)}
```

The generated C code squares or cubes the elements of a 2-by-2 matrix based on the input for `ch`.

- 3 Add the following line to the definition of `SquareOrCube`:

```
ch = 's';
```

The generated C code squares the elements of a 2-by-2 matrix. The choice variable, `ch`, and the other branches of the `if/elseif/if` statement do not appear in the generated code.

## Prevent Code Generation When Input Variable Controls Flow

- 1 Define a function `MathFunc`, which performs different mathematical operations on an input, `in`, depending on the value of the input, `flag`:

```
function out = MathFunc(flag,in) %#codegen
    %# codegen
    switch flag
        case 1
            out=sin(in);
        case 2
            out=cos(in);
        otherwise
            out=sqrt(in);
    end
```

- 2 Generate code for `MathFunc` using the `codegen` command:

```
codegen -config:lib MathFunc -args {1,zeros(2,2)}
```

The generated C code performs different math operations on the elements of a 2-by-2 matrix based on the input for `flag`.

- 3 Generate code for `MathFunc`, declaring the argument, `flag`, as a constant using `coder.Constant`:

```
codegen -config:lib MathFunc -args {coder.Constant(1),zeros(2,2)}
```

The generated C code finds the sine of the elements of a 2-by-2 matrix. The variable, `flag`, and the `switch/case/otherwise` statement do not appear in the generated code.

## More About

- “Excluding Unused Paths from Generated Code”

## Generate Code with Parallel for-Loops (parfor)

This example shows how to generate C code for a MATLAB algorithm that contains a `parfor`-loop.

- 1 Write a MATLAB function that contains a `parfor`-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
    a(i,:)=real(fft(r(i,:)));
end
```

- 2 Generate C code for `test_parfor`. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor
```

Because you did not specify the maximum number of threads to use, the generated C code executes the loop iterations in parallel on the available number of cores.

- 3 To specify a maximum number of threads, rewrite the function `test_parfor` as follows:

```
function a = test_parfor(u) %#codegen
a=ones(10,256);
r=rand(10,256);
parfor (i=1:10,u)
    a(i,:)=real(fft(r(i,:)));
end
```

- 4 Generate C code for `test_parfor`. Use `-args 0` to specify that the input, `u`, is a scalar double. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor -args 0
```

In the generated code, the iterations of the `parfor`-loop run on at most the number of cores specified by the input, `u`. If less than `u` cores are available, the iterations run on the cores available at the time of the call.

### More About

- “Algorithm Acceleration Using Parallel for-Loops (`parfor`)”
- “Classification of Variables in `parfor`-Loops”

- “Reduction Assignments in parfor-Loops”

## Minimize Redundant Operations in Loops

This example shows how to minimize redundant operations in loops. When a loop operation does not depend on the loop index, performing it inside a loop is redundant. This redundancy often goes unnoticed when you are performing multiple operations in a single MATLAB statement inside a loop. For example, in the following code, the inverse of the matrix **B** is being calculated 100 times inside the loop although it does not depend on the loop index:

```
for i=1:100
    C=C + inv(B)*A^i*B;
end
```

Performing such redundant loop operations can lead to unnecessary processing. To avoid unnecessary processing, move operations outside loops as long as they do not depend on the loop index.

- 1 Define a function, `SeriesFunc(A,B,n)`, that calculates the sum of  $n$  terms in the following power series expansion:

$$C = 1 + B^{-1}AB + B^{-1}A^2B + \dots$$

```
function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
C=zeros(size(A));

% Perform the series sum
for i=1:n
    C=C+inv(B)*A^i*B;
end
```

- 2 Generate code for `SeriesFunc` with 4-by-4 matrices passed as input arguments for **A** and **B**:

```
X = coder.typeof(zeros(4));
codegen -config:lib -launchreport SeriesFunc -args {X,X,10}
```

In the generated code, the inversion of **B** is performed  $n$  times inside the loop. It is more economical to perform the inversion operation once outside the loop because it does not depend on the loop index.

- 3 Modify `SeriesFunc` as follows:

```
function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
C=zeros(size(A));

% Perform the inversion outside the loop
inv_B=inv(B);

% Perform the series sum
for i=1:n
    C=C+inv_B*A^i*B;
end
```

This procedure performs the inversion of **B** only once, leading to faster execution of the generated code.

## Unroll for-Loops

Unrolling `for`-loops eliminates the loop logic by creating a separate copy of the loop body in the generated code for each iteration. Within each iteration, the loop index variable becomes a constant.

You can also force loop unrolling for individual functions by wrapping the loop header in a `coder.unroll` function. For more information, see `coder.unroll`.

### Limit Copying the for-loop Body in Generated Code

To limit the number of times that you copy the body of a `for`-loop in generated code:

- 1 Write a MATLAB function `getrand(n)` that uses a `for`-loop to generate a vector of length `n` and assign random numbers to specific elements. Add a test function `test_unroll`. This function calls `getrand(n)` with `n` equal to values both less than and greater than the threshold for copying the `for`-loop in generated code.

```
function [y1, y2] = test_unroll() %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
% Calling getrand 8 times triggers unroll
y1 = getrand(8);
% Calling getrand 50 times does not trigger unroll
y2 = getrand(50);

function y = getrand(n)
% Turn off inlining to make
% generated code easier to read
coder.inline('never');

% Set flag variable downroll to repeat loop body
% only for fewer than 10 iterations
downroll = n < 10;
% Declare size, class, and complexity
% of variable y by assignment
y = zeros(n, 1);
% Loop body begins
for i = coder.unroll(1:2:n, downroll)
    if (i > 2) && (i < n-2)
        y(i) = rand();
    end;
end;
```

```
end;  
% Loop body ends
```

- 2 In the default output folder, `codegen/lib/test_unroll`, generate C static library code for `test_unroll`:

```
codegen -config:lib test_unroll
```

In `test_unroll.c`, the generated C code for `getrand(8)` repeats the body of the `for`-loop (unrolls the loop) because the number of iterations is less than 10:

```
static void getrand(double y[8])  
{  
    /* Turn off inlining to make */  
    /* generated code easier to read */  
    /* Set flag variable downroll to repeat loop body */  
    /* only for fewer than 10 iterations */  
    /* Declare size, class, and complexity */  
    /* of variable y by assignment */  
    memset(&y[0], 0, sizeof(double) << 3);  
  
    /* Loop body begins */  
    y[2] = b_rand();  
    y[4] = b_rand();  
  
    /* Loop body ends */  
}
```

The generated C code for `getrand(50)` does not unroll the `for`-loop because the number of iterations is greater than 10:

```
static void b_getrand(double y[50])  
{  
    int i;  
    int b_i;  
  
    /* Turn off inlining to make */  
    /* generated code easier to read */  
    /* Set flag variable downroll to repeat loop body */  
    /* only for fewer than 10 iterations */  
    /* Declare size, class, and complexity */  
    /* of variable y by assignment */  
    memset(&y[0], 0, 50U * sizeof(double));  
  
    /* Loop body begins */
```



```
for (i = 0; i < 25; i++) {  
    b_i = (i << 1) + 1;  
    if ((b_i > 2) && (b_i < 48)) {  
        y[b_i - 1] = b_rand();  
    }  
}
```

## Support for Integer Overflow and Non-Finites

In addition to code generated for your MATLAB function, the code-generation software generates supporting code for the following situations:

- The result of an integer operation falls outside the range that a data type can represent. This situation is known as integer overflow.
- Non-finite values (`inf` and `NaN`) are generated from an operation. The supporting code is contained in the files `rt_nonfinite.c`, `rtGetInf.c` and `rtGetNaN.c` (with corresponding header files).

You can suppress generation of the supporting code if you know beforehand that such situations will not arise. This action reduces the size and increases the speed of generated code at the cost of potentially producing results that do not match simulation in case the situations arise.

### Disable Support for Integer Overflow

You can disable support for integer overflow in the project settings dialog box or at the command line. On disabling this support, the overflow behavior of your generated code depends on your target C compiler. Most C compilers wrap on overflow.

- In the project settings dialog box:
  - 1 On the **Build** tab **Settings** pane, click the **More settings** link to open the **Project Settings** dialog box.
  - 2 To disable support for integer overflow, on the **Speed** tab, clear **Saturate on integer overflow**.
- At the command line:
  - 1 Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```
  - 2 To disable support for integer overflow, set the `SaturateOnIntegerOverflow` property to `false`.

```
cfg.SaturateOnIntegerOverflow = false;
```

## Disable Support for Non-Finites

You can disable support for non-finites (`inf` and `NaN`) in the project settings dialog box or at the command line.

- In the project settings dialog box:
  - 1** On the **Build** tab **Settings** pane, set the **Output type** to **C/C++ Static Library**, **C/C++ Dynamic Library**, or **C/C++ Executable** (depending on your requirements).
  - 2** Click the **More settings** link to open the **Project Settings** dialog box.
  - 3** To disable support for integer overflow, on the **Speed** tab, clear **Support non-finite numbers**.
- At the command line:
  - 1** Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'` or `'exe'` (depending on your requirements). For example:

```
cfg = coder.config('lib');
```
  - 2** To disable support for integer overflow, set the `SupportNonFinite` property to `false`.

```
cfg.SupportNonFinite = false;
```

## Integrate Custom Code

This example shows how to integrate custom code to enhance performance of generated code. Although MATLAB Coder generates optimized code for most applications, you might have legacy code optimized for your specific requirements. For example:

- You have custom libraries optimized for your target environment.
- You have custom libraries for functions not supported by MATLAB Coder.
- You have custom libraries that meet standards set by your company.

In such cases, you can integrate your custom code with the code generated by MATLAB Coder.

This example illustrates how to integrate the function `cublasSgemm` from the NVIDIA<sup>®</sup> CUDA<sup>®</sup> Basic Linear Algebra Subroutines (CUBLAS) library in generated code. This function performs matrix multiplication on a Graphics Processing Unit (GPU).

- 1 Define a class `ExternalLib_API` that derives from the class `coder.ExternalDependency`. `ExternalLib_API` defines an interface to the CUBLAS library through the following methods:
  - `getDescriptiveName`: Returns a descriptive name for `ExternalLib_API` to be used for error messages.
  - `isSupportedContext`: Determines if the build context supports the CUBLAS library.
  - `updateBuildInfo`: Adds header file paths and link files to the build information.
  - `GPU_MatrixMultiply`: Defines the interface to the CUBLAS library function `cublasSgemm`.

### ExternalLib\_API.m

```
classdef ExternalLib_API < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(-)
            bName = 'ExternalLib_API';
```

```
end

function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('CUBLAS library not available for this target');
    end
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, ~, ~] = ctx.getStdLibInfo();

    % Include header file path
    % Include header files later using coder.cinclude
    hdrFilePath = 'C:\My_Includes';
    buildInfo.addIncludePaths(hdrFilePath);

    % Include link files
    linkFiles = strcat('libcublas', linkLibExt);
    linkPath = 'C:\My_Libs';
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    linkFiles = strcat('libcudart', linkLibExt);
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

end

%API for library function 'cuda_MatrixMultiply'
function C = GPU_MatrixMultiply(A, B)
    assert(isa(A, 'single'), 'A must be single. ');
    assert(isa(B, 'single'), 'B must be single. ');

    if(coder.target('MATLAB'))
        C=A*B;
    else

        % Include header files
```

```
% for external functions and typedefs
% Header path included earlier using updateBuildInfo
coder.cinclude('"cuda_runtime.h"');
coder.cinclude('"cublas_v2.h"');

% Compute dimensions of input matrices
m = int32(size(A, 1));
k = int32(size(A, 2));
n = int32(size(B, 2));

% Declare pointers to matrices on destination GPU
d_A = coder.opaque('float*');
d_B = coder.opaque('float*');
d_C = coder.opaque('float*');

% Compute memory to be allocated for matrices
% Single = 4 bytes
size_A = m*k*4;
size_B = k*n*4;
size_C = m*n*4;

% Define error variables
error = coder.opaque('cudaError_t');
cudaSuccessV = coder.opaque('cudaError_t', ...
    'cudaSuccess');

% Assign memory on destination GPU
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_A), size_A);
assert(error == cudaSuccessV, ...
    'cudaMalloc(A) failed');
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_B), size_B);
assert(error == cudaSuccessV, ...
    'cudaMalloc(B) failed');
error = coder.ceval('cudaMalloc', ...
    coder.wref(d_C), size_C);
assert(error == cudaSuccessV, ...
    'cudaMalloc(C) failed');

% Define direction of copying
hostToDevice = coder.opaque('cudaMemcpyKind', ...
    'cudaMemcpyHostToDevice');
```

```

% Copy matrices to destination GPU
error = coder.ceval('cudaMemcpy', ...
    d_A, coder.rref(A), size_A, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(A) failed');

error = coder.ceval('cudaMemcpy', ...
    d_B, coder.rref(B), size_B, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(B) failed');

% Define type and size for result
C = zeros(m, n, 'single');

error = coder.ceval('cudaMemcpy', ...
    d_C, coder.rref(C), size_C, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

% Define handle variables for external library
handle = coder.opaque('cublasHandle_t');
blasSuccess = coder.opaque('cublasStatus_t', ...
    'CUBLAS_STATUS_SUCCESS');

% Initialize external library
ret = coder.opaque('cublasStatus_t');
ret = coder.ceval('cublasCreate', coder.wref(handle));
assert(ret == blasSuccess, 'cublasCreate failed');

TRANSA = coder.opaque('cublasOperation_t', ...
    'CUBLAS_OP_N');
alpha = single(1);
beta = single(0);

% Multiply matrices on GPU
ret = coder.ceval('cublasSgemm', handle, ...
    TRANSA, TRANSA, m, n, k, ...
    coder.rref(alpha), d_A, m, ...
    d_B, k, ...
    coder.rref(beta), d_C, k);

assert(ret == blasSuccess, 'cublasSgemm failed');

% Copy result back to local host
deviceToHost = coder.opaque('cudaMemcpyKind', ...
    'cudaMemcpyDeviceToHost');

```

```

        error = coder.ceval('cudaMemcpy', coder.wref(C), ...
            d_C, size_C, deviceToHost);
        assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');
    end
end
end
end
end

```

- 2 To perform the matrix multiplication using the interface defined in method `GPU_MatrixMultiply` and the build information in `ExternalLib_API`, include the following line in your MATLAB code:

```
C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

For instance, you can define a MATLAB function `Matrix_Multiply` that solely performs this matrix multiplication.

```
function C = Matrix_Multiply(A, B) %#codegen
    C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

- 3 Define a MEX configuration object using `coder.config`. For using the CUBLAS libraries, set the target language for code generation to C++.

```
cfg=coder.config('mex');
cfg.TargetLang='C++';
```

- 4 Generate code for `Matrix_Multiply` using `cfg` as the configuration object and two 2 X 2 matrices of type `single` as arguments. Since `cublasSgemv` supports matrix multiplication for data type `float`, the corresponding MATLAB matrices must have type `single`.

```
codegen -config cfg Matrix_Multiply ...
        -args {ones(2,'single'),ones(2,'single')}
```

- 5 Test the generated MEX function `Matrix_Multiply_mex` using two 2 X 2 identity matrices of type `single`.

```
Matrix_Multiply_mex(eye(2,'single'),eye(2,'single'))
```

The output is also a 2 X 2 identity matrix.

## See Also

`coder.BuildConfig` | `assert` | `coder.ceval` | `coder.ExternalDependency` | `coder.opaque` | `coder.rref` | `coder.wref`



## **Related Examples**

- “Encapsulate Interface to an External C Library”

## **More About**

- “Encapsulating the Interface to External Code”

## MATLAB Coder Optimizations in Generated Code

### In this section...

“Constant Folding” on page 26-48

“Loop Fusion” on page 26-49

“Successive Matrix Operations Combined” on page 26-49

“Unreachable Code Elimination” on page 26-50

In order to improve the execution speed and memory usage of generated code, MATLAB Coder introduces the following optimizations:

### Constant Folding

When possible, the code generation software evaluates expressions in your MATLAB code that involve compile-time constants only. In the generated code, it replaces these expressions with the result of the evaluations. This behavior is known as constant folding. Because of constant folding, the generated code does not have to evaluate the constants during execution.

The following example shows MATLAB code that is constant-folded during code generation. The function `MultiplyConstant` multiplies every element in a matrix by a scalar constant. The function evaluates this constant using the product of three compile-time constants, `a`, `b` and `c`.

```
function out=MultiplyConstant(in) %#codegen
    a=pi^4;
    b=1/factorial(4);
    c=exp(-1);
    out=in.*(a*b*c);
end
```

The code generation software evaluates the expressions involving compile-time constants, `a`, `b`, and `c`. It replaces these expressions with the result of the evaluation in generated code.

Constant folding can occur when the expressions involve scalars only. To explicitly enforce constant folding of expressions in other cases, use the `coder.const` function. For more information, see “Fold Function Calls into Constants”.

## Control Constant Folding

You can control the maximum number of instructions that can be constant-folded from the command line or the Project Settings dialog box.

- At the command line, create a configuration object for code generation. Set the property `ConstantFoldingTimeout` to the value that you want.

```
cfg=coder.config('lib');  
cfg.ConstantFoldingTimeout = 200;
```

- In the Project Settings dialog box, on the **All Settings** tab, set the field **Constant folding timeout** to the value that you want.

## Loop Fusion

When possible, the code generation software fuses successive loops with the same number of runs into a single loop in the generated code. This optimization reduces loop overhead.

The following code contains successive loops, which are fused during code generation. The function `SumAndProduct` evaluates the sum and product of the elements in an array `Arr`. The function uses two separate loops to evaluate the sum `y_f_sum` and product `y_f_prod`.

```
function [y_f_sum,y_f_prod] = SumAndProduct(Arr) %#codegen  
    y_f_sum = 0;  
    y_f_prod = 1;  
    for i = 1:length(Arr)  
        y_f_sum = y_f_sum+Arr(i);  
    end  
    for i = 1:length(Arr)  
        y_f_prod = y_f_prod*Arr(i);  
    end
```

The code generated from this MATLAB code evaluates the sum and product in a single loop.

## Successive Matrix Operations Combined

When possible, the code generation software converts successive matrix operations in your MATLAB code into a single loop operation in generated code. This optimization

reduces excess loop overhead involved in performing the matrix operations in separate loops.

The following example contains code where successive matrix operations take place. The function `ManipulateMatrix` multiplies every element of a matrix `Mat` with a `factor`. To every element in the result, the function then adds a `shift` :

```
function Res=ManipulateMatrix(Mat,factor,shift)
    Res=Mat*factor;
    Res=Res+shift;
end
```

The generated code combines the multiplication and addition into a single loop operation.

## Unreachable Code Elimination

When possible, the code generation software suppresses code generation from unreachable procedures in your MATLAB code. For instance, if a branch of an `if`, `elseif`, `else` statement is unreachable, then code is not generated for that branch.

The following example contains unreachable code, which is eliminated during code generation. The function `SaturateValue` returns a value based on the range of its input `x`.

```
function y_b = SaturateValue(x) %#codegen
    if x>0
        y_b = x;
    elseif x>10 %This is redundant
        y_b = 10;
    else
        y_b = -x;
    end
```

The second branch of the `if`, `elseif`, `else` statement is unreachable. If the variable `x` is greater than 10, it is also greater than 0. Therefore, the first branch is executed in preference to the second branch.

MATLAB Coder does not generate code for the unreachable second branch.

## Generate Reusable Code

With MATLAB, you can generate reusable code in the following ways:

- Write reusable functions using standard MATLAB function file names which you can call from different locations, for example, in a Simulink model or MATLAB function library.
- Compile external functions on the MATLAB path and integrate them into generated C code for embedded targets.

See “Resolution of Function Calls for Code Generation”.

Common applications include:

- Overriding generated library function with a custom implementation.
- Implementing a reusable library on top of standard library functions that can be used with Simulink.
- Swapping between different implementations of the same function.

