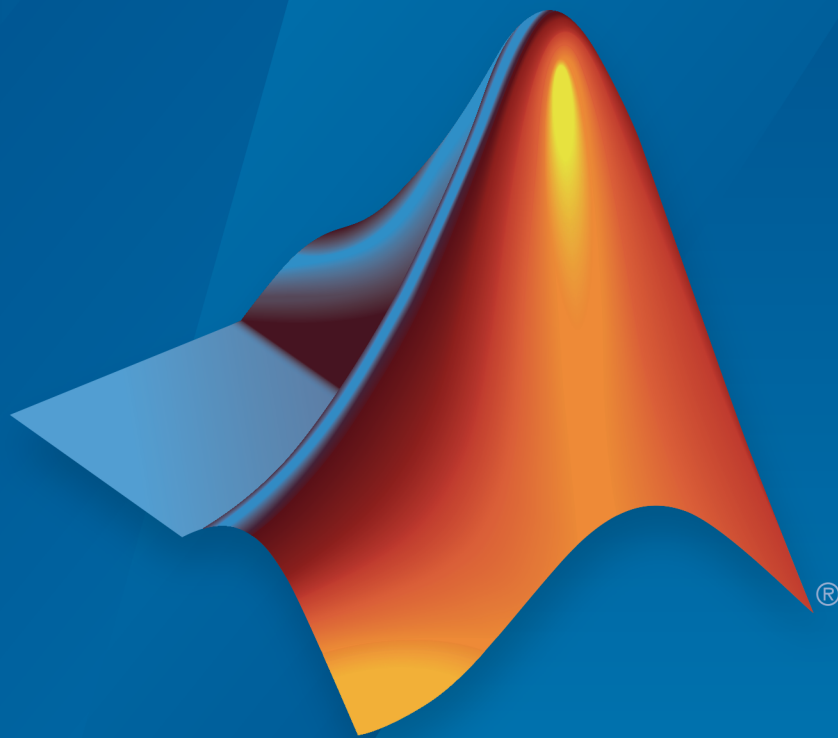


MATLAB[®] Coder[™]

User's Guide



MATLAB[®]

R2015b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

MATLAB[®] Coder[™] User's Guide

© COPYRIGHT 2011–2015 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

April 2011	Online only
September 2011	Online only
March 2012	Online only
September 2012	Online only
March 2013	Online only
September 2013	Online only
March 2014	Online only
October 2014	Online only
March 2015	Online only
September 2015	Online only

New for Version 2 (R2011a)
Revised for Version 2.1 (Release 2011b)
Revised for Version 2.2 (Release 2012a)
Revised for Version 2.3 (Release 2012b)
Revised for Version 2.4 (Release 2013a)
Revised for Version 2.5 (Release 2013b)
Revised for Version 2.6 (Release 2014a)
Revised for Version 2.7 (Release 2014b)
Revised for Version 2.8 (Release 2015a)
Revised for Version 3.0 (Release 2015b)

Check Bug Reports for Issues and Fixes

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

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

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

About MATLAB Coder

1

MATLAB Coder Product Description	1-2
Key Features	1-2
Product Overview	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
Code Generation Workflow	1-5
See Also	1-5

Design Considerations for C/C++ Code Generation

2

When to Generate Code from MATLAB Algorithms	2-2
When Not to Generate Code from MATLAB Algorithms	2-2
Which Code Generation Feature to Use	2-4
Prerequisites for C/C++ Code Generation from MATLAB ...	2-5
MATLAB Code Design Considerations for Code Generation	2-6
See Also	2-7
Differences in Behavior After Compiling MATLAB Code ...	2-8
Why Are There Differences?	2-8
Character Size	2-8
Order of Evaluation in Expressions	2-8
Termination Behavior	2-10

Size of Variable-Size N-D Arrays	2-10
Size of Empty Arrays	2-10
Shape of Array When Index and Assigned Value Are Empty	2-10
Floating-Point Numerical Results	2-11
NaN and Infinity Patterns	2-12
Code Generation Target	2-12
MATLAB Class Initial Values	2-12
Variable-Size Support for Code Generation	2-12
Complex Numbers	2-12

MATLAB Language Features Supported for C/C++ Code

Generation	2-13
MATLAB Features That Code Generation Supports	2-13
MATLAB Language Features That Code Generation Does Not Support	2-14

3 System Objects Supported for Code Generation

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

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

Functions and Objects Supported for C and C++ Code Generation — Alphabetical List	4-2
--	-----

Functions and Objects Supported for C and C++ Code Generation — Category List	4-156
Aerospace Toolbox	4-158
Arithmetic Operations in MATLAB	4-158
Bit-Wise Operations MATLAB	4-159
Casting in MATLAB	4-160
Communications System Toolbox	4-160
Complex Numbers in MATLAB	4-166
Computer Vision System Toolbox	4-167

Control Flow in MATLAB	4-176
Data and File Management in MATLAB	4-177
Data Types in MATLAB	4-181
Desktop Environment in MATLAB	4-182
Discrete Math in MATLAB	4-183
DSP System Toolbox	4-183
Error Handling in MATLAB	4-191
Exponents in MATLAB	4-191
Filtering and Convolution in MATLAB	4-192
Fixed-Point Designer	4-193
HDL Coder	4-203
Histograms in MATLAB	4-203
Image Acquisition Toolbox	4-204
Image Processing in MATLAB	4-204
Image Processing Toolbox	4-204
Input and Output Arguments in MATLAB	4-214
Interpolation and Computational Geometry in MATLAB ..	4-214
Linear Algebra in MATLAB	4-218
Logical and Bit-Wise Operations in MATLAB	4-219
MATLAB Compiler	4-220
Matrices and Arrays in MATLAB	4-220
Neural Network Toolbox	4-229
Numerical Integration and Differentiation in MATLAB ..	4-229
Optimization Functions in MATLAB	4-230
Phased Array System Toolbox	4-231
Polynomials in MATLAB	4-241
Programming Utilities in MATLAB	4-242
Relational Operators in MATLAB	4-242
Robotics System Toolbox	4-242
Rounding and Remainder Functions in MATLAB	4-243
Set Operations in MATLAB	4-244
Signal Processing in MATLAB	4-249
Signal Processing Toolbox	4-250
Special Values in MATLAB	4-255
Specialized Math in MATLAB	4-256
Statistics in MATLAB	4-256
Statistics and Machine Learning Toolbox	4-257
String Functions in MATLAB	4-267
System Identification Toolbox	4-269
Trigonometry in MATLAB	4-270

Defining MATLAB Variables for C/C++ Code Generation

5

Variables Definition for Code Generation	5-2
Best Practices for Defining Variables for C/C++ Code Generation	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
Eliminate Redundant Copies of Variables in Generated Code	5-7
When Redundant Copies Occur	5-7
How to Eliminate Redundant Copies by Defining Uninitialized Variables	5-7
Defining Uninitialized Variables	5-8
Reassignment of Variable Properties	5-9
Define and Initialize Persistent Variables	5-10
Reuse the Same Variable with Different Properties	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
Avoid Overflows in for-Loops	5-15
Supported Variable Types	5-17

Defining Data for Code Generation

6

Data Definition for Code Generation	6-2
--	-----

Code Generation for Complex Data	6-4
Restrictions When Defining Complex Variables	6-4
Code Generation for Complex Data with Zero-Valued Imaginary Parts	6-4
Results of Expressions That Have Complex Operands	6-8
Code Generation for Characters and Strings	6-9
Array Size Restrictions for Code Generation	6-10
See Also	6-10
Code Generation for Constants in Structures and Arrays .	6-11

Code Generation for Variable-Size Data

7

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

C Code Interface for Arrays	7-17
C Code Interface for Statically Allocated Arrays	7-17
C Code Interface for Dynamically Allocated Arrays	7-18
Utility Functions for Creating emxArray Data Structures ..	7-20
Diagnose and Fix Variable-Size Data Errors	7-22
Diagnosing and Fixing Size Mismatch Errors	7-22
Diagnosing and Fixing Errors in Detecting Upper Bounds ..	7-24
Incompatibilities with MATLAB in Variable-Size Support for Code Generation	7-26
Incompatibility with MATLAB for Scalar Expansion	7-26
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays	7-28
Incompatibility with MATLAB in Determining Size of Empty Arrays	7-29
Incompatibility with MATLAB in Determining Class of Empty Arrays	7-30
Incompatibility with MATLAB in Vector-Vector Indexing ..	7-31
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation	7-31
Incompatibility with MATLAB in Concatenating Variable-Size Matrices	7-32
Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements	7-33
Dynamic Memory Allocation Not Supported for MATLAB Function Blocks	7-34
Variable-Sizing Restrictions for Code Generation of Toolbox Functions	7-35
Common Restrictions	7-35
Toolbox Functions with Variable Sizing Restrictions	7-36

Code Generation for MATLAB Structures

8

Structure Definition for Code Generation	8-2
Structure Operations Allowed for Code Generation	8-3

Define Scalar Structures for Code Generation	8-4
Restrictions When Defining Scalar Structures by Assignment	8-4
Adding Fields in Consistent Order on Each Control Flow Path	8-4
Restriction on Adding New Fields After First Use	8-5
Define Arrays of Structures for Code Generation	8-6
Ensuring Consistency of Fields	8-6
Using repmat to Define an Array of Structures with Consistent Field Properties	8-6
Defining an Array of Structures by Using struct	8-7
Defining an Array of Structures Using Concatenation	8-7
Make Structures Persistent	8-8
Index Substructures and Fields	8-9
Assign Values to Structures and Fields	8-11
Pass Structure Arguments by Reference or by Value	8-13
Specify Pass by Reference or by Value Using the MATLAB Coder App	8-13
Specify Pass by Reference or by Value Using the Command- Line Interface	8-14
Pass Input Structure Argument by Reference	8-14
Pass Input Structure Argument by Value	8-15
Pass Output Structure Argument by Reference	8-16
Pass Output Structure Argument by Value	8-16
Pass Input and Output Structure Argument by Reference ..	8-17

Code Generation for Cell Arrays

9

Homogeneous vs. Heterogeneous Cell Arrays	9-2
Control Whether a Cell Array is Homogeneous or Heterogeneous	9-4
Define Cell Array Inputs	9-5

Make a Cell Array Variable Size	9-6
Name the Structure Type That Represents a Cell Array ...	9-8
Cell Array Requirements and Limitations for Code Generation	9-10
Cell Array Definition	9-10
Cell Array Indexing	9-11
Variable-Size Cell Arrays	9-11
Cell Array Contents	9-12
Cell Arrays in Structures	9-12
Passing to External C/C++ Functions	9-12
Cell Arrays in Code Generation Reports	9-13
Cell Array Variable in the MATLAB Code Pane	9-13
Cell Array Variable on the Variables Tab	9-13

Code Generation for Enumerated Data

10

Enumerated Data Definition for Code Generation	10-2
Enumerated Types Supported for Code Generation	10-3
Enumeration Class Base Types for Code Generation	10-3
C Code Representation for Base Type <code>int32</code>	10-4
C Code Representation for Base Type Other Than <code>int32</code> ..	10-4
When to Use Enumerated Data for Code Generation	10-6
Generate Code for Enumerated Data from MATLAB Algorithms	10-7
See Also	10-7
Define Enumerated Data for Code Generation	10-8
Naming Enumerated Types for Code Generation	10-9
Operations on Enumerated Data for Code Generation ...	10-10
Assignment Operator, <code>=</code>	10-10
Relational Operators, <code><</code> <code>></code> <code><=</code> <code>>=</code> <code>==</code> <code>~=</code>	10-10
Cast Operation	10-10

Indexing Operation	10-11
Control Flow Statements: if, switch, while	10-11
Include Enumerated Data in Control Flow Statements ..	10-13
if Statement with Enumerated Data Types	10-13
switch Statement with Enumerated Data Types	10-14
while Statement with Enumerated Data Types	10-16
Customize Enumerated Types for Code Generation	10-19
Customizing Enumerated Types	10-19
Specify a Default Enumerated Value	10-21
Specify a Header File	10-22
Use Enumerated Types in LED Control Function	10-24
Control Names of Enumerated Type Values in Generated Code	10-27
Change and Reload Enumerated Data Types	10-30
Restrictions on Use of Enumerated Data in for-Loops ...	10-31
Toolbox Functions That Support Enumerated Types for Code Generation	10-32

Code Generation for MATLAB Classes

11

MATLAB Classes Definition for Code Generation	11-2
Language Limitations	11-2
Code Generation Features Not Compatible with Classes ...	11-3
Defining Class Properties for Code Generation	11-4
Calls to Base Class Constructor	11-5
Inheritance from Built-In MATLAB Classes Not Supported .	11-6
Classes That Support Code Generation	11-8
Generate Code for MATLAB Value Classes	11-9

Generate Code for MATLAB Handle Classes and System Objects	11-14
MATLAB Classes in Code Generation Reports	11-16
What Reports Tell You About Classes	11-16
How Classes Appear in Code Generation Reports	11-16
How to Generate a Code Generation Report	11-18
Troubleshooting Issues with MATLAB Classes	11-19
Class class does not have a property with name name	11-19
Handle Object Limitations for Code Generation	11-21
A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop	11-21
A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object	11-21
System Objects Requirements and Limitations for Code Generation	11-24

Code Generation for Function Handles

12

Function Handle Definition for Code Generation	12-2
Define and Pass Function Handles for Code Generation ..	12-3
Function Handle Limitations for Code Generation	12-5

Defining Functions for Code Generation

13

Specify Variable Numbers of Arguments	13-2
Supported Index Expressions	13-3

Apply Operations to a Variable Number of Arguments	13-4
When to Force Loop Unrolling	13-4
Using Variable Numbers of Arguments in a for-Loop	13-5
Implement Wrapper Functions	13-6
Passing Variable Numbers of Arguments from One Function to Another	13-6
Pass Property/Value Pairs	13-7
Variable Length Argument Lists for Code Generation	13-9

Calling Functions for Code Generation

14

Resolution of Function Calls for Code Generation	14-2
Key Points About Resolving Function Calls	14-4
Compile Path Search Order	14-4
When to Use the Code Generation Path	14-5
Resolution of File Types on Code Generation Path	14-6
Compilation Directive %#codegen	14-8
Call Local Functions	14-9
Call Supported Toolbox Functions	14-10
Call MATLAB Functions	14-11
Declaring MATLAB Functions as Extrinsic Functions	14-12
Calling MATLAB Functions Using feval	14-16
How MATLAB Resolves Extrinsic Functions During Simulation	14-16
Working with mxArray	14-17
Restrictions on Extrinsic Functions for Code Generation . .	14-19
Limit on Function Arguments	14-19

Detect Dead and Constant-Folded Code	15-2
What Is Dead Code?	15-2
Detect Dead Code	15-2
Fix Dead Code	15-5
Convert MATLAB Code to Fixed-Point C Code	15-7
Propose Fixed-Point Data Types Based on Simulation Ranges	15-9
Propose Fixed-Point Data Types Based on Derived Ranges	15-23
Specify Type Proposal Options	15-38
Detect Overflows	15-42
Replace the exp Function with a Lookup Table	15-53
Replace a Custom Function with a Lookup Table	15-62
Enable Plotting Using the Simulation Data Inspector ...	15-71
Visualize Differences Between Floating-Point and Fixed-Point Results	15-72
Log Data for Histogram	15-83
View and Modify Variable Information	15-85
View Variable Information	15-85
Modify Variable Information	15-85
Revert Changes	15-87
Promote Sim Min and Sim Max Values	15-88
Automated Fixed-Point Conversion	15-89
Automated Fixed-Point Conversion Capabilities	15-89
Code Coverage	15-90
Proposing Data Types	15-93
Locking Proposed Data Types	15-95

Viewing Functions	15-96
Viewing Variables	15-103
Log Data for Histogram	15-105
Function Replacements	15-107
Validating Types	15-108
Testing Numerics	15-108
Detecting Overflows	15-108
Convert Fixed-Point Conversion Project to MATLAB Scripts	15-110
Generated Fixed-Point Code	15-113
Location of Generated Fixed-Point Files	15-113
Minimizing fi-casts to Improve Code Readability	15-114
Avoiding Overflows in the Generated Fixed-Point Code ..	15-114
Controlling Bit Growth	15-115
Avoiding Loss of Range or Precision	15-115
Handling Non-Constant mpower Exponents	15-117
Fixed-Point Code for MATLAB Classes	15-119
Automated Conversion Support for MATLAB Classes ...	15-119
Unsupported Constructs	15-119
Coding Style Best Practices	15-120
Automated Fixed-Point Conversion Best Practices	15-122
Create a Test File	15-122
Prepare Your Algorithm for Code Acceleration or Code Generation	15-123
Check for Fixed-Point Support for Functions Used in Your Algorithm	15-124
Manage Data Types and Control Bit Growth	15-124
Convert to Fixed Point	15-125
Use the Histogram to Fine-Tune Data Type Settings	15-126
Optimize Your Algorithm	15-127
Avoid Explicit Double and Single Casts	15-129
Replacing Functions Using Lookup Table Approximations	15-130
MATLAB Language Features Supported for Automated Fixed-Point Conversion	15-131

Inspecting Data Using the Simulation Data Inspector . . .	15-133
What Is the Simulation Data Inspector?	15-133
Import Logged Data	15-133
Export Logged Data	15-133
Group Signals	15-134
Run Options	15-134
Create Report	15-134
Comparison Options	15-134
Enabling Plotting Using the Simulation Data Inspector . .	15-134
Save and Load Simulation Data Inspector Sessions	15-135
Custom Plot Functions	15-136
Data Type Issues in Generated Code	15-138
Enable the Highlight Option in the MATLAB Coder App . .	15-138
Enable the Highlight Option at the Command Line	15-138
Stowaway Doubles	15-138
Stowaway Singles	15-139
Expensive Fixed-Point Operations	15-139

Automated Fixed-Point Conversion Using Programmatic Workflow

16

Convert MATLAB Code to Fixed-Point C Code	16-2
Propose Fixed-Point Data Types Based on Simulation Ranges	16-5
Propose Fixed-Point Data Types Based on Derived Ranges	16-10
Detect Overflows	16-18
Replace the exp Function with a Lookup Table	16-22
Replace a Custom Function with a Lookup Table	16-24
Enable Plotting Using the Simulation Data Inspector . . .	16-27

Visualize Differences Between Floating-Point and Fixed-Point Results	16-28
---	--------------

Single-Precision Conversion

17

Generate Single-Precision C Code at the Command Line ..	17-2
Prerequisites	17-2
Create a Folder and Copy Relevant Files	17-2
Determine the Type of the Input Argument	17-4
Generate and Run Single-Precision MEX to Verify Numerical Behavior	17-5
Generate Single-Precision C Code	17-5
View the Generated Single-Precision C Code	17-6
View Potential Data Type Issues	17-6
 Generate Single-Precision C Code Using the MATLAB Coder App	17-8
Prerequisites	17-8
Create a Folder and Copy Relevant Files	17-8
Open the MATLAB Coder App	17-10
Enable Single-Precision Conversion	17-11
Select the Source Files	17-11
Define Input Types	17-12
Check for Run-Time Issues	17-12
Generate Single-Precision C Code	17-12
View the Generated C Code	17-13
View Potential Data Type Issues	17-13
 Generate Single-Precision MATLAB Code	17-14
Prerequisites	17-14
Create a Folder and Copy Relevant Files	17-14
Set Up the Single-Precision Configuration Object	17-16
Generate Single-Precision MATLAB Code	17-16
View the Type Proposal Report	17-17
View Generated Single-Precision MATLAB Code	17-18
View Potential Data Type Issues	17-19
Compare the Double-Precision and Single-Precision Variables	17-19
Optionally Generate Single-Precision C Code	17-21

Choose a Single-Precision Conversion Workflow	17-23
Single-Precision Conversion Best Practices	17-24
Use Integers for Index Variables	17-24
Limit Use of assert Statements	17-24
Initialize MATLAB Class Properties in Constructor	17-24
Provide a Test File That Calls Your MATLAB Function	17-24
Prepare Your Code for Code Generation	17-25
Verify Double-Precision Code Before Single-Precision Conversion	17-25
Best Practices for Generation of Single-Precision C/C++ Code	17-25
Best Practices for Generation of Single-Precision MATLAB Code	17-26
Warnings from Conversion to Single-Precision C/C++	
Code	17-28
Function Uses Double-Precision in the C89/C90 Standard	17-28
Built-In Function Is Implemented in Double-Precision	17-29
Built-In Function Returns Double-Precision	17-30
Combining Integers and Double-Precision Numbers	17-32
MATLAB Language Features Supported for Single-Precision Conversion	17-33
MATLAB Language Features Supported for Single-Precision Conversion	17-33
MATLAB Language Features Not Supported for Single- Precision Conversion	17-34

Setting Up a MATLAB Coder Project

18

Set Up a MATLAB Coder Project	18-2
Create a Project	18-2
Open an Existing Project	18-2
Specify Properties of Entry-Point Function Inputs Using the App	18-3
Why Specify Input Properties?	18-3

Specify an Input Definition Using the App	18-3
Automatically Define Input Types Using the App	18-4
Define Input Parameters by Example Using the App	18-5
Define an Input Parameter by Example	18-5
Specify Input Parameters by Example	18-6
Specify a Structure Type Input Parameter by Example	18-7
Specify a Cell Array Type Input Parameter by Example	18-7
Specify an Enumerated Type Input Parameter by Example	18-9
Specify a Fixed-Point Input Parameter by Example	18-10
Define or Edit Input Parameter Type Using the App	18-12
Define or Edit an Input Parameter Type	18-12
Specify an Enumerated Type Input Parameter by Type	18-13
Specify a Fixed-Point Input Parameter by Type	18-14
Specify a Structure Input Parameter	18-14
Specify a Cell Array Input Parameter	18-18
Define Constant Input Parameters Using the App	18-23
Define Inputs Programmatically in the MATLAB File	18-24
Add Global Variables Using the App	18-25
Specify Global Variable Type and Initial Value Using the App	18-26
Why Specify a Type Definition for Global Variables?	18-26
Specify a Global Variable Type	18-26
Define a Global Variable by Example	18-26
Define or Edit Global Variable Type	18-27
Define Global Variable Initial Value	18-28
Define Global Variable Constant Value	18-29
Remove Global Variables	18-29
Changing Output Type	18-30
Project Settings	18-30
Configuration Object Parameters	18-31
Slow Operations in MATLAB Coder App	18-33
Unable to Open a MATLAB Coder Project	18-34

19 Preparing MATLAB Code for C/C++ Code Generation

Workflow for Preparing MATLAB Code for Code Generation	
Generation	19-2
See Also	19-3
Fixing Errors Detected at Design Time	19-4
See Also	19-4
Using the Code Analyzer	19-5
Check Code With the Code Analyzer	19-6
Check Code Using the Code Generation Readiness Tool ..	19-8
Run Code Generation Readiness Tool at the Command Line	19-8
Run Code Generation Readiness Tool from the Current Folder	
Browser	19-8
Run the Code Generation Readiness Tool Using the MATLAB	
Coder App	19-8
Code Generation Readiness Tool	19-9
Information That the Code Generation Readiness Tool	
Provides	19-9
Summary Tab	19-10
Code Structure Tab	19-11
See Also	19-14
Unable to Determine Code Generation Readiness	19-15
Generate MEX Functions Using the MATLAB Coder App ..	19-16
Workflow for Generating MEX Functions Using the MATLAB	
Coder App	19-16
Generate a MEX Function Using the MATLAB Coder App	19-16
Configure Project Settings	19-19
Build a MATLAB Coder Project	19-19
See Also	19-20
Generate MEX Functions at the Command Line	19-21
Command-line Workflow for Generating MEX Functions ..	19-21
Generate MEX Functions at the Command Line	19-21

Generating MEX Functions at the Command Line Using codegen	19-22
See Also	19-22
Fix Errors Detected at Code Generation Time	19-23
See Also	19-23
Design Considerations When Writing MATLAB Code for Code Generation	19-24
See Also	19-25
Running MEX Functions	19-26
Debugging MEX Functions	19-26
Debugging Strategies	19-27

Testing MEX Functions in MATLAB

20

Workflow for Testing MEX Functions in MATLAB	20-2
See Also	20-2
Why Test MEX Functions in MATLAB?	20-4
Running MEX Functions	20-5
Debugging MEX Functions	20-5
Verify MEX Functions Using the MATLAB Coder App	20-6
Verify MEX Functions at the Command Line	20-7
Debug Run-Time Errors	20-8
Viewing Errors in the Run-Time Stack	20-8
Handling Run-Time Errors	20-10
Using MEX Functions That MATLAB Coder Generates ..	20-11
Using Generated C/C++ Code	20-12

Code Generation Workflow	21-3
See Also	21-3
C/C++ Code Generation	21-5
Specify Custom Files to Build	21-5
Generating C/C++ Static Libraries from MATLAB Code ...	21-6
Generate a C Static Library Using the MATLAB Coder App	21-6
Generate a C Static Library at the Command Line	21-8
Generating C/C++ Dynamically Linked Libraries from MATLAB Code	21-10
Dynamic Libraries Generated by MATLAB Coder	21-10
Generate a C Dynamically Linked Library Using the MATLAB Coder App	21-10
Generate a C Dynamic Library at the Command Line ...	21-13
Generating Standalone C/C++ Executables from MATLAB Code	21-15
Generate a C Executable Using the MATLAB Coder App .	21-15
Generate a C Executable at the Command Line	21-24
Specifying main Functions for C/C++ Executables	21-25
Specify main Functions	21-26
Configure Build Settings	21-27
Specify Build Type	21-27
Specify a Language for Code Generation	21-29
Specify Output File Name	21-31
Specify Output File Locations	21-31
Parameter Specification Methods	21-33
Specify Build Configuration Parameters	21-33
Specify Data Types Used in Generated Code	21-39
Specify Data Type Using the MATLAB Coder App	21-39
Specify Data Type at the Command Line	21-39
Standard Math Libraries	21-40

Change the Standard Math Library	21-41
See Also	21-41
Share Build Configuration Settings	21-42
Export Settings	21-42
Import Settings	21-43
See Also	21-43
Convert MATLAB Coder Project to MATLAB Script	21-44
Convert a Project Using the MATLAB Coder App	21-44
Convert a Project Using the Command-Line Interface	21-44
Run the Script	21-44
Primary Function Input Specification	21-46
Why You Must Specify Input Properties	21-46
Properties to Specify	21-46
Rules for Specifying Properties of Primary Inputs	21-49
Methods for Defining Properties of Primary Inputs	21-50
Define Input Properties by Example at the Command Line	21-51
Specify Constant Inputs at the Command Line	21-53
Specify Variable-Size Inputs at the Command Line	21-54
Specify Cell Array Inputs at the Command Line	21-56
Specify Cell Array Inputs by Example	21-56
Specify the Type of the Cell Array Input	21-57
Make a Homogeneous Copy of a Type	21-58
Make a Heterogeneous Copy of a Type	21-59
Specify Variable-Size Cell Array Inputs	21-60
Specify Type Name for Heterogeneous Cell Array Inputs ..	21-61
Specify Constant Cell Array Inputs	21-61
Control Constant Inputs in MEX Function Signatures ...	21-63
Control MEX Function Signature Using the MATLAB Coder App	21-63
Control MEX Function Signature at the Command-Line Interface	21-63
Options for Controlling Constant Inputs in MEX Function Signatures	21-64
Call MEX Function with a Constant Input	21-65
See Also	21-66

Define Input Properties Programmatically in the MATLAB	
File	21-67
How to Use assert with MATLAB Coder	21-67
Rules for Using assert Function	21-73
Specifying General Properties of Primary Inputs	21-73
Specifying Properties of Primary Fixed-Point Inputs	21-74
Specifying Properties of Cell Arrays	21-75
Specifying Class and Size of Scalar Structure	21-77
Specifying Class and Size of Structure Array	21-77
Speed Up Compilation	21-79
Generate Code Only	21-79
Disable Creation of the Code Generation Report	21-80
Paths and File Infrastructure Setup	21-81
Compile Path Search Order	21-81
Specify Folders to Search for Custom Code	21-81
Naming Conventions	21-82
Generate Code for Multiple Entry-Point Functions	21-87
Advantages of Generating Code for Multiple Entry-Point Functions	21-87
Generate Code for More Than One Entry-Point Function Using the MATLAB Coder App	21-87
Generating Code for More Than One Entry-Point Function at the Command Line	21-90
How to Call an Entry-Point Function in a MEX Function .	21-91
How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code	21-92
Generate Code for Global Data	21-93
Workflow	21-93
Declare Global Variables	21-93
Define Global Data	21-94
Synchronizing Global Data with MATLAB	21-95
Define Constant Global Data	21-99
Limitations of Using Global Data	21-102
Specify Global Cell Arrays at the Command Line	21-103
Generate Code for Enumerated Types	21-105
See Also	21-105

Generate Code for Variable-Size Data	21-106
Disable Support for Variable-Size Data	21-106
Control Dynamic Memory Allocation	21-107
Generating Code for MATLAB Functions with Variable-Size Data	21-109
Generate Code for a MATLAB Function That Expands a Vector in a Loop	21-110
Using Dynamic Memory Allocation for an "Atoms" Simulation	21-116
 Code Generation for MATLAB Classes	 21-124
 How MATLAB Coder Partitions Generated Code	 21-125
Partitioning Generated Files	21-125
How to Select the File Partitioning Method	21-125
Partitioning Generated Files with One C/C++ File Per MATLAB File	21-126
Generated Files and Locations	21-131
File Partitioning and Inlining	21-133
 Requirements for Signed Integer Representation	 21-138
 Customize the Post-Code-Generation Build Process	 21-139
Customize Build Using coder.updateBuildInfo	21-139
Customize Build Using Post-Code-Generation Command .	21-139
Build Information Object	21-140
Build Information Methods	21-140
Write Post-Code-Generation Command	21-174
Use Post-Code-Generation Command to Customize Build	21-175
Write and Use Post-Code-Generation Command at the Command Line	21-176
 Troubleshooting	 21-177
Run-time Stack Overflow	21-177

Verify Generated C/C++ Code

22

Generation of Traceable Code	22-2
Code Traceability	22-2

Generate Traceable Code	22-2
Format of Traceability Tags	22-4
Location of Comments in Generated Code	22-5
Traceability Limitations	22-9
Code Generation Reports	22-10
About Code Generation Reports	22-10
View Your MATLAB Code in a Report	22-13
Viewing Call Stack Information	22-14
View Generated C and C++ Code in a Report	22-16
View the Build Summary Information	22-17
View Errors and Warnings in a Report	22-17
Viewing Variables in Your MATLAB Code	22-18
View Target Build Information	22-24
Keyboard Shortcuts for the Code Generation Report	22-25
Report Limitations	22-26
Enable Code Generation Reports	22-27
Enable Code Generation Reports with the MATLAB Coder App	22-27
Enable Code Generation Reports at the Command Line ..	22-27
Run-Time Error Detection and Reporting in Standalone C/C+ + Code	22-28
Generate Standalone Code That Detects and Reports Run- Time Errors	22-30

Code Replacement for MATLAB Code

23

What Is Code Replacement?	23-2
Code Replacement Libraries	23-4
Code Replacement Terminology	23-6
Code Replacement Limitations	23-9
Replace Code Generated from MATLAB Code	23-10

Choose a Code Replacement Library	23-12
About Choosing a Code Replacement Library	23-12
Explore Available Code Replacement Libraries	23-12
Explore Code Replacement Library Contents	23-12

24

Custom Toolchain Registration

Custom Toolchain Registration	24-2
What Is a Custom Toolchain?	24-2
What Is a Factory Toolchain?	24-2
What is a Toolchain Definition?	24-3
Key Terms	24-4
Typical Workflow	24-4
About <code>coder.make.ToolchainInfo</code>	24-6
Create and Edit Toolchain Definition File	24-8
Toolchain Definition File with Commentary	24-10
Steps Involved in Writing a Toolchain Definition File	24-10
Write a Function That Creates a <code>ToolchainInfo</code> Object ...	24-10
Setup	24-11
Macros	24-11
C Compiler	24-12
C++ Compiler	24-12
Linker	24-13
Archiver	24-13
Builder	24-14
Build Configurations	24-14
Create and Validate <code>ToolchainInfo</code> Object	24-16
Register the Custom Toolchain	24-17
Use the Custom Toolchain	24-19
Troubleshooting Custom Toolchain Validation	24-20
Build Tool Command Path Incorrect	24-20
Build Tool Not in System Path	24-20

Tool Path Does Not Exist	24-21
Unsupported Platform	24-21
Toolchain is Not installed	24-22
Project or Configuration is Using the Template Makefile . .	24-22
Skipped Validation of Build Tool “Download” or “Execute” .	24-23
Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers	24-24

Deploying Generated Code

25

Call a C Static Library Function from C Code	25-2
Call a C/C++ Static Library Function from MATLAB Code .	25-4
Call Generated C/C++ Functions	25-6
Conventions for Calling Functions in Generated Code	25-6
How to Call C/C++ Functions from MATLAB Code	25-6
Calling Initialize and Terminate Functions	25-7
Calling C/C++ Functions with Multiple Outputs	25-8
Calling C/C++ Functions that Return Arrays	25-8
Use a C Dynamic Library in a Microsoft Visual Studio Project	25-9
Specify External File Locations	25-12
External File Locations for External Code Integration	25-12
Specify External Files in a Class Derived from coder.ExternalDependency	25-12
Specify External Files in MATLAB Code Using coder.updateBuildInfo	25-12
Specify External Files Using the MATLAB Coder App	25-13
Specify External Files at the Command Line	25-13
Specify External Files with Configuration Objects	25-14
Code Generation of Matrices and Arrays	25-16

Incorporate Generated Code Using an Example Main Function	25-18
Workflow for Using an Example Main Function	25-18
Control Example Main Generation Using the MATLAB Coder App	25-19
Control Example Main Generation Using the Command-Line Interface	25-19
Use an Example C Main in an Application	25-21
Prerequisites	25-21
Create a Folder and Copy Relevant Files	25-22
Run the Sobel Filter on the Image	25-24
Generate and Test a MEX Function	25-26
Generate an Example Main Function for sobel.m	25-26
Copy the Example Main Files	25-29
Modify the Generated Example Main Function	25-29
Generate the Sobel Filter Application	25-42
Run the Sobel Filter Application	25-42
Display the Resulting Image	25-42
Package Code for Other Development Environments	25-44
When to Package Code	25-44
Package Generated Code Using the MATLAB Coder App ..	25-44
Package Generated Code at the Command Line	25-45
Specify packNGo Options	25-47
Structure of Generated Example C/C++ Main Function ..	25-49
Contents of the File main.c or main.cpp	25-49
Contents of the File main.h	25-52
Troubleshoot Failures in Deployed Code	25-53

Accelerating MATLAB Algorithms

26

Workflow for Accelerating MATLAB Algorithms	26-2
See Also	26-3

Best Practices for Using MEX Functions to Accelerate	
MATLAB Algorithms	26-4
Accelerate Code That Dominates Execution Time	26-4
Include Loops Inside MEX Function	26-4
Avoid Generating MEX Functions from Unsupported	
Functions	26-5
Avoid Generating MEX Functions if Built-In MATLAB	
Functions Dominate Run Time	26-6
Minimize MEX Function Calls	26-6
Edge Detection on Images	26-7
Accelerate MATLAB Algorithms	26-14
Modifying MATLAB Code for Acceleration	26-15
How to Modify Your MATLAB Code for Acceleration	26-15
Control Run-Time Checks	26-16
Types of Run-Time Checks	26-16
When to Disable Run-Time Checks	26-16
How to Disable Run-Time Checks	26-17
Algorithm Acceleration Using Parallel for-Loops (parfor)	26-19
Parallel for-Loops (parfor) in Generated Code	26-19
How parfor-Loops Improve Execution Speed	26-20
When to Use parfor-Loops	26-20
When Not to Use parfor-Loops	26-20
parfor-Loop Syntax	26-21
parfor Restrictions	26-21
Control Compilation of parfor-Loops	26-25
When to Disable parfor	26-25
Reduction Assignments in parfor-Loops	26-26
What are Reduction Assignments?	26-26
Multiple Reductions in a parfor-Loop	26-26
Classification of Variables in parfor-Loops	26-27
Overview	26-27
Sliced Variables	26-28
Broadcast Variables	26-29
Reduction Variables	26-29
Temporary Variables	26-34

Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)	26-36
Specify Maximum Number of Threads in parfor-Loops ..	26-37
Troubleshooting parfor-Loops	26-38
Global or Persistent Declarations in parfor-Loop	26-38
Compiler Does Not Support OpenMP	26-38
Accelerating Simulation of Bouncing Balls	26-39

27

Calling C/C++ Functions from Generated Code

External Function Calls from Generated Code	27-2
Calling External Functions from Generated Code	27-2
Why Call External Functions from Generated Code?	27-2
How To Call External Functions	27-2
Pass Arguments by Reference to External Functions	27-3
Manipulate C Data	27-4
Call External Functions Using coder.ceval	27-6
Workflow for Calling External Functions	27-6
Best Practices for Calling External Code from Generated Code	27-7
Return Multiple Values from C Functions	27-8
How MATLAB Coder Infers C/C++ Data Types	27-9
Mapping MATLAB Types to C/C++ Types	27-9
Mapping 64-Bit Integer Types to C/C++	27-10
Mapping Fixed-Point Types to C/C++	27-11
Mapping Arrays to C/C++	27-11
Mapping Complex Values to C/C++	27-12
Mapping Structures to C/C++ Structures	27-13
Mapping Strings to C/C++	27-13
Mapping Multiword Types to C/C++	27-14

External Code Integration for Code Generation	28-2
Encapsulating the Interface to External Code	28-3
Best Practices for Using <code>coder.ExternalDependency</code>	28-4
Terminate Code Generation for Unsupported External Dependency	28-4
Parameterize Methods for MATLAB and Generated Code ..	28-4
Parameterize <code>updateBuildInfo</code> for Multiple Platforms	28-5
Encapsulate Interface to an External C Library	28-6
Update Build Information from MATLAB code	28-9
Call External Functions Encapsulated by <code>coder.ExternalDependency</code>	28-10

Generate Efficient and Reusable Code

Optimization Strategies	29-2
Modularize MATLAB Code	29-5
Eliminate Redundant Copies of Function Inputs	29-6
Inline Code	29-9
Prevent Function Inlining	29-9
Use Inlining in Control Flow Statements	29-9
Control Inlining	29-11
Control Size of Functions Inlined	29-11
Control Size of Functions After Inlining	29-12
Control Stack Size Limit on Inlined Functions	29-12
Fold Function Calls into Constants	29-14

Control Stack Space Usage	29-16
Stack Allocation and Performance	29-17
Dynamic Memory Allocation and Performance	29-18
When Dynamic Memory Allocation Occurs	29-18
Minimize Dynamic Memory Allocation	29-19
Provide Maximum Size for Variable-Size Arrays	29-20
Disable Dynamic Memory Allocation During Code Generation	29-26
Set Dynamic Memory Allocation Threshold	29-27
Set Dynamic Memory Allocation Threshold Using the MATLAB Coder App	29-27
Set Dynamic Memory Allocation Threshold at the Command Line	29-28
Excluding Unused Paths from Generated Code	29-29
Prevent Code Generation for Unused Execution Paths ..	29-30
Prevent Code Generation When Local Variable Controls Flow	29-30
Prevent Code Generation When Input Variable Controls Flow	29-31
Generate Code with Parallel for-Loops (parfor)	29-32
Minimize Redundant Operations in Loops	29-34
Unroll for-Loops	29-36
Limit Copying the for-loop Body in Generated Code	29-36
Support for Integer Overflow and Non-Finites	29-39
Disable Support for Integer Overflow	29-39
Disable Support for Non-Finite Numbers	29-40
Integrate Custom Code	29-41
MATLAB Coder Optimizations in Generated Code	29-47
Constant Folding	29-47

Loop Fusion	29-48
Successive Matrix Operations Combined	29-48
Unreachable Code Elimination	29-49
Generate Reusable Code	29-50
Reuse Large Arrays and Structures	29-51
Variable Reuse in Generated Code	29-52

Generating Reentrant C Code from MATLAB Code

30

Generate Reentrant C Code from MATLAB Code	30-2
About This Tutorial	30-2
Copying Files Locally	30-3
About the Example	30-4
Providing a C main Function	30-5
Configuring Build Parameters	30-6
Generating the C Code	30-7
Viewing the Generated C Code	30-7
Running the Code	30-8
Key Points to Remember	30-8
Learn More	30-9
Reentrant Code	30-10
Specify Generation of Reentrant Code	30-12
Specify Generation of Reentrant Code Using the MATLAB Coder App	30-12
Specify Generation of Reentrant Code Using the Command- Line Interface	30-12
Generated Code API	30-14
Call Reentrant Code in a Single-Thread Environment ...	30-15
Call Reentrant Code in a Multithreaded Environment ...	30-16
Multithreaded Examples	30-16

Call Reentrant Code with No Persistent or Global Data (UNIX Only)	30-17
Provide a Main Function	30-17
Generate Reentrant C Code	30-19
Examine the Generated Code	30-20
Run the Code	30-21
Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)	30-22
MATLAB Code for This Example	30-22
Provide a Main Function	30-23
Generate Reentrant C Code	30-25
Examine the Generated Code	30-25
Run the Code	30-26
Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)	30-27
MATLAB Code for This Example	30-27
Provide a Main Function	30-28
Generate Reentrant C Code	30-30
Examine the Generated Code	30-31
Run the Code	30-32

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[®] Coder[™] generates readable and portable C and C++ code from MATLAB code. It supports most of the MATLAB language and a wide range of toolboxes. You can integrate the generated code into your projects as source code, static libraries, or dynamic libraries. You can also use the generated code within the MATLAB environment to accelerate computationally intensive portions of your MATLAB code. MATLAB Coder lets you incorporate legacy C code into your MATLAB algorithm and into the generated code.

By using MATLAB Coder with Embedded Coder[®], you can further optimize code efficiency and customize the generated code. You can then verify the numerical behavior of the generated code using software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution.

Key Features

- ANSI[®]/ISO[®] compliant C and C++ code generation
- Code generation support for toolboxes including Communications System Toolbox[™], Computer Vision System Toolbox[™], DSP System Toolbox[™], Image Processing Toolbox[™], and Signal Processing Toolbox[™]
- MEX function generation for code verification and acceleration
- Legacy C code integration into MATLAB algorithms and generated code
- Multicore-capable code generation using OpenMP
- Static or dynamic memory-allocation control
- App and equivalent command-line functions for managing code generation projects

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 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:

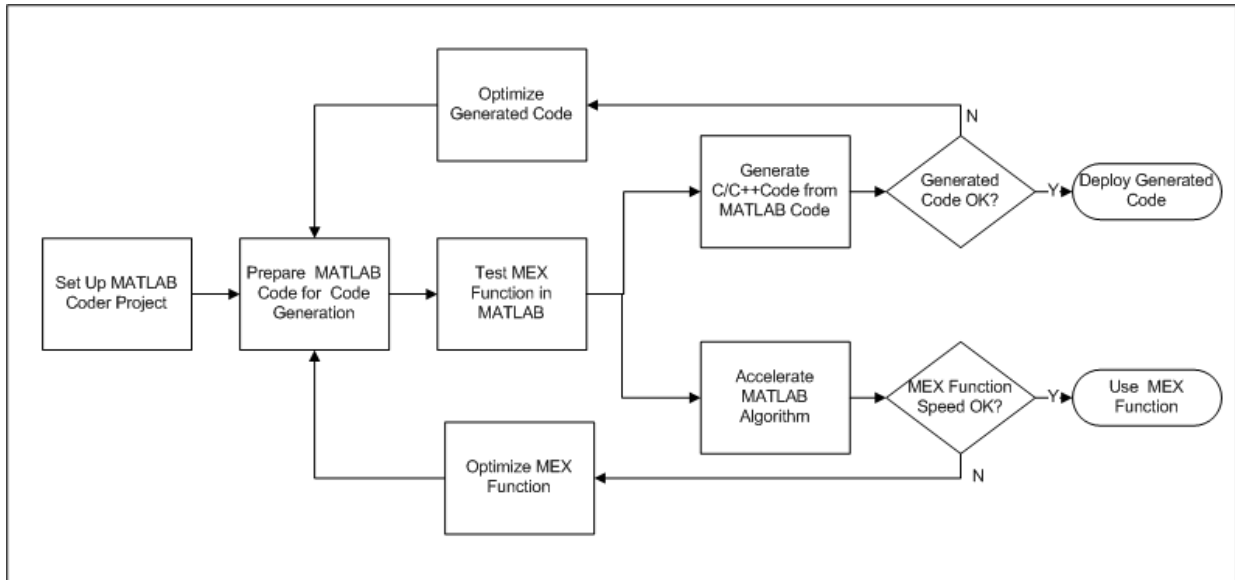
- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize the generated code for a specific target environment.
- 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™ 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

- “Set Up a MATLAB Coder Project” on page 18-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 19-2
- “Workflow for Testing MEX Functions in MATLAB” on page 20-2
- “Code Generation Workflow” on page 21-3
- “Workflow for Accelerating MATLAB Algorithms” on page 26-2

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 2-13

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 Compiler SDK™
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	MATLAB Compiler SDK

To:	Use:
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 app	MATLAB Coder	Try this in “C Code Generation Using the MATLAB Coder App”.
	<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 app	MATLAB Coder	See “Accelerate MATLAB Algorithms” on page 26-14.
	<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” on page 25-12.
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 www.mathworks.com/products/slhdlcoder .

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” on page 7-4
- “Control Dynamic Memory Allocation” on page 21-107
- “Control Run-Time Checks” on page 26-16

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 2-10
“Size of Variable-Size N-D Arrays” on page 2-10
“Size of Empty Arrays” on page 2-10
“Shape of Array When Index and Assigned Value Are Empty” on page 2-10
“Floating-Point Numerical Results” on page 2-11
“NaN and Infinity Patterns” on page 2-12
“Code Generation Target” on page 2-12
“MATLAB Class Initial Values” on page 2-12
“Variable-Size Support for Code Generation” on page 2-12
“Complex Numbers” on page 2-12

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 and Strings” on page 6-9.

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.

- Rewrite

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

as

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

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

- Assign the outputs of a multi-output function call to variables that do not depend on one another. For example, rewrite

```
[y, y.f, y.g] = foo;
```

as

```
[y, a, b] = foo;  
y.f = a;  
y.g = b;
```

- When you access the contents of multiple cells of a cell array, assign the results to variables that do not depend on one another. For example, rewrite

```
[y, y.f, y.g] = z{:};
```

as

```
[y, a, b] = z{:};
```

```
y.f = a;  
y.g = b;
```

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 7-28.

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 7-29.

Shape of Array When Index and Assigned Value Are Empty

When `X` is an empty matrix, generated code does not match MATLAB for the following MATLAB code:

```
A(X) = [];
```

In MATLAB, when `X` is not empty, `A` becomes a row vector. When `X` is empty, `A` does not change. In the generated code, `A` always becomes a row vector, even when `X` is empty.

For example, consider the following function:

```
function A = myfunction()
```



```
A = magic(4);
coder.varsize('A',[20 20],[1 1]);
A([]) = [];
```

In MATLAB, the answer is:

```
16     2     3     13
 5    11    10     8
 9     7     6    12
 4    14    15     1
```

In the generated code, the answer is:

```
16     5     9     4     2    11     7    14     3    10     6    15    13     8    12     1
```

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. These reference implementations can 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

Before code generation, at class loading time, MATLAB computes class initial values. The code generation software uses the value that MATLAB computes. 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” on page 11-4.

Variable-Size Support for Code Generation

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

- “Incompatibility with MATLAB for Scalar Expansion” on page 7-26
- “Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-28
- “Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 7-29
- “Incompatibility with MATLAB in Vector-Vector Indexing” on page 7-31
- “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-31

Complex Numbers

See “Code Generation for Complex Data” on page 6-4.

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

MATLAB Features That Code Generation Supports

Code generation from MATLAB supports the following language features:

- N-dimensional arrays (see “Array Size Restrictions for Code Generation” on page 6-10)
- 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 7-31)
- 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
- 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 “Specify Global Variable Type and Initial Value Using the App” on page 18-26).
- Structures (see “Structure Definition for Code Generation” on page 8-2)
- Cell Arrays (see “Cell Arrays”)

Characters (see “Code Generation for Characters and Strings” on page 6-9)

- Function handles (see “Function Handle Definition for Code Generation” on page 12-2)
- Frames
- Variable length input and output argument lists
- Subset of MATLAB toolbox functions (see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List” on page 4-2)

- Subset of functions and System objects in Aerospace Toolbox, Communications System Toolbox, Computer Vision System Toolbox, DSP System Toolbox, Fixed-Point Designer, Image Processing Toolbox, Signal Processing Toolbox, Phased Array System Toolbox™, Statistics and Machine Learning Toolbox™ (see “Functions and Objects Supported for C and C++ Code Generation — Category List” on page 4-156)
- MATLAB classes (see “MATLAB Classes Definition for Code Generation” on page 11-2)
- Function calls (see “Resolution of Function Calls for Code Generation” on page 14-2)

MATLAB Language Features That Code Generation Does Not Support

Code generation from MATLAB does not support the following frequently used MATLAB constructs:

- Anonymous functions
- Categorical Arrays
- Date and Time Arrays
- Java
- Map Containers
- Nested functions
- Recursion
- Sparse matrices
- Tables
- Time Series objects
- try/catch statements

This list is not exhaustive. To see if a construct is supported for code generation, see “MATLAB Features That Code Generation Supports” on page 2-13.

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"> • <code>imaq.VideoDevice</code>. • “Code Generation with VideoDevice System Object” in the Image Acquisition Toolbox documentation.
Phased Array System Toolbox	“Code Generation” in the Phased Array System Toolbox documentation.
System Identification Toolbox™	“Generate Code for Online Parameter Estimation in MATLAB” in the System Identification 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” on page 4-2 and “Functions and Objects Supported for C and C++ Code Generation — Category List” on page 4-156.

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 “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 4-156

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” on page 4-156.

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	When the input value x is real, but the output should be complex, generates an error during simulation and returns NaN in generated code. To get the complex result, make the input value complex by passing in <code>complex(x)</code> .
acosd	MATLAB	—
acosh	MATLAB	<ul style="list-style-type: none"> 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>.

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” on page 7-35
all	Fixed-Point Designer	—
ambgfun	Phased Array System Toolbox	Does not support variable-size inputs.
and	MATLAB	—
angdiff	Robotics System Toolbox™	Supports MATLAB Function block: No
any	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
any	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
aperture2gain	Phased Array System Toolbox	Does not support variable-size inputs.
asec	MATLAB	—
asecd	MATLAB	—
asech	MATLAB	—
asin	MATLAB	<ul style="list-style-type: none"> 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 $\text{complex}(x)$.
asind	MATLAB	—
asinh	MATLAB	—
assert	MATLAB	<ul style="list-style-type: none"> 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. If called with more than 1 argument, has no effect in standalone code even when run-time error detection is enabled. See “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 22-28. See “Rules for Using assert Function” on page 21-73.
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	—

Name	Product	Remarks and Limitations
atand	MATLAB	—
atanh	MATLAB	<ul style="list-style-type: none"> 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>.
axang2quat	Robotics System Toolbox	Supports MATLAB Function block: No
axang2rotm	Robotics System Toolbox	Supports MATLAB Function block: No
axang2tform	Robotics System Toolbox	Supports MATLAB Function block: No
az2broadside	Phased Array System Toolbox	Does not support variable-size inputs.
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.
bandwidth	MATLAB	—

Name	Product	Remarks and Limitations
barthannwin	Signal Processing Toolbox	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</p>
bboxOverlapRatio	Computer Vision System Toolbox	Supports MATLAB Function block: No
bbox2points	Computer Vision System Toolbox	Supports MATLAB Function block: Yes
beat2range	Phased Array System Toolbox	Does not support variable-size inputs.
besselap	Signal Processing Toolbox	<p>Filter order must be a constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
beta	MATLAB	—

Name	Product	Remarks and Limitations
betacdf	Statistics and Machine Learning Toolbox	—
betafit	Statistics and Machine Learning Toolbox	—
betainc	MATLAB	Always returns a complex result.
betaincinv	MATLAB	Always returns a complex result.
betainv	Statistics and Machine Learning Toolbox	—
betalike	Statistics and Machine Learning Toolbox	—
betaln	MATLAB	—
betapdf	Statistics and Machine Learning Toolbox	—
betarnd	Statistics and Machine Learning 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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
betastat	Statistics and Machine Learning Toolbox	—
bi2de	Communications System Toolbox	—
billingsleyicm	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
bin2dec	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays. Does not match MATLAB when the input is empty.
binaryFeatures	Computer Vision System Toolbox	—
binocdf	Statistics and Machine Learning Toolbox	—
binoinv	Statistics and Machine Learning Toolbox	—
binopdf	Statistics and Machine Learning Toolbox	—
binornd	Statistics and Machine Learning 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"> The output is nonscalar. An input parameter is invalid for the distribution.
binostat	Statistics and Machine Learning Toolbox	—
bitand	MATLAB	—
bitand	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	Fixed-Point Designer	—
bitcmp	MATLAB	—
bitcmp	Fixed-Point Designer	—
bitconcat	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
bitget	MATLAB	—
bitget	Fixed-Point Designer	—
bitor	MATLAB	—
bitor	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	Fixed-Point Designer	—
bitreplicate	Fixed-Point Designer	—
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"> • Generated code may not handle out of range shifting.
bitsra	Fixed-Point Designer	<ul style="list-style-type: none"> • Generated code may not handle out of range shifting.
bitsrl	Fixed-Point Designer	<ul style="list-style-type: none"> • Generated code may not handle out of range shifting.
bitxor	MATLAB	—

Name	Product	Remarks and Limitations
bitxor	Fixed-Point Designer	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	Fixed-Point Designer	—
blackman	Signal Processing Toolbox	<p>Window length must be a constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</p>

Name	Product	Remarks and Limitations
break	MATLAB	—
BRISKPoints	Computer Vision System Toolbox	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> , 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.
broadside2az	Phased Array System Toolbox	Does not support variable-size inputs.
bsxfun	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
buttap	Signal Processing Toolbox	Filter order must be a constant. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
butter	Signal Processing Toolbox	Filter coefficients must be constants. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.

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>Specifying constants</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” on page 21-53.</p>
bwareaopen	Image Processing Toolbox	<p><code>BW</code> must be a 2-D binary image. N-D arrays are not supported. <code>conn</code> can only be one of the two-dimensional connectivities (4 or 8) or a 3-by-3 matrix. The 3-D connectivities (6, 18, and 26) are not supported. Matrices of size 3-by-3-by-...-by-3 are not supported. <code>conn</code> must be a compile-time constant.</p>
bwdist	Image Processing Toolbox	<p>The <code>method</code> argument must be a compile-time constant. Input images must have fewer than 2^{32} pixels.</p> <p>Generated code for this function uses a precompiled, platform-specific shared library.</p>
bweuler	Image Processing Toolbox	<p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, platform-specific shared library</p>
bwlabel	Image Processing Toolbox	<p>When generating code, the parameter <code>n</code> must be a compile-time constant.</p>
bwlookup	Image Processing Toolbox	<p>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>

Name	Product	Remarks and Limitations
bwmorph	Image Processing Toolbox	<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 MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
bwpack	Image Processing Toolbox	Generated code for this function uses a precompiled platform-specific shared library.
bwperim	Image Processing Toolbox	<p>Supports only 2-D images. Does not support any no-output-argument syntaxes. The connectivity matrix input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code 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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
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.

Name	Product	Remarks and Limitations
cameraMatrix	Computer Vision System Toolbox	Supports MATLAB Function block: No
cameraParameters	Computer Vision System Toolbox	Supports MATLAB Function block: No
cameraPose	Computer Vision System Toolbox	Supports MATLAB Function block: No
cart2hom	Robotics System Toolbox	Supports MATLAB Function block: No
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"> • Does not support concatenation of cell arrays. • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
cbfweights	Phased Array System Toolbox	Does not support variable-size inputs.
cdf	Statistics and Machine Learning Toolbox	—
ceil	MATLAB	—
ceil	Fixed-Point Designer	—
cell	MATLAB	“Cell Array Requirements and Limitations for Code Generation” on page 9-10

Name	Product	Remarks and Limitations
cfirpm	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
char	MATLAB	<p>Does not support cell arrays.</p>
cheb1ap	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
cheb1ord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>

Name	Product	Remarks and Limitations
cheb2ap	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
cheb2ord	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
chebwin	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>

Name	Product	Remarks and Limitations
cheby1	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
cheby2	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
chi2cdf	Statistics and Machine Learning Toolbox	—
chi2inv	Statistics and Machine Learning Toolbox	—
chi2pdf	Statistics and Machine Learning Toolbox	—
chi2rnd	Statistics and Machine Learning 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"> • The output is nonscalar. • An input parameter is invalid for the distribution.

Name	Product	Remarks and Limitations
chi2stat	Statistics and Machine Learning Toolbox	—
chol	MATLAB	—
circpol2pol	Phased Array System Toolbox	Does not support variable-size inputs.
circshift	MATLAB	Does not support cell arrays for the first argument.
cl2tf	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
class	MATLAB	—
colon	MATLAB	<ul style="list-style-type: none"> • Does not accept complex inputs. • The input <i>i</i> cannot have a logical value. • Does not accept vector inputs. • Inputs must be constants. • Uses single-precision arithmetic to produce single-precision results.
comm.ACPR	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.AGC	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.AlgebraicDeinterleaver	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”

Name	Product	Remarks and Limitations
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”
comm.CarrierSynchronizer	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.CCDF	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.CoarseFrequency-Compensator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.ConstellationDiagram	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.Convolutional-Deinterleaver	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.ConvolutionalEncoder	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”

Name	Product	Remarks and Limitations
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.DifferentialDecoder	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.DifferentialEncoder	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.ErrorRate	Communications System Toolbox	"System Objects in MATLAB Code Generation"
comm.EVM	Communications System Toolbox	"System Objects in MATLAB Code Generation"

Name	Product	Remarks and Limitations
comm.FMBroadcast-Demodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.FMBroadcast-Modulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.FMDemodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.FMModulator	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.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”
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”

Name	Product	Remarks and Limitations
comm.HDLCRCGenerator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.HDLRSDDecoder	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.HDLRSEncoder	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.HelicalDeinterleaver	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”
comm.LTEMIMOChannel	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.MatrixDeinterleaver	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”

Name	Product	Remarks and Limitations
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.Multiplexed-Deinterleaver	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.MultiplexedInterleaver	Communications System Toolbox	“System Objects in MATLAB Code Generation”
comm.OFDMDemodulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”
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”

Name	Product	Remarks and Limitations
comm.PhaseRequencyOffset	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.PSKCoarseFrequency-Estimator	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”
comm.QAMCoarseFrequency-Estimator	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.RectangularModulator	Communications System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
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	Communications System Toolbox
comm.SymbolSynchronizer	Communications System Toolbox	Communications System Toolbox
comm.ThermalNoise	Communications System Toolbox	“System Objects in MATLAB Code Generation”
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	—

Name	Product	Remarks and Limitations
computer	MATLAB	<ul style="list-style-type: none"> Information about the computer on which the code generation software is running. Use only when the code generation target is S-function (Simulation) or MEX-function.
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Designer	—
conndef	Image Processing Toolbox	Input arguments must be compile-time constants.
continue	MATLAB	—
conv	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
conv	Fixed-Point Designer	<ul style="list-style-type: none"> 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>. For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. 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>.
conv2	MATLAB	—

Name	Product	Remarks and Limitations
convergent	Fixed-Point Designer	—
convn	MATLAB	—
cordicabs	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicangle	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicatan2	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordiccart2pol	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicccexp	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicccos	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicpol2cart	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicrotate	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicsin	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicsincos	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cornerPoints	Computer Vision System Toolbox	<p>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>visionRecovertfomCodeGeneration_kernel.m</code>, which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.</p>

Name	Product	Remarks and Limitations
corrcoef	MATLAB	<ul style="list-style-type: none"> Row-vector input is only supported when the first two inputs are vectors and nonscalar.
cos	MATLAB	—
cos	Fixed-Point Designer	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	<ul style="list-style-type: none"> In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>. In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.
coth	MATLAB	—
cov	MATLAB	<ul style="list-style-type: none"> “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 Does not support the <code>nanflag</code> argument.
cross	MATLAB	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
csc	MATLAB	—
cscd	MATLAB	<ul style="list-style-type: none"> In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>. In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.
csch	MATLAB	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Designer	—
cummin	MATLAB	—

Name	Product	Remarks and Limitations
cummax	MATLAB	—
cumprod	MATLAB	Does not support logical inputs. Cast input to <code>double</code> first.
cumsum	MATLAB	Does not support logical inputs. Cast input to <code>double</code> first.
cumtrapz	MATLAB	—
db2pow	Signal Processing Toolbox	—
dct	Signal Processing Toolbox	<ul style="list-style-type: none"> • Code generation for this function requires the DSP System Toolbox software. • Length of transform dimension must be a power of two. If specified, the <code>pad</code> or <code>truncation</code> value must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</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” on page 21-53.</p>
de2bi	Communications System Toolbox	—
deal	MATLAB	—
deblank	MATLAB	<ul style="list-style-type: none"> • Supports only inputs from the <code>char</code> class. Does not support cell arrays. • Input values must be in the range 0-127.

Name	Product	Remarks and Limitations
dec2bin	MATLAB	<ul style="list-style-type: none"> • If input <i>d</i> is double, <i>d</i> must be less than 2^{52}. • If input <i>d</i> is single, <i>d</i> must be less than 2^{23}. • Unless you specify input <i>n</i> to be constant and <i>n</i> 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, <i>n</i> must be at least 52 for double, 23 for single, 16 for char, 32 for int32, 16 for int16, and so on.
dec2hex	MATLAB	<ul style="list-style-type: none"> • If input <i>d</i> is double, <i>d</i> must be less than 2^{52}. • If input <i>d</i> is single, <i>d</i> must be less than 2^{23}. • Unless you specify input <i>n</i> to be constant, and <i>n</i> 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, <i>n</i> must be at least 13 for double, 6 for single, 4 for char, 8 for int32, 4 for int16, and so on.
dechirp	Phased Array System Toolbox	Does not support variable-size inputs.
deconv	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
del2	MATLAB	—
delayseq	Phased Array System Toolbox	Does not support variable-size inputs.
depressionang	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
det	MATLAB	—
detectBRISKFeatures	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectCheckerboard-Points	Computer Vision System Toolbox	Supports MATLAB Function block: No Code generation will not support specifying images as file names or cell arrays of file names. It supports only checkerboard detection in a single image or stereo pair of images. For example, these syntaxes are supported: <ul style="list-style-type: none"> • detectCheckerboardPoints(I1) • detectCheckerbarPoints(I1,I2) I1 and I2 are single grayscale or RGB images.
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: FilterSize Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
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.

Name	Product	Remarks and Limitations
detrend	MATLAB	<ul style="list-style-type: none">• If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements:<ul style="list-style-type: none">• Be real.• Be sorted in ascending order.• Restrict elements to integers in the interval $[1, n-2]$. n is the number of elements in a column of input argument X, or the number of elements in X when X is a row vector.• Contain all unique values.• “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Name	Product	Remarks and Limitations
diag	MATLAB	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value. • For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), diag: <ul style="list-style-type: none"> • Treats the input as a vector input. • Returns a matrix with the given vector along the specified diagonal. • For variable-size inputs that are not variable-length vectors, diag: <ul style="list-style-type: none"> • Treats the input as a matrix. • Does not support inputs that are vectors at run time. • Returns a variable-length vector. <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> • For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), diag 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 diag to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use: <ul style="list-style-type: none"> • <code>diag(x(:))</code> instead of <code>diag(x)</code> • <code>diag(x(:),k)</code> instead of <code>diag(x,k)</code> • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Name	Product	Remarks and Limitations
diag	Fixed-Point Designer	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
diff	MATLAB	<ul style="list-style-type: none"> 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. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
disparity	Computer Vision System Toolbox	Compile-time constant input: <code>Method</code> . 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"> 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. Complex and imaginary divisors are not supported. The syntax <code>T.divide(a,b)</code> is not supported.
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	All inputs must be constants. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
dsp.AdaptiveLatticeFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AffineProjectionFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AllpassFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation” The System object™ supports code generation only when the <code>Structure</code> property is set to <code>Minimum multiplier</code> or <code>Lattice</code> .
dsp.AllpoleFilter	DSP System Toolbox	<ul style="list-style-type: none"> • “System Objects in MATLAB Code Generation” • Only the <code>Denominator</code> property is tunable for code generation.
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.ArrayVectorMultiplier	DSP System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.ArrayVectorSubtractor	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.AudioFileReader	DSP System Toolbox	“System Objects in MATLAB Code Generation”
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.BurgSpectrumEstimator	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”

Name	Product	Remarks and Limitations
dsp.Crosscorrelator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.CrossSpectrumEstimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
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.DigitalDownConverter	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"> • “System Objects in MATLAB Code Generation” • The <code>SOSMatrix</code> and <code>Scalevalues</code> properties are not supported for code generation.
dsp.FarrowRateConverter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FastTransversalFilter	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"> 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. <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"> “System Objects in MATLAB Code Generation” Only the <code>Numerator</code> property is tunable for code generation.
dsp.FIRHalfbandDecimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.FIRHalfbandInterpolator	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.HighpassFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Histogram	DSP System Toolbox	<ul style="list-style-type: none"> This object has no tunable properties for code generation. “System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
dsp.IDCT	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.IFFT	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.IIRFilter	DSP System Toolbox	<ul style="list-style-type: none"> • Only the Numerator and Denominator properties are tunable for code generation. • “System Objects in MATLAB Code Generation”
dsp.IIRHalfbandDecimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.IIRHalfbandInterpolator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
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.LowerTriangularSolver	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LowpassFilter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.LPCToAutocorrelation	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”

Name	Product	Remarks and Limitations
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”
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”

Name	Product	Remarks and Limitations
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”
dsp.SampleRateConverter	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ScalarQuantizerDecoder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.ScalarQuantizerEncoder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.SignalSource	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.SineWave	DSP System Toolbox	<ul style="list-style-type: none"> • This object has no tunable properties for code generation. • “System Objects in MATLAB Code Generation”
dsp.SpectrumAnalyzer	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.
dsp.SpectrumEstimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.StandardDeviation	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.StateLevels	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.TimeScope	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.

Name	Product	Remarks and Limitations
dsp.TransferFunction-Estimator	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.UDPReceiver	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.UDPSender	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.UpperTriangularSolver	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VariableFractionDelay	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VariableIntegerDelay	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Variance	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VectorQuantizerDecoder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.VectorQuantizerEncoder	DSP System Toolbox	“System Objects in MATLAB Code Generation”
dsp.Window	DSP System Toolbox	<ul style="list-style-type: none"> • This object has no tunable properties for code generation. • “System Objects in MATLAB Code Generation”
dsp.ZeroCrossingDetector	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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>

Name	Product	Remarks and Limitations
effearthradius	Phased Array System Toolbox	Does not support variable-size inputs.
eig	MATLAB	<ul style="list-style-type: none"> • For code generation, QZ algorithm is used in all cases. MATLAB can use different algorithms for different inputs. Consequently, V might represent a different basis of eigenvectors. The eigenvalues in D might not be in the same order as in MATLAB. • With one input, $[V,D] = \text{eig}(A)$, the results are similar to those obtained using $[V,D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')$ in MATLAB, except that for code generation, the columns of V are normalized. • Options 'balance', and 'nobalance' are not supported for the standard eigenvalue problem. 'chol' is not supported for the symmetric generalized eigenvalue problem. • Outputs are of complex type. • Does not support the option to calculate left eigenvectors.
ellip	Signal Processing Toolbox	<p>Inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</p>
end	MATLAB	—
end	Fixed-Point Designer	—
epipolarLine	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions.</p> <p>Supports MATLAB Function block: Yes</p>
eps	MATLAB	—
eps	Fixed-Point Designer	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	MATLAB	—
eq	Fixed-Point Designer	Not supported for fixed-point signals with different biases.
erf	MATLAB	—

Name	Product	Remarks and Limitations
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—
error	MATLAB	Has no effect in standalone code even when run-time error detection is enabled. See “Run-Time Error Detection and Reporting in Standalone C/ C++ Code” on page 22-28.
espritdoa	Phased Array System Toolbox	Does not support variable-size inputs.
estimateFundamental-Matrix	Computer Vision System Toolbox	Compile-time constant input: <code>Method</code> , <code>OutputClass</code> , <code>DistanceType</code> , and <code>ReportRuntimeError</code> . Supports MATLAB Function block: Yes
estimateGeometric-Transform	Computer Vision System Toolbox	Compile-time constant input: <code>transformType</code> Supports MATLAB Function block: No
estimateUncalibrated-Rectification	Computer Vision System Toolbox	Supports MATLAB Function block: Yes
eul2quat	Robotics System Toolbox	Supports MATLAB Function block: No
eul2rotm	Robotics System Toolbox	Supports MATLAB Function block: No
eul2tform	Robotics System Toolbox	Supports MATLAB Function block: No
evcdf	Statistics and Machine Learning Toolbox	—
evinv	Statistics and Machine Learning Toolbox	—
evpdf	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
evrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
evstat	Statistics and Machine Learning Toolbox	—
exp	MATLAB	—
expcdf	Statistics and Machine Learning Toolbox	—
expint	MATLAB	—
expinv	Statistics and Machine Learning Toolbox	—
expm	MATLAB	—
expm1	MATLAB	—
exppdf	Statistics and Machine Learning Toolbox	—
exprnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
expstat	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
extractFeatures	Computer Vision System Toolbox	Generates platform-dependent library: Yes for BRISK, FREAK, and SURF methods only. Compile-time constant input restrictions: '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	Supports MATLAB Function block: No
extractLBPFeatures	Computer Vision System Toolbox	Generates platform-dependent library: No Supports MATLAB Function block: Yes
extrinsics	Computer Vision System Toolbox	Supports MATLAB Function block: No
eye	MATLAB	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size.
factor	MATLAB	<ul style="list-style-type: none"> • The maximum double precision input is 2^{33}. • The maximum single precision input is 2^{25}. • The input <code>n</code> cannot have type <code>int64</code> or <code>uint64</code>.
factorial	MATLAB	—
false	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
fcdf	Statistics and Machine Learning Toolbox	—
fclose	MATLAB	—
feof	MATLAB	—

Name	Product	Remarks and Limitations
fft	MATLAB	<ul style="list-style-type: none"> The length of the input vector must be a power of 2. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
fft2	MATLAB	<ul style="list-style-type: none"> The length of the input matrix dimensions must each be a power of 2.
fftn	MATLAB	<ul style="list-style-type: none"> The <code>siz</code> argument must have a fixed size. The length of the input matrix dimensions must each be a power of 2.
fftshift	MATLAB	—
fi	Fixed-Point Designer	<ul style="list-style-type: none"> Use to create a fixed-point constant or variable. The default constructor syntax without input arguments is not supported. The <code>rand</code> <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>. If the input value is not known at compile time, you must provide complete <code>numericType</code> information. All properties related to data type must be constant for code generation. <code>numericType</code> object information must be available for non-fixed-point Simulink inputs.
fieldnames	MATLAB	Does not support objects. The input must be a structure.

Name	Product	Remarks and Limitations
filter	MATLAB	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
filter	Fixed-Point Designer	<ul style="list-style-type: none"> • 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>.
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>Specifying constants</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” on page 21-53.</p>
fimath	Fixed-Point Designer	<ul style="list-style-type: none"> • 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. • Use to create <code>fimath</code> objects in generated code. • 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. • 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.

Name	Product	Remarks and Limitations
find	MATLAB	<ul style="list-style-type: none"> Issues an error if a variable-size input becomes a row vector at run time. <hr/> <p>Note: This limitation does not apply when the input is scalar or a variable-length row vector.</p> <hr/> <ul style="list-style-type: none"> For variable-size 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. Always returns a variable-length vector. Even when you provide the output vector k, the output cannot be fixed-size because the output can contain fewer than k elements. For example, <code>find(x,1)</code> returns a variable-length vector with 1 or 0 elements.
findpeaks	Signal Processing Toolbox	—
finv	Statistics and Machine Learning Toolbox	—
fir1	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</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>Specifying constants</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” on page 21-53.</p>
fircls1	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
fireqint	DSP System Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p>

Name	Product	Remarks and Limitations
firgr	DSP System Toolbox	<ul style="list-style-type: none"> All inputs must be constant. Expressions or variables are allowed if their values do not change. Does not support syntaxes that have cell array input.
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"> All inputs must be constant. Expressions or variables are allowed if their values do not change. Does not support syntaxes that have cell array input.
firls	Signal Processing Toolbox	<p>All inputs must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</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.
firpr2chfb	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Product	Remarks and Limitations
firpm	Signal Processing Toolbox	All inputs must be constants. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
firpmord	Signal Processing Toolbox	All inputs must be constants. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
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>Specifying constants</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” on page 21-53.</p>
flintmax	MATLAB	—
flip	MATLAB	Does not support cell arrays for the first argument.
flip	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
flipdim	MATLAB	Does not support cell arrays for the first argument.
fliplr	MATLAB	Does not support cell arrays.
fliplr	Fixed-Point Designer	—
flipud	MATLAB	Does not support cell arrays.
flipud	Fixed-Point Designer	—
floor	MATLAB	—
floor	Fixed-Point Designer	—
fminsearch	MATLAB	<ul style="list-style-type: none"> • Ignores the <code>Display</code> option. Does not print status information during execution. Test the <code>exitflag</code> output for the exit condition. • The output structure does not include the <code>algorithm</code> or <code>message</code> fields. • Ignores the <code>OutputFcn</code> and <code>PlotFcns</code> options.

Name	Product	Remarks and Limitations
fopen	MATLAB	<ul style="list-style-type: none"> • Does not support: <ul style="list-style-type: none"> • machineformat, encoding, or fileID inputs • message output • fopen('all') • If you disable extrinsic calls, you cannot return fileIDs created with fopen to MATLAB or extrinsic functions. You can use such fileIDs only internally. • When generating C/C++ executables, static libraries, or dynamic libraries, you can open up to 20 files. • 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> • The behavior of the generated code for fread is compiler-dependent if you: <ol style="list-style-type: none"> 1 Open a file using fopen with a permission of a+. 2 Read the file using fread before calling an I/O function, such as fseek or

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

Name	Product	Remarks and Limitations
fprintf	MATLAB	<ul style="list-style-type: none"> • Does not support: <ul style="list-style-type: none"> • <code>b</code> and <code>t</code> subtypes on <code>%u</code>, <code>%o</code> <code>%x</code>, and <code>%X</code> formats. • <code>\$</code> flag for reusing input arguments. • printing arrays. • There is no automatic casting. Input arguments must match their format types for predictable results. • Escaped characters are limited to the decimal range of 0–127. • 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"> • The <code>fprintf</code> call is inside a <code>parfor</code> loop. • Extrinsic calls are disabled. • 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"> • The format specifier has a corresponding C format specifier, for example, <code>%e</code> or <code>%E</code>. • The <code>printf</code> call is inside a <code>parfor</code> loop. • Extrinsic calls are disabled. • 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. • 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>

Name	Product	Remarks and Limitations
		int64(n), 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"> • precision must be a constant. • The source and output that precision specifies cannot have values long, ulong, unsigned long, bitN, or ubitN. • You cannot use the machineformat input. • 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"> • Match. • Map directly to a MATLAB type. • The source type that precision specifies must map directly to a C type on the target hardware. • If the fread call reads the entire file, all of the data must fit in the largest array available for code generation. • If sizeA is not constant or contains a nonfinite element, then dynamic memory allocation is required. • Treats a char value for source or output as a signed 8-bit integer. Use values between 0 and 127 only. • 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>

Name	Product	Remarks and Limitations
		<p><code>end</code></p> <p><code>...</code></p>
freqspace	MATLAB	—
freqz	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • 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>If the semicolon is added, the function produces a plot of the magnitude and phase response of the filter.</p> <p>See “freqz With No Output Arguments”.</p>
frewind	MATLAB	—
frnd	Statistics and Machine Learning 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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
fspecial	Image Processing Toolbox	<p>Inputs must be compile-time constants. Expressions or variables are allowed if their values do not change.</p>
fsp1	Phased Array System Toolbox	Does not support variable-size inputs.
fstat	Statistics and Machine Learning Toolbox	—
full	MATLAB	—

Name	Product	Remarks and Limitations
fzero	MATLAB	<ul style="list-style-type: none"> • The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument. • Supports up to three output arguments. Does not support the fourth output argument (the output structure).
gain2aperture	Phased Array System Toolbox	Does not support variable-size inputs.
gamcdf	Statistics and Machine Learning Toolbox	—
gaminv	Statistics and Machine Learning Toolbox	—
gamma	MATLAB	—
gammainc	MATLAB	Output is always complex.
gammaincinv	MATLAB	Output is always complex.
gammaIn	MATLAB	—
gampdf	Statistics and Machine Learning Toolbox	—
gamrnd	Statistics and Machine Learning 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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
gamstat	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
gausswin	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
gccphat	Phased Array System Toolbox	Does not support variable-size inputs.
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
generateCheckerboard-Points	Computer Vision System Toolbox	Supports MATLAB Function block: No
geocdf	Statistics and Machine Learning Toolbox	—
geoinv	Statistics and Machine Learning Toolbox	—
geomean	Statistics and Machine Learning Toolbox	—
geopdf	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
geornd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
geostat	Statistics and Machine Learning Toolbox	—
get	Fixed-Point Designer	<ul style="list-style-type: none"> • The syntax <code>structure = get(o)</code> is not supported.
getlsb	Fixed-Point Designer	—
getmsb	Fixed-Point Designer	—
getrangefromclass	Image Processing Toolbox	—
gevcdf	Statistics and Machine Learning Toolbox	—
gevinv	Statistics and Machine Learning Toolbox	—
gevpdf	Statistics and Machine Learning Toolbox	—
gevrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.

Name	Product	Remarks and Limitations
gevstat	Statistics and Machine Learning Toolbox	—
global2localcoord	Phased Array System Toolbox	Does not support variable-size inputs.
gpcdf	Statistics and Machine Learning Toolbox	—
gpinv	Statistics and Machine Learning Toolbox	—
gppdf	Statistics and Machine Learning Toolbox	—
gprnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
gpstat	Statistics and Machine Learning Toolbox	—
gradient	MATLAB	—
grayconnected	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
grazingang	Phased Array System Toolbox	Does not support variable-size inputs.
gt	MATLAB	—
gt	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
hadamard	MATLAB	n must be a fixed-size scalar.

Name	Product	Remarks and Limitations
hamming	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
hankel	MATLAB	—
hann	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
harmmean	Statistics and Machine Learning Toolbox	—
hdl.RAM	MATLAB	—
hex2dec	MATLAB	Does not support cell arrays.
hex2num	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays. • For $n = \text{hex2num}(S)$, $\text{size}(S, 2) \leq \text{length}(\text{num2hex}(0))$
hilb	MATLAB	—

Name	Product	Remarks and Limitations
hist	MATLAB	<ul style="list-style-type: none"> • Histogram bar plotting not supported. Call with at least one output argument. • If supplied, the second argument <code>x</code> must be a scalar constant. • Inputs must be real. <p>For the syntax <code>[nout, xout] = hist(y,x)</code>:</p> <ul style="list-style-type: none"> • When <code>y</code> is a fixed-size vector or variable-length vector: <ul style="list-style-type: none"> • <code>nout</code> is always a row vector. • If <code>x</code> is a vector, <code>xout</code> is a vector with the same orientation as <code>x</code>. • If <code>x</code> is a scalar (fixed-size), <code>xout</code> is a row vector. • <code>nout</code> and <code>xout</code> are column vectors when the following conditions are true: <ul style="list-style-type: none"> • <code>y</code> is a matrix • <code>size(y,1)</code> and <code>size(y,2)</code> do not have fixed length 1 • One of <code>size(y,1)</code> and <code>size(y,2)</code> has length 1 at run time • A variable-size <code>x</code> is interpreted as a vector input even if it is a scalar at run time. • If at least one of the inputs is empty, vector orientations in the output can differ from MATLAB.

Name	Product	Remarks and Limitations
histc	MATLAB	<ul style="list-style-type: none"> • The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector. • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
hom2cart	Robotics System Toolbox	Supports MATLAB Function block: No
horizonrange	Phased Array System Toolbox	Does not support variable-size inputs.
horzcat	Fixed-Point Designer	—
hough	Image Processing Toolbox	The optional parameter names 'Theta' and 'RhoResolution' must be compile-time string constants. The optional Theta vector must have a bounded size.
houghlines	Image Processing Toolbox	The optional parameter names 'FillGap' and 'MinLength' must be compile-time string constants. Their associated values need not be compile-time constants.
houghpeaks	Image Processing Toolbox	The optional parameter names 'Threshold' and 'NHoodSize' must be compile-time string constants. Their associated values need not be compile-time constants.

Name	Product	Remarks and Limitations
hygecdf	Statistics and Machine Learning Toolbox	—
hygeinv	Statistics and Machine Learning Toolbox	—
hygepdf	Statistics and Machine Learning Toolbox	—
hygernd	Statistics and Machine Learning 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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
hygestat	Statistics and Machine Learning Toolbox	—
hypot	MATLAB	—
icdf	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
idct	Signal Processing Toolbox	<ul style="list-style-type: none"> • Code generation for this function requires the 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. <p>Specifying constants</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” on page 21-53.</p>
if, elseif, else	MATLAB	—
idivide	MATLAB	<ul style="list-style-type: none"> • For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.
ifft	MATLAB	<ul style="list-style-type: none"> • The length of the input vector must be a power of 2. • Output of <code>ifft</code> block is complex. • Does not support the 'symmetric' option. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
ifft2	MATLAB	<ul style="list-style-type: none"> • The length of the input matrix dimensions must each be a power of 2. • Does not support the 'symmetric' option.
ifftn	MATLAB	<ul style="list-style-type: none"> • Does not support the 'symmetric' option. • The <code>siz</code> argument must have a fixed size. • The length of the input matrix dimensions must each be a power of 2.
ifftshift	MATLAB	—

Name	Product	Remarks and Limitations
<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"> All inputs must be constant. Expressions or variables are allowed if their values do not change. Does not support syntaxes that have cell array input.
<code>iirlpnorm</code>	DSP System Toolbox	<ul style="list-style-type: none"> All inputs must be constant. Expressions or variables are allowed if their values do not change. Does not support syntaxes that have cell array input.
<code>iirlpnormc</code>	DSP System Toolbox	<ul style="list-style-type: none"> All inputs must be constant. Expressions or variables are allowed if their values do not change. Does not support syntaxes that have cell array input.
<code>iirnotch</code>	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>iirparameq</code>	DSP System Toolbox	—
<code>iirpeak</code>	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
<code>im2double</code>	MATLAB	—
<code>im2int16</code>	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.

Name	Product	Remarks and Limitations
im2single	Image Processing Toolbox	—
im2uint8	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
im2uint16	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
imabsdiff	Image Processing Toolbox	—
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>If you choose the generic MATLAB Host Computer target platform, generated code 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”
imbothat	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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>

Name	Product	Remarks and Limitations
imclearborder	Image Processing Toolbox	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant. Supports only up to 3-D inputs.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imclose	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>If you choose the generic MATLAB Host Computer target platform, generated code 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>
imcrop	Image Processing Toolbox	<p>The interactive syntaxes, such as <code>I2 = imcrop</code>, are not supported. Indexed images are not supported, including the noninteractive syntax <code>X2 = imcrop(X,map,rect);</code>.</p>
imdilate	Image Processing Toolbox	<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 MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>

Name	Product	Remarks and Limitations
imerode	Image Processing Toolbox	<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>
imextendedmax	Image Processing Toolbox	<p>The optional third input argument, <code>conn</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>
imextendedmin	Image Processing Toolbox	<p>The optional third input argument, <code>conn</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>

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.</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 P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-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>If you choose the generic MATLAB Host Computer target platform, generated code 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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imgaborfilt	Image Processing Toolbox	—

Name	Product	Remarks and Limitations
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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imhmax	Image Processing Toolbox	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code 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>If you choose the generic MATLAB Host Computer target platform, generated code 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. You can specify up to four input image arguments.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
immse	Image Processing Toolbox	—
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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>

Name	Product	Remarks and Limitations
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>If you choose the generic MATLAB Host Computer target platform, generated code 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>
imref3d	Image Processing Toolbox	<p>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.</p>
imregionalmax	Image Processing Toolbox	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imregionalmin	Image Processing Toolbox	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imresize	Image Processing Toolbox	<p>Does not support indexed images or custom interpolation kernels. The <code>Scale</code>, <code>OutputSize</code> and all parameter-value pair input arguments must be compile-time constants.</p>

Name	Product	Remarks and Limitations
imrotate	Image Processing Toolbox	<p>The angle, method, and bbox arguments must be compile-time constants.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imtophat	Image Processing Toolbox	<p>The input image IM must be either 2-D or 3-D image. The structuring element input argument SE must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imtranslate	Image Processing Toolbox	<p>The supports only 2-D translation vectors. 3-D translations are not supported</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imwarp	Image Processing Toolbox	<p>The geometric transformation object input, tform, must be either affine2d or projective2d. Additionally, the interpolation method and optional parameter names must be string constants.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
ind2sub	MATLAB	<ul style="list-style-type: none"> • The first argument should be a valid size vector. Size vectors for arrays with more than intmax elements are not supported. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
inf	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.

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
insertText	Computer Vision System Toolbox	Compile-time constant input: Font , FontSize Supports non-ASCII characters: No Supports MATLAB Function block: Yes
int8, int16, int32, int64	MATLAB	No integer overflow detection for int64 in MEX or MATLAB Function block simulation on Windows 32-bit platforms.
int8, int16, int32, int64	Fixed-Point Designer	—
integralBoxFilter	Image Processing Toolbox	The FilterSize input argument and the 'NormalizationFactor' parameter must be compile-time constants.
integralImage	Computer Vision System Toolbox	Supports MATLAB Function block: Yes
interp1	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
interp1q	MATLAB	Might not match MATLAB when some Y values are Inf or NaN.

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

Name	Product	Remarks and Limitations
intersect	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • 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. • 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. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • Sort the real input in complex ascending order (by absolute value). Suppose the real input

Name	Product	Remarks and Limitations
		is x . Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code> . <ul style="list-style-type: none"> • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
intfilt	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
intlut	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code 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	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
ipermute	Fixed-Point Designer	—
iptcheckconn	Image Processing Toolbox	Input arguments must be compile-time constants.
iptcheckmap	Image Processing Toolbox	—

Name	Product	Remarks and Limitations
iqcoef2imbal	Communications System Toolbox	—
iqimbal2coef	Communications System Toolbox	—
iqr	Statistics and Machine Learning Toolbox	—
isa	MATLAB	—
isbanded	MATLAB	—
iscell	MATLAB	—
iscellstr	MATLAB	—
ischar	MATLAB	—
iscolumn	MATLAB	—
iscolumn	Fixed-Point Designer	—
isdeployed	MATLAB Compiler	<ul style="list-style-type: none"> • Returns true and false as appropriate for MEX and SIM targets • Returns false for other targets
isdiag	MATLAB	—
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	—

Name	Product	Remarks and Limitations
isfield	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays for the second argument
isfimath	Fixed-Point Designer	—
isfimathlocal	Fixed-Point Designer	—
isfinite	MATLAB	—
isfinite	Fixed-Point Designer	—
isfloat	MATLAB	—
ishermitian	MATLAB	—
isinf	MATLAB	—
isinf	Fixed-Point Designer	—
isinteger	MATLAB	—
isletter	MATLAB	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0-127.
islogical	MATLAB	—
ismac	MATLAB	<ul style="list-style-type: none"> Returns true or false based on the MATLAB version used for code generation. Use only when the code generation target is S-function (Simulation) or MEX-function.
ismatrix	MATLAB	—
ismcc	MATLAB Compiler	<ul style="list-style-type: none"> Returns true and false as appropriate for MEX and SIM targets. Returns false for other targets.

Name	Product	Remarks and Limitations
ismember	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • The second input, B, must be sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
isnan	MATLAB	—
isnan	Fixed-Point Designer	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Designer	—
isnumericity	Fixed-Point Designer	—
isobject	MATLAB	—
ispc	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version you use for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.
isprime	MATLAB	<ul style="list-style-type: none"> • The maximum double precision input is 2^{33}. • The maximum single precision input is 2^{25}. • The input X cannot have type <code>int64</code> or <code>uint64</code>.
isreal	MATLAB	—
isreal	Fixed-Point Designer	—
isrow	MATLAB	—

Name	Product	Remarks and Limitations
isrow	Fixed-Point Designer	—
isscalar	MATLAB	—
isscalar	Fixed-Point Designer	—
assigned	Fixed-Point Designer	—
issorted	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
isspace	MATLAB	<ul style="list-style-type: none"> • Input values from the <code>char</code> class must be in the range 0–127.
issparse	MATLAB	—
isstrprop	MATLAB	<ul style="list-style-type: none"> • Supports only inputs from <code>char</code> and <code>integer</code> classes. • Input values must be in the range 0-127.
isstruct	MATLAB	—
issymmetric	MATLAB	—
istrellis	Communications System Toolbox	—
istril	MATLAB	—
istriu	MATLAB	—
isunix	MATLAB	<ul style="list-style-type: none"> • Returns true or false based on the MATLAB version used for code generation. • Use only when the code generation target is S-function (Simulation) or MEX-function.

Name	Product	Remarks and Limitations
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>Specifying constants</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” on page 21-53.</p>
kaiserord	Signal Processing Toolbox	—
kron	MATLAB	—
kmeans	Statistics and Machine Learning Toolbox	<ul style="list-style-type: none"> • If the <code>Start</code> method uses random selections, the initial centroid cluster positions might not match MATLAB. • If the number of rows in <code>X</code> is fixed, does not remove rows of <code>X</code> that contain a <code>NaN</code>. • The cluster centroid locations in <code>C</code> can have a different order than in MATLAB. In this case, the cluster indices in <code>idx</code> have corresponding differences. • If you provide <code>Display</code>, its value must be <code>'off'</code>. • If you provide <code>Streams</code>, it must be empty and <code>UseSubstreams</code> must be <code>false</code>. • When you set the <code>UseParallel</code> option to <code>true</code>, some computations can execute in parallel even when <code>Replicates</code> is 1. For large data sets, when <code>Replicates</code> is 1, consider setting the <code>UseParallel</code> option to <code>true</code>.

Name	Product	Remarks and Limitations
kurtosis	Statistics and Machine Learning Toolbox	—
label2rgb	Image Processing Toolbox	Referring to the standard syntax: <code>RGB = label2rgb(L, map, zerocolor, order)</code> <ul style="list-style-type: none"> • Submit at least two input arguments: the label matrix, L, and the colormap matrix, map. • map must be an n-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. • If you set the boundary color zerocolor to the same color as one of the regions, label2rgb does not issue a warning. • If you supply a value for order, it must be 'noshuffle'.
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"> • Not supported for fixed-point signals with different biases.
length	MATLAB	—
length	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
levinson	Signal Processing Toolbox	<ul style="list-style-type: none"> • Code generation for this function requires the DSP System Toolbox software. • If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</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” on page 21-53.</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"> • The option structure must be a constant. • Supports only a scalar option structure input. It does not support arrays of option structures. • Only optimizes these cases: <ul style="list-style-type: none"> • UT • LT • UHESS = true (the TRANSA can be either true or false) • SYM = true and POSDEF = true <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"> • Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use <code>coder.load</code>. • 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>. • 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. • Arguments to <code>load</code> must be compile-time constant strings. • Does not support loading objects. • If the MAT-file contains unsupported constructs, use <code>load(filename, variables)</code> to load only the supported constructs. • 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. <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"> 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>.
log2	MATLAB	—
log10	MATLAB	—
log1p	MATLAB	—
logical	MATLAB	—
logical	Fixed-Point Designer	—
logncdf	Statistics and Machine Learning Toolbox	—
logninv	Statistics and Machine Learning Toolbox	—
lognpdf	Statistics and Machine Learning Toolbox	—
lognrnd	Statistics and Machine Learning 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"> The output is nonscalar. An input parameter is invalid for the distribution.
lognstat	Statistics and Machine Learning Toolbox	—
logspace	MATLAB	—

Name	Product	Remarks and Limitations
lower	MATLAB	<ul style="list-style-type: none"> • Supports only char inputs. Does not support cell arrays. • Input values must be in the range 0-127.
lowerbound	Fixed-Point Designer	—
lsb	Fixed-Point Designer	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
lsqnonneg	MATLAB	<ul style="list-style-type: none"> • You must enable support for variable-size arrays. • The <code>message</code> string in the output structure <code>output</code> (the fifth output) is not translated.
lt	MATLAB	—
lteZadoffChuSeq	Communications System Toolbox	—
lt	Fixed-Point Designer	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
lu	MATLAB	—
mad	Statistics and Machine Learning Toolbox	Input <code>dim</code> cannot be empty.
magic	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
matchFeatures	Computer Vision System Toolbox	Generates platform-dependent library: Yes for MATLAB host. The function generates portable C code for non-host target. Compile-time constant input: <code>Method</code> and <code>Metric</code> . Supports MATLAB Function block: Yes Generated code for this function uses a precompiled platform-specific shared library.

Name	Product	Remarks and Limitations
max	MATLAB	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4..
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>Specifying constants</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” on page 21-53.</p>
mdltest	Phased Array System Toolbox	Does not support variable-size inputs.
mean	MATLAB	<ul style="list-style-type: none"> • Does not support the <code>'native'</code> output class option for integer types. • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument.
mean	Fixed-Point Designer	N/A
mean2	Image Processing Toolbox	—

Name	Product	Remarks and Limitations
medfilt2	Image Processing Toolbox	The <code>padopt</code> argument must be a compile-time constant. If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
median	MATLAB	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
median	Fixed-Point Designer	—
meshgrid	MATLAB	—
mfilename	MATLAB	—
min	MATLAB	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
min	Fixed-Point Designer	—
minus	MATLAB	—
minus	Fixed-Point Designer	<ul style="list-style-type: none"> • 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.

Name	Product	Remarks and Limitations
mkpp	MATLAB	<ul style="list-style-type: none"> • 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. <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"> • In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB. • In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure. <ul style="list-style-type: none"> • 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. • To define a piecewise constant polynomial, <code>coefs</code> must be a column vector or <code>d</code> must have at least two elements. • 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. • 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"> 1 Suppose that <code>m = length(d)</code> and <code>npieces = length(breaks) - 1</code>. <pre style="margin-left: 20px;">size(coefs,j) = d(j) size(coefs,m+1) = npieces</pre>

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>2 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"> If you do not provide <code>d</code>, the following statements must be true: 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.
mldivide	MATLAB	—
mnpdf	Statistics and Machine Learning Toolbox	—
mod	MATLAB	<ul style="list-style-type: none"> Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors. <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"> Does not support third output argument <code>C</code> (cell array). If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Name	Product	Remarks and Limitations
moment	Statistics and Machine Learning Toolbox	If <code>order</code> is nonintegral and <code>X</code> is real, use <code>moment(complex(X), order)</code> .
mpower	MATLAB	If <code>A</code> is a 2-by-2 or larger matrix and <code>B</code> is <code>Inf</code> or <code>-Inf</code> , <code>mpower(A,B)</code> returns a matrix of NaNs.
mpower	Fixed-Point Designer	<ul style="list-style-type: none"> • The exponent input, <code>k</code>, must be constant; that is, its value must be known at compile time. • Variable-size inputs are supported only when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. • For variable-size signals, you can see different results between MATLAB and the generated code. <ul style="list-style-type: none"> • In generated code, the output for variable-size signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. • 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>.

Name	Product	Remarks and Limitations
mpy	Fixed-Point Designer	<ul style="list-style-type: none"> 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>. When you provide complex inputs to the <code>mpy</code> function inside a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code>.
mrdivide	MATLAB	—
mrdivide	Fixed-Point Designer	—
MSERRegions	Computer Vision System Toolbox	<p>Compile-time constant input: No restrictions. Supports MATLAB Function block: Yes For code generation, you must specify both the <code>pixellist</code> cell array and the <code>length</code> of each array, as the second input. The object outputs, <code>regions.PixelList</code> as an array. The region sizes are defined in <code>regions.Lengths</code>. Generated code for this function uses a precompiled platform-specific shared library.</p>
mtimes	MATLAB	<ul style="list-style-type: none"> 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, $(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Inf}i$. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Name	Product	Remarks and Limitations
mtimes	Fixed-Point Designer	<ul style="list-style-type: none"> • 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. • 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>. • For variable-sized signals, you may see different results between MATLAB and the generated code. <ul style="list-style-type: none"> • In generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. • 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>.
multithresh	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
mvdweights	Phased Array System Toolbox	Does not support variable-size inputs.
NaN or nan	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
nancov	Statistics and Machine Learning Toolbox	If the input is variable-size and is <code>[]</code> at run time, returns <code>[]</code> not NaN.
nanmax	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
nanmean	Statistics and Machine Learning Toolbox	—
nanmedian	Statistics and Machine Learning Toolbox	—
nanmin	Statistics and Machine Learning Toolbox	—
nanstd	Statistics and Machine Learning Toolbox	—
nansum	Statistics and Machine Learning Toolbox	—
nanvar	Statistics and Machine Learning Toolbox	—
nargchk	MATLAB	<ul style="list-style-type: none"> • Output structure does not include stack information. <hr/> <p>Note: nargchk will be removed in a future release.</p>
nargin	MATLAB	—
narginchk	MATLAB	—
nargout	MATLAB	<ul style="list-style-type: none"> • For a function with no output arguments, returns 1 if called without a terminating semicolon. <hr/> <p>Note: This behavior also affects extrinsic calls with no terminating semicolon. nargout is 1 for the called function in MATLAB.</p>

Name	Product	Remarks and Limitations
nargoutchk	MATLAB	—
nbincdf	Statistics and Machine Learning Toolbox	—
nbiniinv	Statistics and Machine Learning Toolbox	—
nbinpdf	Statistics and Machine Learning Toolbox	—
nbirrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
nbinstat	Statistics and Machine Learning Toolbox	—
ncfcdf	Statistics and Machine Learning Toolbox	—
ncfinv	Statistics and Machine Learning Toolbox	—
ncfpdf	Statistics and Machine Learning Toolbox	—
ncfrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.

Name	Product	Remarks and Limitations
ncfstat	Statistics and Machine Learning Toolbox	—
nchoosek	MATLAB	<ul style="list-style-type: none"> • When the first input, <code>x</code>, is a scalar, <code>nchoosek</code> returns a binomial coefficient. In this case, <code>x</code> must be a nonnegative integer. It cannot have type <code>int64</code> or <code>uint64</code>. • When the first input, <code>x</code>, is a vector, <code>nchoosek</code> treats it as a set. In this case, <code>x</code> can have type <code>int64</code> or <code>uint64</code>. • The second input, <code>k</code>, cannot have type <code>int64</code> or <code>uint64</code>. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
nctcdf	Statistics and Machine Learning Toolbox	—
nctinv	Statistics and Machine Learning Toolbox	—
nctpdf	Statistics and Machine Learning Toolbox	—
nctrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
nctstat	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
ncx2cdf	Statistics and Machine Learning Toolbox	—
ncx2rnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
ncx2stat	Statistics and Machine Learning Toolbox	—
ndgrid	MATLAB	—
ndims	MATLAB	—
ndims	Fixed-Point Designer	—
ne	MATLAB	—
ne	Fixed-Point Designer	• Not supported for fixed-point signals with different biases.
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 and Machine Learning Toolbox	—
normest	MATLAB	—

Name	Product	Remarks and Limitations
<code>norminv</code>	Statistics and Machine Learning Toolbox	—
<code>normpdf</code>	Statistics and Machine Learning Toolbox	—
<code>normrnd</code>	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
<code>normstat</code>	Statistics and Machine Learning Toolbox	—
<code>not</code>	MATLAB	—
<code>npwgntresh</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"> • Might return a different basis than MATLAB • Does not support rational basis option (second input)
<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	—

Name	Product	Remarks and Limitations
numerictype	Fixed-Point Designer	<ul style="list-style-type: none"> • 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. • Returns the data type when the input is a nonfixed-point signal. • Use to create <code>numerictype</code> objects in the generated code. • All <code>numerictype</code> object properties related to the data type must be constant.
nutallwin	Signal Processing Toolbox	<p>Inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>
ocr	Computer Vision System Toolbox	<p>Compile-time constant input: <code>TextLayout</code>, <code>Language</code>, and <code>CharacterSet</code>.</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>

Name	Product	Remarks and Limitations
ode23	MATLAB	<ul style="list-style-type: none"> • All <code>odeset</code> option arguments must be constant. • Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function . • You must provide at least the two output arguments <code>T</code> and <code>Y</code>. • Input types must be homogeneous—all double or all single. • Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.
ode45	MATLAB	<ul style="list-style-type: none"> • All <code>odeset</code> option arguments must be constant. • Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function . • You must provide at least the two output arguments <code>T</code> and <code>Y</code>. • Input types must be homogeneous—all double or all single. • Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.
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"> • Dimensions must be real, nonnegative integers. • The input <code>optimfun</code> must be a function supported for code generation.

Name	Product	Remarks and Limitations
opticalFlow	Computer Vision System Toolbox	Supports MATLAB Function block: Yes
opticalFlowFarneback	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlowHS	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlowLK	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlowLKDoG	Computer Vision System Toolbox	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
optimget	MATLAB	Input parameter names must be constant.
optimset	MATLAB	<ul style="list-style-type: none"> • Does not support the syntax that has no input or output arguments: <code>optimset</code> • Functions specified in the options must be supported for code generation. • The fields of the options structure <code>oldopts</code> must be fixed-size fields. • For code generation, optimization functions ignore the <code>Display</code> option. • 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.

Name	Product	Remarks and Limitations
ordfilt2	Image Processing Toolbox	The <code>padopt</code> argument must be a compile-time constant. If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
or	MATLAB	—
orth	MATLAB	<ul style="list-style-type: none"> • Can return a different basis than MATLAB
padarray	Image Processing Toolbox	<ul style="list-style-type: none"> • Support only up to 3-D inputs. • Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.
parfor	MATLAB	<ul style="list-style-type: none"> • Treated as a <code>for</code>-loop in a MATLAB Function block or when used with <code>fiaccl</code>. • See the <code>parfor</code> reference page in the MATLAB Coder documentation. • “Generate Code with Parallel for-Loops (<code>parfor</code>)” on page 29-32.
parzenwin	Signal Processing Toolbox	Inputs must be constant. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
pascal	MATLAB	—

Name	Product	Remarks and Limitations
pca	Statistics and Machine Learning Toolbox	<ul style="list-style-type: none"> • Ignores the 'Display' value for 'Options' when 'Algorithm' is 'als'. • If supplied, 'Weights' and 'VariableWeights' must be real. • Always returns the fifth output explained as a column vector. • Always returns the sixth output mu as a row vector. • If mu is empty, pca returns mu as a 1-by-0 array. pca does not convert mu to a 0-by-0 empty array. • Does not treat an input matrix X that has all NaN values as a special case. The outputs have the sizes that they have when some of the inputs are finite.
pchip	MATLAB	<ul style="list-style-type: none"> • Input x must be strictly increasing. • Does not remove y entries with NaN values. • If you generate code for the pp = pchip(x,y) syntax, you cannot input pp to the ppval function in MATLAB. To create a MATLAB pp structure from a pp structure created by the code generation software: <ul style="list-style-type: none"> • In code generation, use unmkpp to return the piecewise polynomial details to MATLAB. • In MATLAB, use mkpp to create the pp structure.
pdf	Statistics and Machine Learning Toolbox	—
pearsrnd	Statistics and Machine Learning Toolbox	Matches MATLAB only when generated output r is scalar.

Name	Product	Remarks and Limitations
permute	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays for the first argument. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
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	“System Objects in MATLAB Code Generation”
phased.AngleDoppler-Response	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.ArrayGain	Phased Array System Toolbox	<ul style="list-style-type: none"> Does not support arrays containing polarized antenna elements, that is, the <code>phased.ShortDipoleAntennaElement</code> or <code>phased.CrossedDipoleAntennaElement</code> antennas. “System Objects in MATLAB Code Generation”
phased.ArrayResponse	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.BarrageJammer	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.BeamscanEstimator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Beamscan-Estimator2D	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Beamspace-ESPRITEstimator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.CFARDetector	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Collector	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
phased.ConformalArray	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.ConstantGammaClutter	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.CosineAntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.CrossedDipoleAntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.CustomAntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.CustomMicrophoneElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.DPCACanceller	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.ElementDelay	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.ESPRITEstimator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
phased.FMCWWaveform	Phased Array System Toolbox	<ul style="list-style-type: none"> • plot method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.FreeSpace	Phased Array System Toolbox	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
phased.FrostBeamformer	Phased Array System Toolbox	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
phased.GCCEstimator	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phased.Isotropic-AntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • pattern, patternAzimuth, patternElevation, and plotResponse methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.LCMVBeamformer	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phased.LinearFMWaveform	Phased Array System Toolbox	<ul style="list-style-type: none"> • plot method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.MatchedFilter	Phased Array System Toolbox	<ul style="list-style-type: none"> • The CustomSpectrumWindow property is not supported. • “System Objects in MATLAB Code Generation”
phased.MFSKWaveform	Phased Array System Toolbox	<ul style="list-style-type: none"> • plot method is not supported. • See “System Objects in MATLAB Code Generation”.

Name	Product	Remarks and Limitations
phased.MVDRBeamformer	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.MVDREstimator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.MVDREstimator2D	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Omnidirectional-MicrophoneElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.PartitionedArray	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.PhaseCoded-Waveform	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.PhaseShift-Beamformer	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Platform	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.RadarTarget	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Radiator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Range-DopplerResponse	Phased Array System Toolbox	<ul style="list-style-type: none"> • The <code>CustomRangeWindow</code> and the <code>CustomDopplerWindow</code> properties are not supported. • “System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
phased.Rectangular-Waveform	Phased Array System Toolbox	<ul style="list-style-type: none"> • plot method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.ReceiverPreamp	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.ReplicatedSubarray	Phased Array System Toolbox	<ul style="list-style-type: none"> • pattern, patternAzimuth, patternElevation, plotResponse, and viewArray methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.RootMUSICEstimator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.RootWSFEstimator	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.ShortDipole-AntennaElement	Phased Array System Toolbox	<ul style="list-style-type: none"> • pattern, patternAzimuth, patternElevation, and plotResponse methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.STAPSMI-Beamformer	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.SteeringVector	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.SteppedFMWaveform	Phased Array System Toolbox	<ul style="list-style-type: none"> • plot method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.StretchProcessor	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Subband-MVDRBeamformer	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phased.SubbandPhase-ShiftBeamformer	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
phased.SumDifference-MonopulseTracker	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phased.SumDifference-MonopulseTracker2D	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phased.TimeDelay-Beamformer	Phased Array System Toolbox	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • “System Objects in MATLAB Code Generation”
phased.TimeDelayLCMV-Beamformer	Phased Array System Toolbox	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • “System Objects in MATLAB Code Generation”
phased.TimeVaryingGain	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.Transmitter	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.TwoRayChannel	Phased Array System Toolbox	“System Objects in MATLAB Code Generation”
phased.UCA	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.ULA	Phased Array System Toolbox	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.

Name	Product	Remarks and Limitations
phased.URA	Phased Array System Toolbox	<ul style="list-style-type: none"> • pattern, patternAzimuth, patternElevation, plotResponse, and viewArray methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.WidebandCollector	Phased Array System Toolbox	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • “System Objects in MATLAB Code Generation”.
phased.WidebandFreeSpace	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phased.WidebandRadiator	Phased Array System Toolbox	See “System Objects in MATLAB Code Generation”.
phitheta2azel	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2azelpat	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2uv	Phased Array System Toolbox	Does not support variable-size inputs.
phitheta2uvpat	Phased Array System Toolbox	Does not support variable-size inputs.
physconst	Phased Array System Toolbox	Does not support variable-size inputs.
pi	MATLAB	—
pilotcalib	Phased Array System Toolbox	Does not support variable-size inputs.
pinv	MATLAB	—
planerot	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
plus	MATLAB	—

Name	Product	Remarks and Limitations
plus	Fixed-Point Designer	<ul style="list-style-type: none"> 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.
poisscdf	Statistics and Machine Learning Toolbox	—
poissinv	Statistics and Machine Learning Toolbox	—
poisspdf	Statistics and Machine Learning Toolbox	—
poissrnd	Statistics and Machine Learning 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"> The output is nonscalar. An input parameter is invalid for the distribution.
poisstat	Statistics and Machine Learning 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"> Does not support variable-size inputs. Supported only when output arguments are specified.

Name	Product	Remarks and Limitations
poly	MATLAB	<ul style="list-style-type: none"> • Does not discard nonfinite input values • Complex input produces complex output • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
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.
polyfit	MATLAB	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
polyint	MATLAB	—
polyval	MATLAB	—
polyvalm	MATLAB	—
pow2	Fixed-Point Designer	—
pow2db	Signal Processing Toolbox	—
power	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation. When both X and Y are real, but $\text{power}(X, Y)$ is complex, returns NaN in the generated code. To get the complex result, make the input value X complex by passing in $\text{complex}(X)$. For example, $\text{power}(\text{complex}(X), Y)$. • Generates an error during simulation. When both X and Y are real, but $X.^Y$ is complex, returns NaN in generated code. To get the complex result, make the input value X complex by using $\text{complex}(X)$. For example, $\text{complex}(X).^Y$.

Name	Product	Remarks and Limitations
power	Fixed-Point Designer	<ul style="list-style-type: none"> The exponent input, <code>k</code>, must be constant. Its value must be known at compile time.
ppval	MATLAB	<p>The size of output <code>v</code> does not match MATLAB when both of the following statements are true:</p> <ul style="list-style-type: none"> The input <code>x</code> is a variable-size array that is not a variable-length vector. <code>x</code> becomes a row vector at run time. <p>The code generation software does not remove the singleton dimensions. However, MATLAB might remove singleton dimensions.</p> <p>For example, suppose that <code>x</code> 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 2-by-3 fixed-size array. <code>v</code> 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 <code>[2,3,1,5]</code>. In MATLAB, the size is <code>[2,3,5]</code>.</p>

Name	Product	Remarks and Limitations
prctile	Statistics and Machine Learning Toolbox	<ul style="list-style-type: none"> • “Automatic dimension restriction” on page 7-35 • 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"> • You do not supply the <code>dim</code> input. • X is a variable-size array. • X is not a variable-length vector. • X is a vector at run time. • The orientation of the vector X does not match the orientation of the vector p. <p>In this case, the output Y matches the orientation of X not the orientation of p.</p>
primes	MATLAB	<ul style="list-style-type: none"> • The maximum double precision input is 2^{32}. • The maximum single precision input is 2^{24}. • The input n cannot have type <code>int64</code> or <code>uint64</code>.
prod	MATLAB	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
projective2d	Image Processing Toolbox	When generating code, you can only specify single objects—arrays of objects are not supported.
psi	MATLAB	—
psnr	Image Processing Toolbox	—
pulsint	Phased Array System Toolbox	Does not support variable-size inputs.
qr	MATLAB	—

Name	Product	Remarks and Limitations
quad2d	MATLAB	<ul style="list-style-type: none"> 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.
quadgk	MATLAB	—
quantile	Statistics and Machine Learning Toolbox	—
quantize	Fixed-Point Designer	—
quat2axang	Robotics System Toolbox	Supports MATLAB Function block: No
quat2eul	Robotics System Toolbox	Supports MATLAB Function block: No
quat2rotm	Robotics System Toolbox	Supports MATLAB Function block: No
quat2tform	Robotics System Toolbox	Supports MATLAB Function block: No
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.
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.

Name	Product	Remarks and Limitations
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"> • <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>. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
randg	Statistics and Machine Learning Toolbox	—
randi	MATLAB	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Name	Product	Remarks and Limitations
randn	MATLAB	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
random	Statistics and Machine Learning Toolbox	—
randperm	MATLAB	—
randsample	Statistics and Machine Learning Toolbox	When sampling without replacement, the order of the output values might not match 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 and Machine Learning Toolbox	—
raylinv	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
raylpdf	Statistics and Machine Learning Toolbox	—
raylrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
raylstat	Statistics and Machine Learning Toolbox	—
rcond	MATLAB	—
rcosdesign	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</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” on page 21-53.
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	—

Name	Product	Remarks and Limitations
realmax	Fixed-Point Designer	—
realmin	MATLAB	—
realmin	Fixed-Point Designer	—
realpow	MATLAB	—
realsqrt	MATLAB	—
reconstructScene	Computer Vision System Toolbox	Supports MATLAB Function block: No
rectifyStereoImages	Computer Vision System Toolbox	Compile-time constant input restriction: 'interp' and 'OutputView' Supports MATLAB Function block: No
rectint	MATLAB	—
recursiveAR	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveARMA	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveARMAX	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Name	Product	Remarks and Limitations
recursiveARX	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveBJ	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveLS	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Least Squares Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveOE	System Identification Toolbox	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
rectwin	Signal Processing Toolbox	<p>All inputs must be constant. Expressions or variables are allowed if their values do not change.</p> <p>Specifying constants</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” on page 21-53.</p>

Name	Product	Remarks and Limitations
regionprops	Image Processing Toolbox	<p>Supports only 2-D images. Does not accept the connected component structure (CC) returned by <code>bwconncomp</code>. Use <code>bwlabel</code> to create a label matrix, or pass the image to <code>regionprops</code> directly. Does not support the table output type. Does not accept cell arrays as input—use a comma-separated list instead. Does not support the properties <code>ConvexArea</code>, <code>ConvexHull</code>, <code>ConvexImage</code>, <code>Solidity</code>, and <code>SubarrayIdx</code>.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
reinterpretpcast	Fixed-Point Designer	—
rem	MATLAB	<ul style="list-style-type: none"> • Performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors. • If one of the inputs has type <code>int64</code> or <code>uint64</code>, then both inputs must have the same type.
removefimath	Fixed-Point Designer	—
repmat	MATLAB	Size arguments must have a fixed size.
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>Specifying constants</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” on page 21-53.</p>

Name	Product	Remarks and Limitations
rescale	Fixed-Point Designer	—
reshape	MATLAB	<ul style="list-style-type: none"> • If the input is a compile-time empty cell array, then the size arguments must be constants. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
reshape	Fixed-Point Designer	—
return	MATLAB	—
rgb2gray	MATLAB	—
rgb2ycbcr	Image Processing Toolbox	—
rng	MATLAB	<ul style="list-style-type: none"> • For library code generation targets, executable code generation targets, and MEX targets with extrinsic calls disabled: <ul style="list-style-type: none"> • Does not support the 'shuffle' input. • For the generator input, supports 'twister', 'v4', and 'v5normal'. <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"> • For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by <code>rng</code>.
robotics.PurePursuit	Robotics System Toolbox	Supports MATLAB Function block: No

Name	Product	Remarks and Limitations
robotics.VectorFieldHistogram	Robotics System Toolbox	Supports MATLAB Function block: No
rocdfa	Phased Array System Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. The <code>NonfluctuatingNoncoherent</code> signal type is not supported.
rocsnr	Phased Array System Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Does not support the <code>NonfluctuatingNoncoherent</code> signal type.
rootmusicdoa	Phased Array System Toolbox	Does not support variable-size inputs.
roots	MATLAB	<ul style="list-style-type: none"> Output is variable size. Output is complex. Roots are not always in the same order as MATLAB. Roots of poorly conditioned polynomials do not always match MATLAB. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
rosser	MATLAB	—
rot90	MATLAB	Does not support cell arrays for the first argument.
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.
rotm2axang	Robotics System Toolbox	Supports MATLAB Function block: No
rotm2eul	Robotics System Toolbox	Supports MATLAB Function block: No
rotm2quat	Robotics System Toolbox	Supports MATLAB Function block: No
rotm2tform	Robotics System Toolbox	Supports MATLAB Function block: No

Name	Product	Remarks and Limitations
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	Supports only the syntax $Y = \text{round}(X)$.
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"> • In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>. • In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.
sech	MATLAB	—
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"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' and 'rows' options, the output <code>ia</code> is a column vector. If <code>ia</code> is empty, it is 0-by-1, never 0-by-0, even if the output <code>C</code> is 0-by-0. • 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. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • Sort the real input in complex ascending order (by absolute value). Suppose the real input

Name	Product	Remarks and Limitations
		<p>is x. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</p> <ul style="list-style-type: none"> • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
setfimath	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
setxor	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • 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. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • 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, C, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • Sort the real input in complex ascending order (by absolute

Name	Product	Remarks and Limitations
		value). Suppose the real input is x . Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code> . <ul style="list-style-type: none"> • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
sfi	Fixed-Point Designer	<ul style="list-style-type: none"> • All properties related to data type must be constant for code generation.
sgolay	Signal Processing Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change. <p>Specifying constants</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” on page 21-53.
shiftdim	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • Second argument must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
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	—

Name	Product	Remarks and Limitations
<code>sind</code>	MATLAB	—
<code>single</code>	MATLAB	—
<code>single</code>	Fixed-Point Designer	—
<code>sinh</code>	MATLAB	—
<code>size</code>	MATLAB	—
<code>size</code>	Fixed-Point Designer	—
<code>skewness</code>	Statistics and Machine Learning Toolbox	—
<code>sort</code>	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • 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 with Zero-Valued Imaginary Parts” on page 6-4.
<code>sort</code>	Fixed-Point Designer	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
<code>sortrows</code>	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • 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 with Zero-Valued Imaginary Parts” on page 6-4.

Name	Product	Remarks and Limitations
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.
spline	MATLAB	<ul style="list-style-type: none"> • Input x must be strictly increasing. • Does not remove Y entries with NaN values. • Does not report an error for infinite endslopes in Y. • 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"> • In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB. • In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.
spsmooth	Phased Array System Toolbox	Does not support variable-size inputs.
squeeze	MATLAB	Does not support cell arrays.
squeeze	Fixed-Point Designer	—
sqrt	MATLAB	<ul style="list-style-type: none"> • 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>.

Name	Product	Remarks and Limitations
sqrt	Fixed-Point Designer	<ul style="list-style-type: none"> • Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sqrtm	MATLAB	—
std	MATLAB	<ul style="list-style-type: none"> • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the nanflag argument.
steervec	Phased Array System Toolbox	Does not support variable-size inputs.
stereoParameters	Computer Vision System Toolbox	Supports MATLAB Function block: No
stokes	Phased Array System Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Supported only when output arguments are specified.
storedInteger	Fixed-Point Designer	—
storedIntegerToDouble	Fixed-Point Designer	—
str2double	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays. • Always returns a complex result.
str2func	MATLAB	String must be constant/known at compile time.
strcmp	MATLAB	<ul style="list-style-type: none"> • When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector. • When both inputs are empty character arrays that have different sizes, returns true.

Name	Product	Remarks and Limitations
strcmpi	MATLAB	<ul style="list-style-type: none"> • Input values from the <code>char</code> class must be in the range 0-127. • When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector. • When both inputs are empty character arrays that have different sizes, returns true.
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	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
strfind	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays. • 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>. • Inputs must be character row vectors.
strjoin	MATLAB	Always returns a string that has size 1-by-n. For empty inputs, returns a 1-by-0 string not <code>[]</code> .
strjust	MATLAB	Does not support a cell array of strings for the first argument.
strncmp	MATLAB	When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.

Name	Product	Remarks and Limitations
strncmpi	MATLAB	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0-127. When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.
strrep	MATLAB	<ul style="list-style-type: none"> Does not support cell arrays. 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>. Inputs must be character row vectors.
strtok	MATLAB	Does not support cell arrays for the first argument.
strtrim	MATLAB	<ul style="list-style-type: none"> Supports only inputs from the <code>char</code> class. Does not support cell arrays. Input values must be in the range 0-127.
struct	MATLAB	<ul style="list-style-type: none"> You cannot create a structure that contains a cell array. For example, you cannot generate code for: <code>s = struct('a', {{1 2}})</code> If the <code>value</code> argument is a cell array, all elements must have the same type.
struct2cell	MATLAB	<ul style="list-style-type: none"> For a variable-size structure array, the resulting cell array must be homogeneous. If <code>s</code> is a variable-size structure array, the fields must have the same type. If <code>struct2cell</code> cannot convert <code>s</code> to a homogeneous cell array, the output cell array is heterogeneous. A heterogeneous output cell array can have a maximum of 1024 elements.
structfun	MATLAB	<ul style="list-style-type: none"> Does not support the <code>ErrorHandler</code> option. The number of outputs must be less than or equal to three.

Name	Product	Remarks and Limitations
sub	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> .
sub2ind	MATLAB	<ul style="list-style-type: none"> • The first argument must be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
subsasgn	Fixed-Point Designer	—
subspace	MATLAB	—
subsref	Fixed-Point Designer	—
sum	MATLAB	<ul style="list-style-type: none"> • Specify <code>dim</code> as a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument.
sum	Fixed-Point Designer	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision or Keep LSB</code>.
surfacegamma	Phased Array System Toolbox	Does not support variable-size inputs.
surfcluttercs	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
SURFPoints	Computer Vision System Toolbox	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>visionRecoverformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
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-double inputs, the input port data types must be specified, not inherited.
switch, case, otherwise	MATLAB	<ul style="list-style-type: none"> • 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. • 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.
systemp	Phased Array System Toolbox	Does not support variable-size inputs.
tan	MATLAB	—
tand	MATLAB	<ul style="list-style-type: none"> • In some cases, returns <code>-Inf</code> when MATLAB returns <code>Inf</code>. • In some cases, returns <code>Inf</code> when MATLAB returns <code>-Inf</code>.

Name	Product	Remarks and Limitations
tanh	MATLAB	—
taylorwin	Signal Processing Toolbox	Inputs must be constant Specifying constants 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” on page 21-53.
tcdf	Statistics and Machine Learning Toolbox	—
tf2ca	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tf2c1	DSP System Toolbox	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tform2axang	Robotics System Toolbox	Supports MATLAB Function block: No
tform2eu1	Robotics System Toolbox	Supports MATLAB Function block: No
tform2quat	Robotics System Toolbox	Supports MATLAB Function block: No
tform2rotm	Robotics System Toolbox	Supports MATLAB Function block: No
tform2trvec	Robotics System Toolbox	Supports MATLAB Function block: No
time2range	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
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, $(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Infi}$.
times	Fixed-Point Designer	<ul style="list-style-type: none"> 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. 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 Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code>.
tinvs	Statistics and Machine Learning Toolbox	—
toeplitz	MATLAB	—
tpdf	Statistics and Machine Learning Toolbox	—
trace	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Designer	—
trapz	MATLAB	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Name	Product	Remarks and Limitations
triang	Signal Processing Toolbox	Inputs must be constant Specifying constants 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” on page 21-53.
triangulate	Computer Vision System Toolbox	Supports MATLAB Function block: No
tril	MATLAB	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
tril	Fixed-Point Designer	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
triu	MATLAB	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
triu	Fixed-Point Designer	<ul style="list-style-type: none"> If supplied, the index, k, must be a real and scalar integer value that is not a <code>fi</code> object.
trnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> The output is nonscalar. An input parameter is invalid for the distribution.
true	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
trvec2tform	Robotics System Toolbox	Supports MATLAB Function block: No
tstat	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
tukeywin	Signal Processing Toolbox	<p>Inputs must be constant.</p> <p>Specifying constants</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” on page 21-53.</p>
typecast	MATLAB	<ul style="list-style-type: none"> • Value of string input argument <code>type</code> must be lowercase. • 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 input port data types explicitly. • Integer input or result classes must map directly to a C type on the target hardware. • The input must be a variable-length vector or a fixed-size vector. See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • The output vector always has the same orientation as the input vector.
ufi	Fixed-Point Designer	<ul style="list-style-type: none"> • All properties related to data type must be constant for code generation.
uint8, uint16, uint32, uint64	MATLAB	No integer overflow detection for <code>int64</code> in MEX or MATLAB Function block simulation on Windows 32-bit platforms.
uint8, uint16, uint32, uint64	Fixed-Point Designer	—
uminus	MATLAB	—
uminus	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
undistortImage	Computer Vision System Toolbox	Compile-time constant input restriction: 'interp' and 'OutputView' Supports MATLAB Function block: No
unidcdf	Statistics and Machine Learning Toolbox	—
unidinv	Statistics and Machine Learning Toolbox	—
unidpdf	Statistics and Machine Learning Toolbox	—
unidrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
unidstat	Statistics and Machine Learning Toolbox	—
unifcdf	Statistics and Machine Learning Toolbox	—
unifinv	Statistics and Machine Learning Toolbox	—
unifpdf	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
unifrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none">• The output is nonscalar.• An input parameter is invalid for the distribution.
unifstat	Statistics and Machine Learning Toolbox	—
unigrd	Phased Array System Toolbox	Does not support variable-size inputs.

Name	Product	Remarks and Limitations
union	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • 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. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • 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, C, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • Sort the real input in complex ascending order (by absolute

Name	Product	Remarks and Limitations
		<p>value). Suppose the real input is <code>x</code>. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>.</p> <ul style="list-style-type: none"> • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
unique	MATLAB	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • The input <code>A</code> must be a vector. If you specify the 'legacy' option, the input <code>A</code> must be a row vector. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'rows' option and the 'legacy' option, outputs <code>ia</code> and <code>ic</code> are column vectors. If these outputs are empty, they are 0-by-1, even if the output <code>C</code> is 0-by-0. • When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the input <code>A</code> must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>.

Name	Product	Remarks and Limitations
unmkpp	MATLAB	<ul style="list-style-type: none"> • pp 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. • Does not support pp structures created by <code>interp1</code> in MATLAB.
unwrap	MATLAB	<ul style="list-style-type: none"> • Row vector input is only supported when the first two inputs are vectors and nonscalar • Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors
upfirdn	Signal Processing Toolbox	<ul style="list-style-type: none"> • Code generation for this function requires the 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. <p>Specifying constants</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” on page 21-53.</p>
uplus	MATLAB	—
uplus	Fixed-Point Designer	—
upper	MATLAB	<ul style="list-style-type: none"> • Supports only inputs from the <code>char</code> class. Does not support cell arrays. • Input values must be in the range 0-127.
upperbound	Fixed-Point Designer	—

Name	Product	Remarks and Limitations
upsample	Signal Processing Toolbox	Either declare input n as constant, or use the <code>assert</code> function in the calling function to set upper bounds for n . For example, <code>assert (n<10)</code>
uv2azel	Phased Array System Toolbox	Does not support variable-size inputs.
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"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument.
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”

Name	Product	Remarks and Limitations
vision.CascadeObjectDetector	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.ColorSpaceConverter	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.DemosaicInterpolator	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.DeployableVideoPlayer	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”
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”

Name	Product	Remarks and Limitations
vision.ForegroundDetector	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.GammaCorrector	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricRotator	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricScaler	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricShearer	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricTransformer	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.GeometricTranslator	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Histogram	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Histogram-BasedTracker	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.HistogramEqualizer	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.HoughLines	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.HoughTransform	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.IDCT	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.IFFT	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ImageComplementer	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
vision.ImageFilter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.ImageDataType-Converter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
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.LocalMaximaFinder	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.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: No “System Objects in MATLAB Code Generation”
vision.PSNR	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Pyramid	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”

Name	Product	Remarks and Limitations
vision.ShapeInserter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.StandardDeviation	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.TemplateMatcher	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.TextInserter	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.Variance	Computer Vision System Toolbox	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
vision.VideoFileReader	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”
vision.VideoFileWriter	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”
watershed	Image Processing Toolbox	Supports only 2-D images. Supports only 4- or 8-connectivity. Supports only up to 65,535 regions. The output type is always <code>uint16</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
wblcdf	Statistics and Machine Learning Toolbox	—
wblinv	Statistics and Machine Learning Toolbox	—
wblpdf	Statistics and Machine Learning Toolbox	—

Name	Product	Remarks and Limitations
wblrnd	Statistics and Machine Learning Toolbox	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
wblstat	Statistics and Machine Learning Toolbox	—
while	MATLAB	—
wilkinson	MATLAB	n must be a fixed-size scalar.
xcorr	Signal Processing Toolbox	—
xor	MATLAB	—
ycbcr2rgb	Image Processing Toolbox	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
yulewalk	Signal Processing Toolbox	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change. Specifying constants 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” on page 21-53.
zeros	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
zp2tf	MATLAB	—
zscore	Statistics and Machine Learning 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” on page 4-2.

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 4-158
“Arithmetic Operations in MATLAB” on page 4-158
“Bit-Wise Operations MATLAB” on page 4-159
“Casting in MATLAB” on page 4-160
“Communications System Toolbox” on page 4-160
“Complex Numbers in MATLAB” on page 4-166
“Computer Vision System Toolbox” on page 4-167
“Control Flow in MATLAB” on page 4-176
“Data and File Management in MATLAB” on page 4-177
“Data Types in MATLAB” on page 4-181
“Desktop Environment in MATLAB” on page 4-182
“Discrete Math in MATLAB” on page 4-183
“DSP System Toolbox” on page 4-183
“Error Handling in MATLAB” on page 4-191
“Exponents in MATLAB” on page 4-191
“Filtering and Convolution in MATLAB” on page 4-192

In this section...

- “Fixed-Point Designer” on page 4-193
- “HDL Coder” on page 4-203
- “Histograms in MATLAB” on page 4-203
- “Image Acquisition Toolbox” on page 4-204
- “Image Processing in MATLAB” on page 4-204
- “Image Processing Toolbox” on page 4-204
- “Input and Output Arguments in MATLAB” on page 4-214
- “Interpolation and Computational Geometry in MATLAB” on page 4-214
- “Linear Algebra in MATLAB” on page 4-218
- “Logical and Bit-Wise Operations in MATLAB” on page 4-219
- “MATLAB Compiler” on page 4-220
- “Matrices and Arrays in MATLAB” on page 4-220
- “Neural Network Toolbox” on page 4-229
- “Numerical Integration and Differentiation in MATLAB” on page 4-229
- “Optimization Functions in MATLAB” on page 4-230
- “Phased Array System Toolbox” on page 4-231
- “Polynomials in MATLAB” on page 4-241
- “Programming Utilities in MATLAB” on page 4-242
- “Relational Operators in MATLAB” on page 4-242
- “Robotics System Toolbox” on page 4-242
- “Rounding and Remainder Functions in MATLAB” on page 4-243
- “Set Operations in MATLAB” on page 4-244
- “Signal Processing in MATLAB” on page 4-249
- “Signal Processing Toolbox” on page 4-250
- “Special Values in MATLAB” on page 4-255
- “Specialized Math in MATLAB” on page 4-256
- “Statistics in MATLAB” on page 4-256
- “Statistics and Machine Learning Toolbox” on page 4-257
- “String Functions in MATLAB” on page 4-267

In this section...

“System Identification Toolbox” on page 4-269

“Trigonometry in MATLAB” on page 4-270

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	<ul style="list-style-type: none">For efficient generated code, MATLAB rules for divide by zero are supported only for the 'round' option.
isa	—
ldivide	—
minus	—
mldivide	—

Function	Remarks and Limitations
<code>mpower</code>	If <code>A</code> is a 2-by-2 or larger matrix and <code>B</code> is <code>Inf</code> or <code>-Inf</code> , <code>mpower(A,B)</code> returns a matrix of NaNs.
<code>mrdivide</code>	—
<code>mtimes</code>	<ul style="list-style-type: none"> 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, $(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Infi}$. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
<code>plus</code>	—
<code>power</code>	<ul style="list-style-type: none"> Generates an error during simulation. When both <code>X</code> and <code>Y</code> 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 <code>X</code> complex by passing in <code>complex(X)</code>. For example, <code>power(complex(X),Y)</code>. Generates an error during simulation. When both <code>X</code> and <code>Y</code> are real, but <code>X.^Y</code> is complex, returns NaN in generated code. To get the complex result, make the input value <code>X</code> complex by using <code>complex(X)</code>. For example, <code>complex(X).^Y</code>.
<code>rdivide</code>	—
<code>times</code>	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, $(\text{Inf} + 1i) * 1i = (\text{Inf} * 0 - 1 * 1) + (\text{Inf} * 1 + 1 * 0)i = \text{NaN} + \text{Infi}$.
<code>transpose</code>	—
<code>uminus</code>	—
<code>uplus</code>	—

Bit-Wise Operations MATLAB

Function	Remarks and Limitations
<code>flintmax</code>	—

Function	Remarks and Limitations
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	Does not support cell arrays.
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"> • Value of string input argument <code>type</code> must be lowercase. • 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 input port data types explicitly. • Integer input or result classes must map directly to a C type on the target hardware. • The input must be a variable-length vector or a fixed-size vector. See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • The output vector always has the same orientation as the input vector.
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
Input and Output	
comm.BarkerCode	“System Objects in MATLAB Code Generation”
comm.GoldSequence	“System Objects in MATLAB Code Generation”
comm.HadamardCode	“System Objects in MATLAB Code Generation”
comm.KasamiSequence	“System Objects in MATLAB Code Generation”
comm.WalshCode	“System Objects in MATLAB Code Generation”
comm.PNSequence	“System Objects in MATLAB Code Generation”
lteZadoffChuSeq	—
Signal and Delay Management	
bi2de	—
de2bi	—
Display and Visual Analysis	
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”
Source Coding	
comm.DifferentialDecoder	“System Objects in MATLAB Code Generation”
comm.DifferentialEncoder	“System Objects in MATLAB Code Generation”
Cyclic Redundancy Check Coding	
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”
BCH Codes	
comm.BCHDecoder	“System Objects in MATLAB Code Generation”
comm.BCHEncoder	“System Objects in MATLAB Code Generation”
Reed-Solomon Codes	

Name	Remarks and Limitations
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”
LDPC Codes	
comm.LDPCDecoder	“System Objects in MATLAB Code Generation”
comm.LDPCEncoder	“System Objects in MATLAB Code Generation”
Convolutional Coding	
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	—
Signal Operations	
comm.Descrambler	“System Objects in MATLAB Code Generation”
comm.Scrambler	“System Objects in MATLAB Code Generation”
Interleaving	
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”

Name	Remarks and Limitations
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”
Frequency Modulation	
comm.FSKDemodulator	“System Objects in MATLAB Code Generation”
comm.FSKModulator	“System Objects in MATLAB Code Generation”
Phase Modulation	
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”
Amplitude Modulation	
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”

Name	Remarks and Limitations
comm.RectangularQAMDemodulator	“System Objects in MATLAB Code Generation”
comm.RectangularQAMModulator	“System Objects in MATLAB Code Generation”
Continuous Phase Modulation	
comm.CPFSKDemodulator	“System Objects in MATLAB Code Generation”
comm.CPFSKModulator	“System Objects in MATLAB Code Generation”
comm.CPMDemodulator	“System Objects in MATLAB Code Generation”
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”
Trellis Coded Modulation	
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”
Orthogonal Frequency-Division Modulation	
comm.OFDMDemodulator	“System Objects in MATLAB Code Generation”
comm.OFDMModulator	“System Objects in MATLAB Code Generation”
Analog Baseband Modulation	
comm.FMBroadcastDemodulator	“System Objects in MATLAB Code Generation”
comm.FMBroadcastModulator	“System Objects in MATLAB Code Generation”
comm.FMDemodulator	“System Objects in MATLAB Code Generation”
comm.FMModulator	“System Objects in MATLAB Code Generation”
Filtering	
comm.IntegrateAndDumpFilter	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
comm.RaisedCosineReceiveFilter	“System Objects in MATLAB Code Generation”
comm.RaisedCosineTransmitFilter	“System Objects in MATLAB Code Generation”
Carrier Phase Synchronization	
comm.CarrierSynchronizer	“System Objects in MATLAB Code Generation”
comm.CPMCarrierPhaseSynchronizer	“System Objects in MATLAB Code Generation”
comm.CoarseFrequencyCompensator	“System Objects in MATLAB Code Generation”
Timing Phase Synchronization	
comm.SymbolSynchronizer	“System Objects in MATLAB Code Generation”
comm.GMSKTimingSynchronizer	“System Objects in MATLAB Code Generation”
comm.MSKTimingSynchronizer	“System Objects in MATLAB Code Generation”
Synchronization Utilities	
comm.DiscreteTimeVCO	“System Objects in MATLAB Code Generation”
Equalization	
comm.MLSEEqualizer	“System Objects in MATLAB Code Generation”
MIMO	
comm.LTEMIMOChannel	“System Objects in MATLAB Code Generation”
comm.MIMOChannel	“System Objects in MATLAB Code Generation”
comm.OSTBCCCombiner	“System Objects in MATLAB Code Generation”
comm.OSTBCEncoder	“System Objects in MATLAB Code Generation”
comm.SphereDecoder	“System Objects in MATLAB Code Generation”
Channel Modeling and RF Impairments	
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”

Name	Remarks and Limitations
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	—
Measurements and Analysis	
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"> Row vector input is only supported when the first two inputs are vectors and nonscalar

Function	Remarks and Limitations
	<ul style="list-style-type: none"> Performs arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors

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
Feature Detection, Extraction, and Matching	
BRISKPoints	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>visionRecovertoformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
cornerPoints	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>visionRecovertoformCodeGeneration_kernel.m</code> , which is used in the “Introduction to Code Generation with Feature Matching and Registration” example.
detectBRISKFeatures	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectFASTFeatures	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectHarrisFeatures	Compile-time constant input: 'FilterSize' Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.

Name	Remarks and Limitations
detectMinEigenFeatures	Compile-time constant input: 'FilterSize' Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
detectMSERFeatures	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> .
detectSURFFeatures	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
extractFeatures	Generates platform-dependent library: Yes for BRISK, FREAK, and SURF methods only. Compile-time constant input restrictions: 'Method' Supports MATLAB Function block: Yes for Block method only. Generated code for this function uses a precompiled platform-specific shared library.
extractHOGFeatures	Supports MATLAB Function block: No
extractLBPFeatures	Generates platform-dependent library: No Supports MATLAB Function block: Yes
matchFeatures	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

Name	Remarks and Limitations
MSERRegions	Supports MATLAB Function block: Yes For code generation, you must specify both the <code>pixellist</code> cell array and the <code>length</code> of each array, as the second input. The object outputs, <code>regions.Pixellist</code> as an array. The region sizes are defined in <code>regions.Lengths</code> . Generated code for this function uses a precompiled platform-specific shared library.
SURFPoints	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”
Image Registration and Geometric Transformations	
estimateGeometricTransform	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”
Object Detection and Recognition	

Name	Remarks and Limitations
ocr	Compile-time constant input: 'TextLayout', 'Language', and 'CharacterSet'. Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
ocrText	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"
Tracking and Motion Estimation	
assignDetectionsToTracks	Supports MATLAB Function block: Yes
opticalFlowFarneback	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlowHS	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlowLKDoG	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlowLK	Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
opticalFlow	Supports MATLAB Function block: Yes

Name	Remarks and Limitations
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.PointTracker	Supports MATLAB Function block: No “System Objects in MATLAB Code Generation”
vision.TemplateMatcher	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
Camera Calibration and Stereo Vision	
bboxOverlapRatio	Supports MATLAB Function block: No
bbox2points	Supports MATLAB® Function block: Yes
disparity	Compile-time constant input restriction: 'Method'. Supports MATLAB Function block: No Generated code for this function uses a precompiled platform-specific shared library.
cameraMatrix	Supports MATLAB Function block: No
cameraPose	Supports MATLAB Function block: No
cameraParameters	Supports MATLAB Function block: No Use the toStruct method to pass a cameraParameters object into generated code. See the “Code Generation for Depth Estimation From Stereo Video” example.

Name	Remarks and Limitations
detectCheckerboardPoints	<p>Supports MATLAB Function block: No Code generation will not support specifying images as file names or cell arrays of file names. It supports only checkerboard detection in a single image or stereo pair of images. For example, these syntaxes are supported:</p> <ul style="list-style-type: none"> • detectCheckerboardPoints(I1) • detectCheckerboardPoints(I1, I2) <p>I1 and I2 are single grayscale or RGB images.</p>
generateCheckerboardPoints	Supports MATLAB Function block: No
epipolarline	Supports MATLAB Function block: Yes
estimateFundamentalMatrix	<p>Compile-time constant input restriction: 'Method', 'OutputClass', 'DistanceType', and 'ReportRuntimeError'. Supports MATLAB Function block: Yes</p>
estimateUncalibratedRectification	Supports MATLAB Function block: Yes
extrinsics	Supports MATLAB Function block: No
isEpipoleInImage	Supports MATLAB Function block: Yes
lineToBorderPoints	Supports MATLAB Function block: Yes
reconstructScene	Supports MATLAB Function block: No
rectifyStereoImages	<p>Compile-time constant input restriction: 'interp' and 'OutputView'. Supports MATLAB Function block: No</p>
selectStrongestBbox	Supports MATLAB Function block: No
stereoParameters	<p>Supports MATLAB Function block: No Use the toStruct method to pass a stereoParameters object into generated code. See the “Code Generation for Depth Estimation From Stereo Video” example.</p>
triangulate	Supports MATLAB Function block: No

Name	Remarks and Limitations
undistortImage	Compile-time constant input restriction: 'interp' and 'OutputView' Supports MATLAB Function block: No
Statistics	
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”
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”
Filters, Transforms, and Enhancements	
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”
Video Loading, Saving, and Streaming	
vision.DeployableVideoPlayer	Supports MATLAB Function block: Yes 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” Does not generate code for reading compressed images on the Mac.
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”
Color Space Formatting and Conversions	
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”
Graphics	
insertMarker	Compile-time constant input: 'Shape' and 'Color' Supports MATLAB Function block: Yes

Name	Remarks and Limitations
insertShape	Compile-time constant input: 'Color' and 'SmoothEdges' Supports MATLAB Function block: Yes
insertText	Compile-time constant input: Font, FontSize Supports non-ASCII characters: No Supports MATLAB Function block: Yes
vision.AlphaBlender	Supports MATLAB Function block: Yes “System Objects in MATLAB Code Generation”
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"> • Treated as a for-loop in a MATLAB Function block or when used with <code>fiaccel</code>. • See the <code>parfor</code> reference page in the MATLAB Coder documentation. • “Generate Code with Parallel for-Loops (<code>parfor</code>)” on page 29-32.
return	—
switch, case, otherwise	<ul style="list-style-type: none"> • 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.

Function	Remarks and Limitations
	<ul style="list-style-type: none"> 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.
<code>while</code>	—

Data and File Management in MATLAB

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

Function	Remarks and Limitations
	<pre data-bbox="437 298 986 378">% fopen successful, okay to call fread A = fread(fid); ...</pre> <ul data-bbox="408 395 1305 586" style="list-style-type: none"><li data-bbox="408 395 1305 460">• The behavior of the generated code for <code>fread</code> is compiler-dependent if you:<ol data-bbox="437 482 1283 586" style="list-style-type: none"><li data-bbox="437 482 1120 512">1 Open a file using <code>fopen</code> with a permission of <code>a+</code>.<li data-bbox="437 524 1283 586">2 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.

Function	Remarks and Limitations
fprintf	<ul style="list-style-type: none"> • Does not support: <ul style="list-style-type: none"> • <code>b</code> and <code>t</code> subtypes on <code>%u</code>, <code>%o</code>, <code>%x</code>, and <code>%X</code> formats. • <code>\$</code> flag for reusing input arguments. • printing arrays. • There is no automatic casting. Input arguments must match their format types for predictable results. • Escaped characters are limited to the decimal range of 0–127. • 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"> • The <code>fprintf</code> call is inside a <code>parfor</code> loop. • Extrinsic calls are disabled. • 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"> • The format specifier has a corresponding C format specifier, for example, <code>%e</code> or <code>%E</code>. • The <code>fprintf</code> call is inside a <code>parfor</code> loop. • Extrinsic calls are disabled. • 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. • 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.

Function	Remarks and Limitations
fread	<ul style="list-style-type: none"> • precision must be a constant. • The source and output that precision specifies cannot have values long, ulong, unsigned long, bitN, or ubitN. • You cannot use the machineformat input. • 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"> • Match. • Map directly to a MATLAB type. • The source type that precision specifies must map directly to a C type on the target hardware. • If the fread call reads the entire file, all of the data must fit in the largest array available for code generation. • If sizeA is not constant or contains a nonfinite element, then dynamic memory allocation is required. • Treats a char value for source or output as a signed 8-bit integer. Use values between 0 and 127 only. • 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>
frewind	—

Function	Remarks and Limitations
load	<ul style="list-style-type: none"> • Use only when generating MEX or code for Simulink simulation. To load compile-time constants, use <code>coder.load</code>. • 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>. • 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. • Arguments to <code>load</code> must be compile-time constant strings. • Does not support loading objects. • If the MAT-file contains unsupported constructs, use <code>load(filename, variables)</code> to load only the supported constructs. • 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. <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
cell	“Cell Array Requirements and Limitations for Code Generation” on page 9-10
deal	—
fieldnames	Does not support objects. The input must be a structure.
iscell	—
isfield	Does not support cell arrays for the second argument.
isobject	—
isstruct	—

Function	Remarks and Limitations
nargchk	<ul style="list-style-type: none"> Output structure does not include stack information. <hr/> <p>Note: nargchk will be removed in a future release.</p>
narginchk	—
nargoutchk	—
str2func	<ul style="list-style-type: none"> String must be constant/known at compile time
struct	<ul style="list-style-type: none"> You cannot create a structure that contains a cell array. For example, you cannot generate code for: <code>s = struct('a',{{1 2}})</code> If the value argument is a cell array, all elements must have the same type.
struct2cell	<ul style="list-style-type: none"> For a variable-size structure array, the resulting cell array must be homogeneous. If <code>s</code> is a variable-size structure array, the fields must have the same type. If <code>struct2cell</code> cannot convert <code>s</code> to a homogeneous cell array, the output cell array is heterogeneous. A heterogeneous output cell array can have a maximum of 1024 elements.
structfun	<ul style="list-style-type: none"> Does not support the <code>ErrorHandler</code> option. The number of outputs must be less than or equal to three.

Desktop Environment in MATLAB

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

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

Discrete Math in MATLAB

Function	Remarks and Limitations
factor	<ul style="list-style-type: none"> The maximum double precision input is 2^{33}. The maximum single precision input is 2^{25}. The input n cannot have type <code>int64</code> or <code>uint64</code>.
gcd	—
isprime	<ul style="list-style-type: none"> The maximum double precision input is 2^{33}. The maximum single precision input is 2^{25}. The input X cannot have type <code>int64</code> or <code>uint64</code>.
lcm	—
nchoosek	<ul style="list-style-type: none"> When the first input, x, is a scalar, <code>nchoosek</code> returns a binomial coefficient. In this case, x must be a nonnegative integer. It cannot have type <code>int64</code> or <code>uint64</code>. When the first input, x, is a vector, <code>nchoosek</code> treats it as a set. In this case, x can have type <code>int64</code> or <code>uint64</code>. The second input, k, cannot have type <code>int64</code> or <code>uint64</code>. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
primes	<ul style="list-style-type: none"> The maximum double precision input is 2^{32}. The maximum single precision input is 2^{24}. The input n cannot have type <code>int64</code> or <code>uint64</code>.

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 Using the App” on page 18-23 and “Specify Constant Inputs at the Command Line” on page 21-53.

Name	Remarks and Limitations
Estimation	
dsp.BurgAREstimator	“System Objects in MATLAB Code Generation”
dsp.BurgSpectrumEstimator	“System Objects in MATLAB Code Generation”
dsp.CepstralToLPC	“System Objects in MATLAB Code Generation”
dsp.CrossSpectrumEstimator	“System Objects in MATLAB Code Generation”
dsp.LevinsonSolver	“System Objects in MATLAB Code Generation”
dsp.LPCToAutocorrelation	“System Objects in MATLAB Code Generation”
dsp.LPCToCepstral	“System Objects in MATLAB Code Generation”
dsp.LPCToLSF	“System Objects in MATLAB Code Generation”
dsp.LPCToLSP	“System Objects in MATLAB Code Generation”
dsp.LPCToRC	“System Objects in MATLAB Code Generation”
dsp.LSFToLPC	“System Objects in MATLAB Code Generation”
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”
Filters	
ca2tf	All inputs must be constant. Expressions or variables are allowed if their values do not change.
c12tf	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.AllpassFilter	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
	The System object supports code generation only when the Structure property is set to Minimum multiplier or Lattice.
dsp.AllpoleFilter	<ul style="list-style-type: none"> • “System Objects in MATLAB Code Generation” • Only the Denominator property is tunable for code generation.
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”
dsp.FilterCascade	<ul style="list-style-type: none"> • 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. • “System Objects in MATLAB Code Generation”
dsp.FilteredXLMSFilter	“System Objects in MATLAB Code Generation”
dsp.FIRDecimator	“System Objects in MATLAB Code Generation”
dsp.FIRFilter	<ul style="list-style-type: none"> • “System Objects in MATLAB Code Generation” • Only the Numerator property is tunable for code generation.
dsp.FIRHalfbandDecimator	“System Objects in MATLAB Code Generation”
dsp.FIRHalfbandInterpolator	“System Objects in MATLAB Code Generation”
dsp.FIRInterpolator	“System Objects in MATLAB Code Generation”

Name	Remarks and Limitations
dsp.FIRRateConverter	“System Objects in MATLAB Code Generation”
dsp.FrequencyDomainAdaptiveFilter	“System Objects in MATLAB Code Generation”
dsp.HighpassFilter	“System Objects in MATLAB Code Generation”
dsp.IIRFilter	<ul style="list-style-type: none"> • Only the Numerator and Denominator properties are tunable for code generation. • “System Objects in MATLAB Code Generation”
dsp.IIRHalfbandDecimator	“System Objects in MATLAB Code Generation”
dsp.IIRHalfbandInterpolator	“System Objects in MATLAB Code Generation”
dsp.KalmanFilter	“System Objects in MATLAB Code Generation”
dsp.LMSFilter	“System Objects in MATLAB Code Generation”
dsp.LowpassFilter	“System Objects in MATLAB Code Generation”
dsp.RLSFilter	“System Objects in MATLAB Code Generation”
dsp.SampleRateConverter	“System Objects in MATLAB Code Generation”
firceqrip	All inputs must be constant. Expressions or variables are allowed if their values do not change.
fireqint	All inputs must be constant. Expressions or variables are allowed if their values do not change.
firgr	<ul style="list-style-type: none"> • All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
firhalfband	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Name	Remarks and Limitations
firlpnorm	<ul style="list-style-type: none"> • All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
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.
iirgrpdelay	<ul style="list-style-type: none"> • All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.
iirlpnorm	<ul style="list-style-type: none"> • All inputs must be constant. Expressions or variables are allowed if their values do not change. • Does not support syntaxes that have cell array input.

Name	Remarks and Limitations
iirlpnormc	<ul style="list-style-type: none"> All inputs must be constant. Expressions or variables are allowed if their values do not change. Does not support syntaxes that have cell array input.
iirnotch	All inputs must be constant. Expressions or variables are allowed if their values do not change.
iirparameq	—
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.
Math Operations	
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”
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”

Name	Remarks and Limitations
Quantizers	
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”
Scopes	
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.
Signal Management	
dsp.Counter	“System Objects in MATLAB Code Generation”
dsp.DelayLine	“System Objects in MATLAB Code Generation”
Signal Operations	
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”
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”

Name	Remarks and Limitations
dsp.Window	<ul style="list-style-type: none"> This object has no tunable properties for code generation. “System Objects in MATLAB Code Generation”
dsp.ZeroCrossingDetector	“System Objects in MATLAB Code Generation”
Sinks	
dsp.AudioPlayer	“System Objects in MATLAB Code Generation”
dsp.AudioFileWriter	“System Objects in MATLAB Code Generation”
dsp.UDPSender	“System Objects in MATLAB Code Generation”
Sources	
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"> This object has no tunable properties for code generation. “System Objects in MATLAB Code Generation”
dsp.UDPReceiver	“System Objects in MATLAB Code Generation”
Statistics	
dsp.Autocorrelator	“System Objects in MATLAB Code Generation”
dsp.Crosscorrelator	“System Objects in MATLAB Code Generation”
dsp.Histogram	<ul style="list-style-type: none"> This object has no tunable properties for code generation. “System Objects in MATLAB Code Generation”
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”

Name	Remarks and Limitations
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”
Transforms	
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"> 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. If called with more than 1 argument, has no effect in standalone code even when run-time error detection is enabled. See “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 22-28. See “Rules for Using assert Function” on page 21-73.
error	Has no effect in standalone code even when run-time error detection is enabled. See “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 22-28.

Exponents in MATLAB

Function	Remarks and Limitations
exp	—

Function	Remarks and Limitations
expm	—
expm1	—
factorial	—
log	<ul style="list-style-type: none"> 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>.
log2	—
log10	—
log1p	—
nextpow2	—
nthroot	—
reallog	—
realpow	—
realsqrt	—
sqrt	<ul style="list-style-type: none"> 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>.

Filtering and Convolution in MATLAB

Function	Remarks and Limitations
conv	—
conv2	—
convn	—
deconv	—
detrend	<ul style="list-style-type: none"> If supplied and not empty, the input argument <code>bp</code> must satisfy the following requirements: <ul style="list-style-type: none"> Be real.

Function	Remarks and Limitations
	<ul style="list-style-type: none"> • Be sorted in ascending order. • Restrict elements to integers in the interval $[1, n-2]$. n is the number of elements in a column of input argument X, or the number of elements in X when X is a row vector. • Contain all unique values. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
<code>filter</code>	—
<code>filter2</code>	—

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” on page 2-13.

Function	Remarks/Limitations
abs	N/A
accumneg	N/A
accumpos	N/A
add	<ul style="list-style-type: none"> 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.
bitxorreduce	N/A
ceil	N/A
complex	N/A

Function	Remarks/Limitations
conj	N/A
conv	<ul style="list-style-type: none"> • 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>. • For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> • In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. • 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>.
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
diag	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.

Function	Remarks/Limitations
<code>divide</code>	<ul style="list-style-type: none"> 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. Complex and imaginary divisors are not supported. Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
<code>double</code>	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”.
<code>end</code>	N/A
<code>eps</code>	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
<code>eq</code>	Not supported for fixed-point signals with different biases.
<code>fi</code>	<ul style="list-style-type: none"> The default constructor syntax without any input arguments is not supported. 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. All properties related to data type must be constant for code generation. <code>numericType</code> object information must be available for nonfixed-point Simulink inputs.
<code>filter</code>	<ul style="list-style-type: none"> 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>.

Function	Remarks/Limitations
<code>fimath</code>	<ul style="list-style-type: none"> 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. Use to create <code>fimath</code> objects in the generated code. 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. 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.
<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"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
<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"> • When the exponent <code>k</code> is a variable and the input is a scalar, the ProductMode property of the governing <code>fimath</code> must be SpecifyPrecision. • When the exponent <code>k</code> is a variable and the input is not scalar, the SumMode property of the governing <code>fimath</code> must be SpecifyPrecision. • Variable-sized inputs are only supported when the SumMode property of the governing <code>fimath</code> is set to SpecifyPrecision or Keep LSB. • For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> • In the generated code, the output for variable-sized signals is computed using the SumMode property of the governing <code>fimath</code>. • In MATLAB, the output for variable-sized signals is computed using the SumMode 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 ProductMode of the governing <code>fimath</code>.
mpy	<ul style="list-style-type: none"> • 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>. • 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 Ports and data manager and set the Complexity parameter for all known complex inputs to On.
mrdivide	N/A

Function	Remarks/Limitations
<code>mtimes</code>	<ul style="list-style-type: none"> 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. 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>. For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> In the generated code, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code>. 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>.
<code>ndims</code>	N/A
<code>ne</code>	Not supported for fixed-point signals with different biases.
<code>nearest</code>	N/A
<code>numberofelements</code>	<code>numberofelements</code> will be removed in a future release. Use <code>numel</code> instead.
<code>numel</code>	N/A
<code>numerictype</code>	<ul style="list-style-type: none"> 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. Returns the data type when the input is a nonfixed-point signal. Use to create <code>numerictype</code> objects in generated code. All <code>numerictype</code> object properties related to the data type must be constant.
<code>permute</code>	The dimensions argument must be a built-in type; it cannot be a <code>fi</code> object.
<code>plus</code>	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.
<code>pow2</code>	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"> All properties related to data type must be constant for code generation.
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"> Complex and [Slope Bias] inputs error out. Negative inputs yield a 0 result.
storedInteger	N/A
storedIntegerToDouble	N/A
sub	<ul style="list-style-type: none"> 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>.
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"> 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. 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 Ports and data manager and set the Complexity parameter for all known complex inputs to <code>On</code>.
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"> All properties related to data type must be constant for code generation.
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"> • Histogram bar plotting not supported. Call with at least one output argument. • If supplied, the second argument <code>x</code> must be a scalar constant. • Inputs must be real. <p>For the syntax <code>[nout, xout] = hist(y,x)</code>:</p> <ul style="list-style-type: none"> • When <code>y</code> is a fixed-size vector or variable-length vector: <ul style="list-style-type: none"> • <code>nout</code> is always a row vector. • If <code>x</code> is a vector, <code>xout</code> is a vector with the same orientation as <code>x</code>. • If <code>x</code> is a scalar (fixed-size), <code>xout</code> is a row vector. • <code>nout</code> and <code>xout</code> are column vectors when the following conditions are true: <ul style="list-style-type: none"> • <code>y</code> is a matrix • <code>size(y,1)</code> and <code>size(y,2)</code> do not have fixed length 1 • One of <code>size(y,1)</code> and <code>size(y,2)</code> has length 1 at run time • A variable-size <code>x</code> is interpreted as a vector input even if it is a scalar at run time. • If at least one of the inputs is empty, vector orientations in the output can differ from MATLAB.

Function	Remarks and Limitations
histc	<ul style="list-style-type: none"> • The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector. • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

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	—
rgb2gray	—

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>bwareaopen</code>	BW must be a 2-D binary image. N-D arrays are not supported. <code>conn</code> can only be one of the two-dimensional connectivities (4 or 8) or a 3-by-3 matrix. The 3-D connectivities (6, 18, and 26) are not supported. Matrices of size 3-by-3-by-...-by-3 are not supported. <code>conn</code> must be a compile-time constant.
<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>bweuler</code>	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
<code>bwlabel</code>	When generating code, the parameter <code>n</code> must be a compile-time constant.
<code>bwlookup</code>	For best results, specify an input image of class <code>logical</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
<code>bwmorph</code>	The text string specifying the operation must be a constant and, for best results, specify an input image of class <code>logical</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.

Function	Remarks/Limitations
<code>bwpack</code>	Generated code for this function uses a precompiled platform-specific shared library.
<code>bwperim</code>	<p>Supports only 2-D images. Does not support any no-output-argument syntaxes. The connectivity matrix input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<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>If you choose the generic MATLAB Host Computer target platform, generated code 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>If you choose the generic MATLAB Host Computer target platform, generated code 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: 'nonreflectivesimilarity', 'similarity', 'affine', or 'projective'.</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
grayconnected	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
histeq	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> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
hough	The optional parameter names 'Theta' and 'RhoResolution' must be compile-time string constants. The optional Theta vector must have a bounded size.
houghlines	The optional parameter names 'FillGap' and 'MinLength' must be compile-time string constants. Their associated values need not be compile-time constants.
houghpeaks	The optional parameter names 'Threshold' and 'NHoodSize' must be compile-time string constants. Their associated values need not be compile-time constants.
im2int16	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
im2uint8	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
im2uint16	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
im2single	—
im2double	—
imabsdiff	—
imadjust	Does not support syntaxes that include indexed images. This includes all syntaxes that accept <code>map</code> as input and return <code>newmap</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.

Function	Remarks/Limitations
imbothat	<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>
imclearborder	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant. Supports only up to 3-D inputs.</p> <p>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, platform-specific shared library.</p>
imclose	<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>
imcomplement	<p>Does not support <code>int64</code> and <code>uint64</code> data types.</p>
imcrop	<p>The interactive syntaxes, such as <code>I2 = imcrop</code>, are not supported. Indexed images are not supported, including the noninteractive syntax <code>X2 = imcrop(X, map, rect);</code>.</p>
imdilate	<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>

Function	Remarks/Limitations
imerode	<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>
imextendedmax	<p>The optional third input argument, <code>conn</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>
imextendedmin	<p>The optional third input argument, <code>conn</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>
imfill	<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.</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 P-by-1 vector, in which case it contains the linear indices of the starting locations. <code>locations</code> can also be a P-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>If you choose the generic <code>MATLAB Host Computer</code> target platform, generated code uses a precompiled, platform-specific shared library.</p>

Function	Remarks/Limitations
<code>imfilter</code>	<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 MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>imgaborfilt</code>	—
<code>imhist</code>	<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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>imhmax</code>	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>imhmin</code>	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>imlincomb</code>	<p>The <code>output_class</code> argument must be a compile-time constant. You can specify up to four input image arguments.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>immse</code>	—
<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 MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>imquantize</code>	—

Function	Remarks/Limitations
imreconstruct	<p>The optional third input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imref2d	<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>
imref3d	<p>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.</p>
imregionalmax	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imregionalmin	<p>The optional second input argument, <code>conn</code>, must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imresize	<p>Does not support indexed images or custom interpolation kernels. The <code>Scale</code>, <code>OutputSize</code> and all parameter-value pair input arguments must be compile-time constants.</p>
imrotate	<p>The <code>angle</code>, <code>method</code>, and <code>bbox</code> arguments must be compile-time constants.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
imtophat	<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 MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>

Function	Remarks/Limitations
<code>imtranslate</code>	<p>The supports only 2-D translation vectors. 3-D translations are not supported</p> <p>If you choose the generic MATLAB Host Computer 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>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>integralBoxFilter</code>	<p>The <code>FilterSize</code> input argument and the 'NormalizationFactor' parameter must be compile-time constants</p>
<code>intlut</code>	<p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>
<code>iptcheckconn</code>	<p>Input arguments must be compile-time constants.</p>
<code>iptcheckmap</code>	<p>—</p>
<code>label2rgb</code>	<p>Referring to the standard syntax:</p> <pre>RGB = label2rgb(L, map, zerocolor, order)</pre> <ul style="list-style-type: none"> • Submit at least two input arguments: the label matrix, <code>L</code>, and the colormap matrix, <code>map</code>. • <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. • 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. • If you supply a value for <code>order</code>, it must be 'noshuffle'.
<code>mean2</code>	<p>—</p>
<code>medfilt2</code>	<p>The <code>padopt</code> argument must be a compile-time constant.</p> <p>If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.</p>

Function	Remarks/Limitations
<code>multithresh</code>	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
<code>ordfilt2</code>	The <code>padopt</code> argument must be a compile-time constant. If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
<code>padarray</code>	Support only up to 3-D inputs. Input arguments, <code>padval</code> and <code>direction</code> are expected to be compile-time constants.
<code>projective2d</code>	When generating code, you can only specify single objects—arrays of objects are not supported.
<code>psnr</code>	
<code>regionprops</code>	Supports only 2-D images. Does not accept the connected component structure (CC) returned by <code>bwconncomp</code> . Use <code>bwlabel</code> to create a label matrix, or pass the image to <code>regionprops</code> directly. Does not support the table output type. Does not accept cell arrays as input—use a comma-separated list instead. Does not support the properties <code>ConvexArea</code> , <code>ConvexHull</code> , <code>ConvexImage</code> , <code>Solidity</code> , and <code>SubarrayIdx</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
<code>rgb2gray</code>	—
<code>rgb2ycbcr</code>	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
<code>strel</code>	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.
<code>stretchlim</code>	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.

Function	Remarks/Limitations
watershed	Supports only 2-D images. Supports only 4- or 8-connectivity. Supports only up to 65,535 regions. The output type is always <code>uint16</code> . If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.
ycbcr2rgb	If you choose the generic MATLAB Host Computer target platform, generated code uses a precompiled, platform-specific shared library.

Input and Output Arguments in MATLAB

Function	Remarks and Limitations
nargin	—
nargout	<ul style="list-style-type: none"> For a function with no output arguments, returns 1 if called without a terminating semicolon. <p>Note: This behavior also affects extrinsic calls with no terminating semicolon. <code>nargout</code> is 1 for the called function in MATLAB.</p>

Interpolation and Computational Geometry in MATLAB

Function	Remarks and Limitations
cart2pol	—
cart2sph	—
interp1	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
interp1q	Might not match MATLAB when some Y values are <code>Inf</code> or <code>NaN</code> .
interp2	<ul style="list-style-type: none"> <code>Xq</code> and <code>Yq</code> must be the same size. Use <code>meshgrid</code> to evaluate on a grid. For best results, provide <code>X</code> and <code>Y</code> as vectors. 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.

Function	Remarks and Limitations
	<ul style="list-style-type: none"> • For best results when you use the 'spline' method: <ul style="list-style-type: none"> • Use <code>meshgrid</code> to create the inputs <code>Xq</code> and <code>Yq</code>. • 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.
<p><code>interp3</code></p>	<ul style="list-style-type: none"> • <code>Xq</code>, <code>Yq</code>, and <code>Zq</code> must be the same size. Use <code>meshgrid</code> to evaluate on a grid. • For best results, provide <code>X</code>, <code>Y</code>, and <code>Z</code> as vectors. • For the 'cubic' method, reports an error if the grid does not have uniform spacing. In this case, use the 'spline' method. • For best results when you use the 'spline' method: <ul style="list-style-type: none"> • Use <code>meshgrid</code> to create the inputs <code>Xq</code>, <code>Yq</code>, and <code>Zq</code>. • 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.
<p><code>meshgrid</code></p>	<p>—</p>

Function	Remarks and Limitations
mkpp	<ul style="list-style-type: none"> • 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. <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"> • In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB. • In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure. <ul style="list-style-type: none"> • 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. • To define a piecewise constant polynomial, <code>coefs</code> must be a column vector or <code>d</code> must have at least two elements. • 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. • 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"> 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> 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> • If you do not provide <code>d</code>, the following statements must be true:

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"> • Input x must be strictly increasing. • Does not remove y entries with NaN values. • 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"> • In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB. • In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.
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"> • The input x is a variable-size array that is not a variable-length vector. • x becomes a row vector at run time. 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, $\text{size}(x,1) = 1$ and $\text{size}(x,2) = 5$. In the generated code, the $\text{size}(v)$ 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"> • Input <code>x</code> must be strictly increasing. • Does not remove <code>Y</code> entries with NaN values. • Does not report an error for infinite end slopes in <code>Y</code>. • 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"> • In code generation, use <code>unmkpp</code> to return the piecewise polynomial details to MATLAB. • In MATLAB, use <code>mkpp</code> to create the <code>pp</code> structure.
unmkpp	<ul style="list-style-type: none"> • <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. • Does not support <code>pp</code> structures created by <code>interp1</code> in MATLAB.

Linear Algebra in MATLAB

Function	Remarks and Limitations
bandwidth	—
isbanded	—
isdiag	—
ishermitian	—
istril	—
istriu	—
issymmetric	—
linsolve	<ul style="list-style-type: none"> • The option structure must be a constant. • Supports only a scalar option structure input. It does not support arrays of option structures. • Only optimizes these cases: <ul style="list-style-type: none"> • UT • LT

Function	Remarks and Limitations
	<ul style="list-style-type: none"> • UHESS = true (the TRANSA can be either true or false) • SYM = true and POSDEF = true Other options are equivalent to using <code>mldivide</code> .
<code>lsqnonneg</code>	<ul style="list-style-type: none"> • You must enable support for variable-size arrays. • The message string in the output structure <code>output</code> (the fifth output) is not translated.
<code>null</code>	<ul style="list-style-type: none"> • Might return a different basis than MATLAB • Does not support rational basis option (second input)
<code>orth</code>	<ul style="list-style-type: none"> • Can return a different basis than MATLAB
<code>rsf2csf</code>	—
<code>schur</code>	Can return a different Schur decomposition in generated code than in MATLAB.
<code>sqrtn</code>	—

Logical and Bit-Wise Operations in MATLAB

Function	Remarks and Limitations
<code>and</code>	—
<code>bitand</code>	—
<code>bitcmp</code>	—
<code>bitget</code>	—
<code>bitor</code>	—
<code>bitset</code>	—
<code>bitshift</code>	—
<code>bitxor</code>	—
<code>not</code>	—
<code>or</code>	—
<code>xor</code>	—

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

Matrices and Arrays in MATLAB

Function	Remarks and Limitations
abs	—
all	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
angle	—
any	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
blkdiag	—
bsxfun	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
cat	<ul style="list-style-type: none"> • Does not support concatenation of cell arrays. • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
circshift	Does not support cell arrays for the first argument.
colon	<ul style="list-style-type: none"> • Does not accept complex inputs. • The input <code>i</code> cannot have a logical value. • Does not accept vector inputs. • Inputs must be constants. • Uses single-precision arithmetic to produce single-precision results.

Function	Remarks and Limitations
companion	—
cond	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
cov	<ul style="list-style-type: none"> • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 • Does not support the <code>nanflag</code> argument.
cross	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
cumprod	Does not support logical inputs. Cast input to <code>double</code> first.
cumsum	Does not support logical inputs. Cast input to <code>double</code> first.
det	—

Function	Remarks and Limitations
diag	<ul style="list-style-type: none"> • If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value. • For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), diag: <ul style="list-style-type: none"> • Treats the input as a vector input. • Returns a matrix with the given vector along the specified diagonal. • For variable-size inputs that are not variable-length vectors, diag: <ul style="list-style-type: none"> • Treats the input as a matrix. • Does not support inputs that are vectors at run time. • Returns a variable-length vector. <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> • For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), diag 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 diag to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use: <ul style="list-style-type: none"> • <code>diag(x(:))</code> instead of <code>diag(x)</code> • <code>diag(x(:),k)</code> instead of <code>diag(x,k)</code> • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
diff	<ul style="list-style-type: none"> • If supplied, the arguments representing the number of times to apply diff and the dimension along which to calculate the difference must be constants. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
dot	—

Function	Remarks and Limitations
eig	<ul style="list-style-type: none"> • For code generation, QZ algorithm is used in all cases. MATLAB can use different algorithms for different inputs. Consequently, V might represent a different basis of eigenvectors. The eigenvalues in D might not be in the same order as in MATLAB. • With one input, $[V, D] = \text{eig}(A)$, the results are similar to those obtained using $[V, D] = \text{eig}(A, \text{eye}(\text{size}(A)), 'qz')$ in MATLAB, except that for code generation, the columns of V are normalized. • Options 'balance', and 'nobalance' are not supported for the standard eigenvalue problem. 'chol' is not supported for the symmetric generalized eigenvalue problem. • Outputs are of complex type. • Does not support the option to calculate left eigenvectors.
eye	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size.
false	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integers.
find	<ul style="list-style-type: none"> • Issues an error if a variable-size input becomes a row vector at run time. <hr/> <p>Note: This limitation does not apply when the input is scalar or a variable-length row vector.</p> <ul style="list-style-type: none"> • For variable-size 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. • Always returns a variable-length vector. Even when you provide the output vector <code>k</code>, the output cannot be fixed-size because the output can contain fewer than <code>k</code> elements. For example, <code>find(x, 1)</code> returns a variable-length vector with 1 or 0 elements.
flip	<p>Does not support cell arrays for the first argument.</p>

Function	Remarks and Limitations
flipdim	Does not support cell arrays for the first argument.
flipplr	Does not support cell arrays.
flipud	Does not support cell arrays.
full	—
hadamard	n must be a fixed-size scalar.
hankel	—
hilb	—
ind2sub	<ul style="list-style-type: none"> • The first argument should be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
inv	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	—
ipermute	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
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” on page 7-35.
max	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35. • Does not support the <code>nanflag</code> argument. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
min	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35. • Does not support the <code>nanflag</code> argument. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
ndgrid	—
ndims	—
nnz	—
nonzeros	—
norm	—
normest	—
numel	—

Function	Remarks and Limitations
ones	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative integers. • The input <code>optimfun</code> must be a function supported for code generation.
pascal	—
permute	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
pinv	—
planerot	“Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
prod	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
qr	—
rand	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
randi	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Function	Remarks and Limitations
randn	<ul style="list-style-type: none"> • <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>. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
randperm	—
rank	—
rcond	—
repmat	Size arguments must have a fixed size.
reshape	<ul style="list-style-type: none"> • If the input is a compile-time empty cell array, then the size arguments must be constants. • Size arguments must have a fixed size. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
rng	<ul style="list-style-type: none"> • For library code generation targets, executable code generation targets, and MEX targets with extrinsic calls disabled: <ul style="list-style-type: none"> • Does not support the <code>'shuffle'</code> input. • For the <code>generator</code> input, supports <code>'twister'</code>, <code>'v4'</code>, and <code>'v5normal'</code>. <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> • For MEX targets, if extrinsic calls are enabled, you cannot access the data in the structure returned by <code>rng</code>.
rosser	—
rot90	Does not support cell arrays for the first argument.

Function	Remarks and Limitations
<code>shiftdim</code>	<ul style="list-style-type: none"> Does not support cell arrays for the first argument. Second argument must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
<code>sign</code>	—
<code>size</code>	—
<code>sort</code>	<ul style="list-style-type: none"> Does not support cell arrays for the first argument. 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 with Zero-Valued Imaginary Parts” on page 6-4.
<code>sortrows</code>	<ul style="list-style-type: none"> Does not support cell arrays for the first argument. 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 with Zero-Valued Imaginary Parts” on page 6-4.
<code>squeeze</code>	Does not support cell arrays.
<code>sub2ind</code>	<ul style="list-style-type: none"> The first argument must be a valid size vector. Size vectors for arrays with more than <code>intmax</code> elements are not supported. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
<code>subspace</code>	—
<code>sum</code>	<ul style="list-style-type: none"> Specify <code>dim</code> as a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 Does not support the <code>nanflag</code> argument.
<code>toeplitz</code>	—
<code>trace</code>	—

Function	Remarks and Limitations
<code>tril</code>	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
<code>triu</code>	<ul style="list-style-type: none"> If supplied, the argument representing the order of the diagonal matrix must be a real and scalar integer value.
<code>true</code>	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
<code>vander</code>	—
<code>wilkinson</code>	<code>n</code> must be a fixed-size scalar.
<code>zeros</code>	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.

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 the `genSim` function. See “Deploy Neural Network Functions”.

Numerical Integration and Differentiation in MATLAB

Function	Remarks and Limitations
<code>cumtrapz</code>	—
<code>de12</code>	—
<code>diff</code>	<ul style="list-style-type: none"> 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.
<code>gradient</code>	—
<code>ode23</code>	<ul style="list-style-type: none"> All <code>odeset</code> option arguments must be constant. Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function . You must provide at least the two output arguments <code>T</code> and <code>Y</code>. Input types must be homogeneous—all double or all single. Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.

Function	Remarks and Limitations
ode45	<ul style="list-style-type: none"> • All <code>odeset</code> option arguments must be constant. • Does not support a constant mass matrix in the options structure. Provide a mass matrix as a function . • You must provide at least the two output arguments <code>T</code> and <code>Y</code>. • Input types must be homogeneous—all double or all single. • Variable-sizing support must be enabled. Requires dynamic memory allocation when <code>tspan</code> has two elements or you use event functions.
odeget	The <code>name</code> argument must be constant.
odeset	All inputs must be constant.
quad2d	<ul style="list-style-type: none"> • 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.
quadgk	—
trapz	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Optimization Functions in MATLAB

Function	Remarks and Limitations
fminsearch	<ul style="list-style-type: none"> • Ignores the <code>Display</code> option. Does not print status information during execution. Test the <code>exitflag</code> output for the exit condition. • The output structure does not include the <code>algorithm</code> or <code>message</code> fields. • Ignores the <code>OutputFcn</code> and <code>PlotFcns</code> options.
fzero	<ul style="list-style-type: none"> • The first argument must be a function handle. Does not support structure, inline function, or string inputs for the first argument. • Supports up to three output arguments. Does not support the fourth output argument (the <code>output</code> structure).
optimget	Input parameter names must be constant.
optimset	<ul style="list-style-type: none"> • Does not support the syntax that has no input or output arguments:

Function	Remarks and Limitations
	<p><code>optimset</code></p> <ul style="list-style-type: none"> • Functions specified in the options must be supported for code generation. • The fields of the options structure <code>oldopts</code> must be fixed-size fields. • For code generation, optimization functions ignore the Display option. • 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.

Phased Array System Toolbox

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

Name	Remarks and Limitations
Antenna and Microphone Elements	
<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"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
<code>phased.CrossedDipoleAntennaElement</code>	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.

Name	Remarks and Limitations
phased.CustomAntennaElement	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.CustomMicrophoneElement	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.IsotropicAntennaElement	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.OmnidirectionalMicrophoneElement	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.ShortDipoleAntennaElement	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, and <code>plotResponse</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
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.

Name	Remarks and Limitations
polsignature	<ul style="list-style-type: none"> Does not support variable-size inputs. Supported only when output arguments are specified.
stokes	<ul style="list-style-type: none"> Does not support variable-size inputs. Supported only when output arguments are specified.
uv2azelpat	Does not support variable-size inputs.
uv2phithetapat	Does not support variable-size inputs.
Array Geometries and Analysis	
az2broadside	Does not support variable-size inputs.
broadside2az	Does not support variable-size inputs.
pilotcalib	Does not support variable-size inputs.
phased.ArrayGain	<ul style="list-style-type: none"> Does not support arrays containing polarized antenna elements, that is, the <code>phased.ShortDipoleAntennaElement</code> or <code>phased.CrossedDipoleAntennaElement</code> antennas. See “System Objects in MATLAB Code Generation”.
phased.ArrayResponse	See “System Objects in MATLAB Code Generation”.
phased.ConformalArray	<ul style="list-style-type: none"> <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. See “System Objects in MATLAB Code Generation”.
phased.ElementDelay	See “System Objects in MATLAB Code Generation”.

Name	Remarks and Limitations
phased.PartitionedArray	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.ReplicatedSubarray	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.SteeringVector	See “System Objects in MATLAB Code Generation”.
phased.UCA	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.ULA	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
phased.URA	<ul style="list-style-type: none"> • <code>pattern</code>, <code>patternAzimuth</code>, <code>patternElevation</code>, <code>plotResponse</code>, and <code>viewArray</code> methods are not supported. • See “System Objects in MATLAB Code Generation”.
Signal Radiation and Collection	
phased.Collector	See “System Objects in MATLAB Code Generation”.
phased.Radiator	See “System Objects in MATLAB Code Generation”.

Name	Remarks and Limitations
phased.WidebandCollector	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
phased.WidebandRadiator	See “System Objects in MATLAB Code Generation”.
sensorsig	Does not support variable-size inputs.
Waveforms	
ambgfun	Does not support variable-size inputs.
phased.FMCWWaveform	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.LinearFMWaveform	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.MFSKWaveform	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.PhaseCodedWaveform	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.RectangularWaveform	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
phased.SteppedFMWaveform	<ul style="list-style-type: none"> • <code>plot</code> method is not supported. • See “System Objects in MATLAB Code Generation”.
range2bw	Does not support variable-size inputs.

Name	Remarks and Limitations
range2time	Does not support variable-size inputs.
time2range	Does not support variable-size inputs.
unigrd	Does not support variable-size inputs.
Transmitters and Receivers	
delayseq	Does not support variable-size inputs.
noisepow	Does not support variable-size inputs.
phased.ReceiverPreamplifier	See “System Objects in MATLAB Code Generation”.
phased.Transmitter	See “System Objects in MATLAB Code Generation”.
systemp	Does not support variable-size inputs.
Beamforming	
cbfweights	Does not support variable-size inputs.
lcmvweights	Does not support variable-size inputs.
mvdweights	Does not support variable-size inputs.
phased.FrostBeamformer	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
phased.LCMVBeamformer	See “System Objects in MATLAB Code Generation”.
phased.MVDRBeamformer	See “System Objects in MATLAB Code Generation”.
phased.PhaseShiftBeamformer	See “System Objects in MATLAB Code Generation”.
phased.SteeringVector	See “System Objects in MATLAB Code Generation”.
phased.SubbandMVDRBeamformer	See “System Objects in MATLAB Code Generation”.

Name	Remarks and Limitations
phased.SubbandPhaseShiftBeamformer	See “System Objects in MATLAB Code Generation”.
phased.TimeDelayBeamformer	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
phased.TimeDelayLCMVBeamformer	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
sensorcov	Does not support variable-size inputs.
steervec	Does not support variable-size inputs.
Direction of Arrival (DOA) Estimation	
aicstest	Does not support variable-size inputs.
espritdoa	Does not support variable-size inputs.
mdltest	Does not support variable-size inputs.
phased.BeamscanEstimator	See “System Objects in MATLAB Code Generation”.
phased.BeamscanEstimator2D	See “System Objects in MATLAB Code Generation”.
phased.BeamspaceESPRITEstimator	See “System Objects in MATLAB Code Generation”.
phased.ESPRITEstimator	See “System Objects in MATLAB Code Generation”.
phased.GCCEstimator	See “System Objects in MATLAB Code Generation”.
phased.MVDREstimator	See “System Objects in MATLAB Code Generation”.
phased.MVDREstimator2D	See “System Objects in MATLAB Code Generation”.

Name	Remarks and Limitations
phased.RootMUSICEstimator	See “System Objects in MATLAB Code Generation”.
phased.RootWSFEstimator	See “System Objects in MATLAB Code Generation”.
phased.SumDifferenceMonopulseTracker	See “System Objects in MATLAB Code Generation”.
phased.SumDifferenceMonopulseTracker2D	See “System Objects in MATLAB Code Generation”.
rootmusicdoa	Does not support variable-size inputs.
spsmooth	Does not support variable-size inputs.
Space-Time Adaptive Processing (STAP)	
dopsteeringvec	Does not support variable-size inputs.
phased.ADPCACanceller	See “System Objects in MATLAB Code Generation”.
phased.AngleDopplerResponse	See “System Objects in MATLAB Code Generation”.
phased.DPCACanceller	See “System Objects in MATLAB Code Generation”.
phased.STAPSMIBeamformer	See “System Objects in MATLAB Code Generation”.
val2ind	Does not support variable-size inputs.
Signal Propagation and Environment	
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.
gccphat	Does not support variable-size inputs.
grazingang	Does not support variable-size inputs.
horizonrange	Does not support variable-size inputs.

Name	Remarks and Limitations
phased.BarrageJammer	See “System Objects in MATLAB Code Generation”.
phased.ConstantGammaClutter	See “System Objects in MATLAB Code Generation”.
phased.FreeSpace	<ul style="list-style-type: none"> • Requires dynamic memory allocation. See “Limitations for System Objects that Require Dynamic Memory Allocation”. • See “System Objects in MATLAB Code Generation”.
phased.RadarTarget	See “System Objects in MATLAB Code Generation”.
phased.TwoRayChannel	See “System Objects in MATLAB Code Generation”.
phased.WidebandFreeSpace	See “System Objects in MATLAB Code Generation”.
physconst	Does not support variable-size inputs.
surfacegamma	Does not support variable-size inputs.
surfclutterrcs	Does not support variable-size inputs.
Detection and System Analysis	
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	See “System Objects in MATLAB Code Generation”.
phased.MatchedFilter	<ul style="list-style-type: none"> • The CustomSpectrumWindow property is not supported. • See “System Objects in MATLAB Code Generation”.

Name	Remarks and Limitations
phased.RangeDopplerResponse	<ul style="list-style-type: none"> The CustomRangeWindow and the CustomDopplerWindow properties are not supported. See “System Objects in MATLAB Code Generation”.
phased.StretchProcessor	See “System Objects in MATLAB Code Generation”.
phased.TimeVaryingGain	See “System Objects in MATLAB 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"> Does not support variable-size inputs. The NonfluctuatingNoncoherent signal type is not supported.
rocsnr	<ul style="list-style-type: none"> Does not support variable-size inputs. The NonfluctuatingNoncoherent signal type is not supported.
shnidman	Does not support variable-size inputs.
stretchfreq2rng	Does not support variable-size inputs.
Motion Modeling and Coordinate Systems	
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.

Name	Remarks and Limitations
global2localcoord	Does not support variable-size inputs.
local2globalcoord	Does not support variable-size inputs.
phased.Platform	See “System Objects in MATLAB 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"> Does not discard nonfinite input values Complex input produces complex output “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
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” on page 7-35
polyint	—
polyval	—
polyvalm	—

Function	Remarks and Limitations
roots	<ul style="list-style-type: none"> • Output is variable size. • Output is complex. • Roots are not always in the same order as MATLAB. • Roots of poorly conditioned polynomials do not always match MATLAB.

Programming Utilities in MATLAB

Function	Remarks and Limitations
mfilename	—

Relational Operators in MATLAB

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

Robotics System Toolbox

C/C++ code generation for the following functions requires the Robotics System Toolbox software.

Name	Remarks and Limitations
Algorithm Design	
robotics.PurePursuit	Supports MATLAB Function block: No
robotics.VectorFieldHistogram	Supports MATLAB Function block: No
Coordinate System Transformations	
angdiff	Supports MATLAB Function block: No
axang2quat	Supports MATLAB Function block: No

Name	Remarks and Limitations
axang2rotm	Supports MATLAB Function block: No
axang2tform	Supports MATLAB Function block: No
cart2hom	Supports MATLAB Function block: No
eul2quat	Supports MATLAB Function block: No
eul2rotm	Supports MATLAB Function block: No
eul2tform	Supports MATLAB Function block: No
hom2cart	Supports MATLAB Function block: No
quat2axang	Supports MATLAB Function block: No
quat2eul	Supports MATLAB Function block: No
quat2rotm	Supports MATLAB Function block: No
quat2tform	Supports MATLAB Function block: No
rotm2axang	Supports MATLAB Function block: No
rotm2eul	Supports MATLAB Function block: No
rotm2quat	Supports MATLAB Function block: No
rotm2tform	Supports MATLAB Function block: No
tform2axang	Supports MATLAB Function block: No
tform2eul	Supports MATLAB Function block: No
tform2quat	Supports MATLAB Function block: No
tform2rotm	Supports MATLAB Function block: No
tform2trvec	Supports MATLAB Function block: No
trvec2tform	Supports MATLAB Function block: No

Rounding and Remainder Functions in MATLAB

Function	Remarks and Limitations
ceil	—
fix	—
floor	—

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

Set Operations in MATLAB

Function	Remarks and Limitations
<code>intersect</code>	<ul style="list-style-type: none"> Does not support cell arrays for the first or second arguments. When you do not specify the <code>'rows'</code> option: <ul style="list-style-type: none"> 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. 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. 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. If you specify the <code>'legacy'</code> option, empty outputs are row vectors, 1-by-0, never 0-by-0. 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. When the <code>setOrder</code> is <code>'sorted'</code> or when you specify the <code>'legacy'</code> option, the inputs must already be sorted in ascending order. The first output, <code>C</code>, is sorted in ascending order. Complex inputs must be <code>single</code> or <code>double</code>.

Function	Remarks and Limitations
	<ul style="list-style-type: none"> • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to <code>'stable'</code>. • 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>. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
ismember	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • The second input, <code>B</code>, must be sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
issorted	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.

Function	Remarks and Limitations
<p><code>setdiff</code></p>	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • Inputs A and B must be vectors. If you specify the 'legacy' option, inputs A and B must be row vectors. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' and 'rows' options, the output ia is a column vector. If ia is empty, it is 0-by-1, never 0-by-0, even if the output C is 0-by-0. • 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, C, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • Sort the real input in complex ascending order (by absolute value). Suppose the real input is x. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.

Function	Remarks and Limitations
setxor	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • 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. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • 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, C, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • 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>. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.

Function	Remarks and Limitations
union	<ul style="list-style-type: none"> • Does not support cell arrays for the first or second arguments. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • 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. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'legacy' option and the 'rows' option, the outputs ia and ib are column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output C is 0-by-0. • 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, C, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>. • When one input is complex and the other input is real, do one of the following: <ul style="list-style-type: none"> • Set <code>setOrder</code> to 'stable'. • Sort the real input in complex ascending order (by absolute value). Suppose the real input is x. Use <code>sort(complex(x))</code> or <code>sortrows(complex(x))</code>. • “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.

Function	Remarks and Limitations
unique	<ul style="list-style-type: none"> • Does not support cell arrays for the first argument. • When you do not specify the 'rows' option: <ul style="list-style-type: none"> • The input A must be a vector. If you specify the 'legacy' option, the input A must be a row vector. • 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. • 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. • If you specify the 'legacy' option, empty outputs are row vectors, 1-by-0, never 0-by-0. • When you specify both the 'rows' option and the 'legacy' option, outputs <code>ia</code> and <code>ic</code> are column vectors. If these outputs are empty, they are 0-by-1, even if the output C is 0-by-0. • When the <code>setOrder</code> is 'sorted' or when you specify the 'legacy' option, the input A must already be sorted in ascending order. The first output, C, is sorted in ascending order. • Complex inputs must be <code>single</code> or <code>double</code>.

Signal Processing in MATLAB

Function	Remarks and Limitations
chol	—
conv	—
fft	<ul style="list-style-type: none"> • The length of the input vector must be a power of 2. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
fft2	<ul style="list-style-type: none"> • The length of the input matrix dimensions must each be a power of 2.
fftn	<ul style="list-style-type: none"> • The <code>siz</code> argument must have a fixed size. • The length of the input matrix dimensions must each be a power of 2.
fftshift	—

Function	Remarks and Limitations
filter	<ul style="list-style-type: none"> • If supplied, <code>dim</code> must be a constant. • <code>v</code>
freqspace	—
ifft	<ul style="list-style-type: none"> • The length of the input vector must be a power of 2. • Output of <code>ifft</code> block is complex. • Does not support the 'symmetric' option. • “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
ifft2	<ul style="list-style-type: none"> • The length of the input matrix dimensions must each be a power of 2. • Does not support the 'symmetric' option.
ifftn	<ul style="list-style-type: none"> • Does not support the 'symmetric' option. • The <code>siz</code> argument must have a fixed size. • The length of the input matrix dimensions must each be a power of 2.
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.
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.
buttdord	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.

Function	Remarks/Limitations
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	—
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.

Function	Remarks/Limitations
<code>firls</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>firpm</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>firpmord</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>flattopwin</code>	All inputs must be constants. Expressions or variables are allowed if their values do not change.
<code>freqz</code>	<ul style="list-style-type: none"> • Does not support variable-size inputs. • 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>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>	—

Function	Remarks/Limitations
levinson	C and C++ code generation for <code>levinson</code> requires DSP System Toolbox software. If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.
maxflat	All inputs must be constant. Expressions or variables are allowed if their values do not change.
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.

Function	Remarks/Limitations
upfirdn	<p>C and C++ code generation for <code>upfirdn</code> requires DSP System Toolbox software.</p> <p>Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change.</p> <p>Variable-size inputs are not supported.</p>
upsample	<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<10)</pre>
xcorr	—
yulewalk	<p>If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.</p>

Special Values in MATLAB

Function	Remarks and Limitations
eps	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
inf	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
intmax	—
intmin	—
NaN or nan	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integers.
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	—
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"> Row-vector input is only supported when the first two inputs are vectors and nonscalar.
cummin	—
cummax	—
mean	<ul style="list-style-type: none"> Does not support the 'native' output class option for integer types. If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

Function	Remarks and Limitations
	<ul style="list-style-type: none"> Does not support the <code>nanflag</code> argument.
<code>median</code>	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 Does not support the <code>nanflag</code> argument. “Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4.
<code>mode</code>	<ul style="list-style-type: none"> Does not support third output argument <code>C</code> (cell array). If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35
<code>std</code>	<ul style="list-style-type: none"> “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 Does not support the <code>nanflag</code> argument.
<code>var</code>	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant. “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35 Does not support the <code>nanflag</code> argument.

Statistics and Machine Learning Toolbox

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

Function	Remarks and Limitations
<code>betacdf</code>	—
<code>betafit</code>	—
<code>betainv</code>	—
<code>betalike</code>	—
<code>betapdf</code>	—

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

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

Function	Remarks and Limitations
hygeinv	—
hygepdf	—
hygernd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
hygestat	—
icdf	—
iqr	—
kmeans	<ul style="list-style-type: none"> • If the Start method uses random selections, the initial centroid cluster positions might not match MATLAB. • If the number of rows in X is fixed, does not remove rows of X that contain a NaN. • The cluster centroid locations in C can have a different order than in MATLAB. In this case, the cluster indices in idx have corresponding differences. • If you provide Display, its value must be 'off'. • If you provide Streams, it must be empty and UseSubstreams must be false. • When you set the UseParallel option to true, some computations can execute in parallel even when Replicates is 1. For large data sets, when Replicates is 1, consider setting the UseParallel option to true.
kurtosis	—
logncdf	—
logninv	—
lognpdf	—

Function	Remarks and Limitations
lognrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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	—
nansum	—
nanvar	—
nbincdf	—
nbinv	—
nbincpdf	—
nbincrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
nbinstat	—
ncfcdf	—
ncfinv	—
ncfpdf	—

Function	Remarks and Limitations
ncfrnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
nctstat	—
ncx2cdf	—
ncx2rnd	Can return a different sequence of numbers than MATLAB if either of the following is true: <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
normstat	—

Function	Remarks and Limitations
pca	<ul style="list-style-type: none"> • Ignores the 'Display' value for 'Options' when 'Algorithm' is 'als'. • If supplied, 'Weights' and 'VariableWeights' must be real. • Always returns the fifth output explained as a column vector. • Always returns the sixth output mu as a row vector. • If mu is empty, pca returns mu as a 1-by-0 array. pca does not convert mu to a 0-by-0 empty array. • Does not treat an input matrix X that has all NaN values as a special case. The outputs have the sizes that they have when some of the inputs are finite.
pdf	—
pearsrnd	Matches MATLAB only when generated output r is scalar.
poisscdf	—
poissinv	—
poisspdf	—
poissrnd	<p>Can return a different sequence of numbers than MATLAB if either of the following is true:</p> <ul style="list-style-type: none"> • The output is nonscalar. • An input parameter is invalid for the distribution.
poisstat	—

Function	Remarks and Limitations
prctile	<ul style="list-style-type: none"> • “Automatic dimension restriction” on page 7-35 • 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"> • You do not supply the <code>dim</code> input. • X is a variable-size array. • X is not a variable-length vector. • X is a vector at run time. • The orientation of the vector X does not match the orientation of the vector <code>p</code>. <p>In this case, the output Y matches the orientation of X not the orientation of <code>p</code>.</p>
quantile	—
randg	—
random	—
randsample	When sampling without replacement, the order of the output values might not match MATLAB.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.
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"> • The output is nonscalar. • An input parameter is invalid for the distribution.

Function	Remarks and Limitations
wblstat	—
zscore	—

String Functions in MATLAB

Function	Remarks and Limitations
bin2dec	<ul style="list-style-type: none"> Does not support cell arrays. Does not match MATLAB when the input is empty.
blanks	—
char	Does not support cell arrays.
deblank	<ul style="list-style-type: none"> Supports only inputs from the <code>char</code> class. Does not support cell arrays. Input values must be in the range 0-127.
dec2bin	<ul style="list-style-type: none"> If input <code>d</code> is <code>double</code>, <code>d</code> must be less than 2^{52}. If input <code>d</code> is <code>single</code>, <code>d</code> must be less than 2^{23}. 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.
dec2hex	<ul style="list-style-type: none"> If input <code>d</code> is <code>double</code>, <code>d</code> must be less than 2^{52}. If input <code>d</code> is <code>single</code>, <code>d</code> must be less than 2^{23}. 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.
hex2dec	Does not support cell arrays.
hex2num	<ul style="list-style-type: none"> Does not support cell arrays. For <code>n = hex2num(S)</code>, <code>size(S,2) <= length(num2hex(0))</code>
iscellstr	—
ischar	—

Function	Remarks and Limitations
<code>isletter</code>	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0-127.
<code>isspace</code>	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0–127.
<code>isstrprop</code>	<ul style="list-style-type: none"> Supports only inputs from <code>char</code> and <code>integer</code> classes. Input values must be in the range 0-127.
<code>lower</code>	<ul style="list-style-type: none"> Supports only <code>char</code> inputs. Does not support cell arrays. Input values must be in the range 0-127.
<code>num2hex</code>	—
<code>str2double</code>	<ul style="list-style-type: none"> Does not support cell arrays. Always returns a complex result.
<code>strcmp</code>	<ul style="list-style-type: none"> When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector. When both inputs are empty character arrays that have different sizes, returns true.
<code>strcmpi</code>	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0-127. When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector. When both inputs are empty character arrays that have different sizes, returns true.
<code>strfind</code>	<ul style="list-style-type: none"> Does not support cell arrays. 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>. Inputs must be character row vectors.
<code>strjoin</code>	Always returns a string that has size 1-by-n. For empty inputs, returns a 1-by-0 string not <code>[]</code> .
<code>strjust</code>	Does not support a cell array of strings for the first argument.
<code>strncmp</code>	When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.
<code>strncmpi</code>	<ul style="list-style-type: none"> Input values from the <code>char</code> class must be in the range 0-127. When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.

Function	Remarks and Limitations
strrep	<ul style="list-style-type: none"> • Does not support cell arrays. • If oldSubstr does not exist in origStr, returns blanks(0). To check for an empty return, use isempty. • Inputs must be character row vectors.
strtok	Does not support a cell arrays for the first argument.
strtrim	<ul style="list-style-type: none"> • Supports only inputs from the char class. Does not support cell arrays. • Input values must be in the range 0-127.
upper	<ul style="list-style-type: none"> • Supports only inputs from the char class. Does not support cell arrays. • Input values must be in the range 0-127.

System Identification Toolbox

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

Name	Remarks and Limitations
recursiveAR	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveARMA	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator. • For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveARMAX	<ul style="list-style-type: none"> • For Simulink-based workflows, use Recursive Polynomial Model Estimator.

Name	Remarks and Limitations
	<ul style="list-style-type: none"> For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveARX	<ul style="list-style-type: none"> For Simulink-based workflows, use Recursive Polynomial Model Estimator. For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveBJ	<ul style="list-style-type: none"> For Simulink-based workflows, use Recursive Polynomial Model Estimator. For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveLS	<ul style="list-style-type: none"> For Simulink-based workflows, use Recursive Least Squares Estimator. For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.
recursiveOE	<ul style="list-style-type: none"> For Simulink-based workflows, use Recursive Polynomial Model Estimator. For limitations, see “Generate Code for Online Parameter Estimation in MATLAB”.

Trigonometry in MATLAB

Function	Remarks and Limitations
acos	When the input value x is real, but the output should be complex, generates an error during simulation and returns NaN in generated code. To get the complex result, make the input value complex by passing in <code>complex(x)</code> .

Function	Remarks and Limitations
acosd	—
acosh	<ul style="list-style-type: none"> 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>.
acot	—
acotd	—
acoth	—
acsc	—
acscd	—
acsch	—
asec	—
asecd	—
asech	—
asin	<ul style="list-style-type: none"> 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>.
asind	—
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	—

Function	Remarks and Limitations
cot	—
cotd	<ul style="list-style-type: none">• In some cases, returns -Inf when MATLAB returns Inf.• In some cases, returns Inf when MATLAB returns -Inf.
coth	—
csc	—
cscd	<ul style="list-style-type: none">• In some cases, returns -Inf when MATLAB returns Inf.• In some cases, returns Inf when MATLAB returns -Inf.
csch	—
hypot	—
sec	—
secd	<ul style="list-style-type: none">• In some cases, returns -Inf when MATLAB returns Inf.• In some cases, returns Inf when MATLAB returns -Inf.
sech	—
sin	—
sind	—
sinh	—
tan	—
tand	<ul style="list-style-type: none">• In some cases, returns -Inf when MATLAB returns Inf.• In some cases, returns Inf when MATLAB returns -Inf.
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” on page 8-2.

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 7-31.

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” on page 22-18).

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[®] 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"> The loop counter increments by 1 The end value equals the maximum value of the integer type The loop is not covering the full range of the integer type 	<p>Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace:</p> <pre>N=intmax('int16') for k=N-10:N</pre> <p>with:</p> <pre>for k=1:10</pre>
<ul style="list-style-type: none"> The loop counter decrements by 1 The end value equals the minimum value of the integer type The loop is not covering the full range of the integer type 	<p>Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace:</p> <pre>N=intmin('int32') for k=N+10:-1:N</pre> <p>with:</p> <pre>for k=10:-1:1</pre>
<ul style="list-style-type: none"> The loop counter increments or decrements by 1 The start value equals the minimum or maximum value of the integer type The end value equals the maximum or minimum value of the integer type <p>The loop covers the full range of the integer type.</p>	<p>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:</p> <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">• The loop counter increments or decrements by a value not equal to 1• On last loop iteration, the loop variable value is not equal to the end value	end
Note: 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 and Strings” on page 6-9
- “Array Size Restrictions for Code Generation” on page 6-10
- “Code Generation for Constants in Structures and Arrays” on page 6-11

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 6-10
Complex numbers	<ul style="list-style-type: none"> Complexity of variables must be set at time of assignment and before first use Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero <p>Note: 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 and Strings” on page 6-9
Enumerated data	<ul style="list-style-type: none"> Supports integer-based enumerated types only Restricted use in <code>switch</code> statements and <code>for</code>-loops 	“Enumerated Data”
Function handles	<ul style="list-style-type: none"> Same bound variable cannot reference 	“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 data-bbox="609 513 911 604">• Cannot view function handles from the debugger	

Code Generation for Complex Data

In this section...

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

“Code Generation for Complex Data with Zero-Valued Imaginary Parts” on page 6-4

“Results of Expressions That Have Complex Operands” on page 6-8

Restrictions When Defining Complex Variables

For code generation, you must set the complexity of variables at the time of assignment. Assign a complex constant to the variable or use the `complex` function. For example:

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

After assignment, you cannot change the complexity of a variable. Code generation for the following function fails because $x(k) = 3 + 4i$ changes the complexity of x .

```
function x = test1( )  
x = zeros(3,3); % x is real  
for k = 1:numel(x)  
    x(k) = 3 + 4i;  
end  
end
```

To resolve this issue, assign a complex constant to x .

```
function x = test1( )  
x = zeros(3,3)+ 0i; %x is complex  
for k = 1:numel(x)  
    x(k) = 3 + 4i;  
end  
end
```

Code Generation for Complex Data with Zero-Valued Imaginary Parts

For code generation, complex data that has all zero-valued imaginary parts remains complex. This data does not become real. This behavior has the following implications:

- In some cases, results from functions that sort complex data by absolute value can differ from the MATLAB results. See “Functions That Sort Complex Values by Absolute Value” on page 6-5.
- For functions that require that complex inputs are sorted by absolute value, complex inputs with zero-valued imaginary parts must be sorted by absolute value. These functions include `ismember`, `union`, `intersect`, `setdiff`, and `setxor`.

Functions That Sort Complex Values by Absolute Value

Functions that sort complex values by absolute value include `sort`, `issorted`, `sortrows`, `median`, `min`, and `max`. These functions sort complex numbers by absolute value even when the imaginary parts are zero. In general, sorting the absolute values produces a different result than sorting the real parts. Therefore, when inputs to these functions are complex with zero-valued imaginary parts in generated code, but real in MATLAB, the generated code can produce different results than MATLAB. In the following examples, the input to `sort` is real in MATLAB, but complex with zero-valued imaginary parts in the generated code:

- **You Pass Real Inputs to a Function Generated for Complex Inputs**

- 1 Write this function:

```
function myout = mysort(A)
myout = sort(A);
end
```

- 2 Call `mysort` in MATLAB.

```
A = -2:2;
mysort(A)

ans =

    -2    -1     0     1     2
```

- 3 Generate a MEX function for complex inputs.

```
A = -2:2;
codegen mysort -args {complex(A)} -report
```

- 4 Call the MEX Function with real inputs.

```
mysort_mex(A)

ans =
```

```
0    1   -1    2   -2
```

You generated the MEX function for complex inputs, therefore, it treats the real inputs as complex numbers with zero-valued imaginary parts. It sorts the numbers by the absolute values of the complex numbers. Because the imaginary parts are zero, the MEX function returns the results to the MATLAB workspace as real numbers. See “Inputs and Outputs for MEX Functions Generated for Complex Arguments” on page 6-7.

- **Input to `sort` Is Output from a Function That Returns Complex in Generated Code**

- 1 Write this function:

```
function y = myfun(A)
x = eig(A);
y = sort(x, 'descend');
```

The output from `eig` is the input to `sort`. In generated code, `eig` returns a complex result. Therefore, in the generated code, `x` is complex.

- 2 Call `myfun` in MATLAB.

```
A = [2 3 5;0 5 5;6 7 4];
myfun(A)
```

```
ans =
```

```
12.5777
 2.0000
-3.5777
```

The result of `eig` is real. Therefore, the inputs to `sort` are real.

- 3 Generate a MEX function for complex inputs.

```
codegen myfun -args {complex(A)}
```

- 4 Call the MEX function.

```
myfun_mex(A)
```

```
ans =
```

```
12.5777
-3.5777
 2.0000
```


In the MEX function, `eig` returns a complex result. Therefore, the inputs to `sort` are complex. The MEX function sorts the inputs in descending order of the absolute values.

Inputs and Outputs for MEX Functions Generated for Complex Arguments

For MEX functions created by MATLAB Coder:

- Suppose that you generate the MEX function for complex inputs. If you call the MEX function with real inputs, the MEX function transforms the real inputs to complex values with zero-valued imaginary parts.
- If the MEX function returns complex values that have all zero-valued imaginary parts, the MEX function returns the values to the MATLAB workspace as real values. For example, consider this function:

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

If you generate a MEX function for `foo` and view the code generation report, you see that `y` is complex.

```
codegen foo -report
```

Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	Yes

If you run the MEX function, you see that in the MATLAB workspace, the result of `foo_mex` is the real value 1.

```
z = foo_mex

ans =

    1
```

Results of Expressions That Have Complex Operands

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 line of code:

```
z = x + y;
```

Suppose that at run time, x has the value $2 + 3i$ and y has the value $2 - 3i$. In MATLAB, this code produces the real result $z = 4$. During code generation, the types for x and y are known, but their values are not known. Because either or both operands in this expression are complex, z is defined as a complex variable requiring storage for 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:

- When the imaginary parts of complex results are zero, MEX functions return the results to the MATLAB workspace as real values. See “Inputs and Outputs for MEX Functions Generated for Complex Arguments” on page 6-7.
- When the imaginary part of the argument is zero, complex arguments to extrinsic functions are real.

```
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 and Strings

The code generation software translates the 16-bit Unicode[®] encoding of a character in MATLAB to an 8-bit encoding that the locale setting determines. The code generation software does not support characters that require more than 1 byte in MATLAB. This restriction applies to constant strings and strings that are passed between MATLAB and the code generation software. For example, the restriction applies to entry-point function inputs and to outputs from extrinsic calls. For code generation, some MATLAB functions accept only 7-bit ASCII strings. See “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List” on page 4-2.

If a character is not in the 7-bit ASCII codeset, casting the character to a numeric type, such as double, produces a different result in the generated code than in MATLAB. A best practice for code generation is to avoid performing arithmetic with characters.

More About

- “Locale Settings for MATLAB Process”

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 Constants in Structures and Arrays

The code generation software does not recognize constant structure fields or array elements in the following cases:

Fields or elements are assigned inside control constructs

In the following code, the code generation software recognizes that the structure fields `s.a` and `s.b` are constants.

```
function y = mystruct()
s.a = 3;
s.b = 5;
y = zeros(s.a,s.b);
```

If any structure field is assigned inside a control construct, the code generation software does not recognize the constant fields. This limitation also applies to arrays with constant elements. Consider the following code:

```
function y = mystruct(x)
s.a = 3;
if x > 1
    s.b = 4;
else
    s.b = 5;
end
y = zeros(s.a,s.b);
```

The code generation software does not recognize that `s.a` and `s.b` are constant. If variable-sizing is enabled, `y` is treated as a variable-size array. If variable-sizing is disabled, the code generation software reports an error.

Constants are assigned to array elements using non-scalar indexing

In the following code, the code generation software recognizes that `a(1)` is constant.

```
function y = myarray()
a = zeros(1,3);
a(1) = 20;
y = coder.const(a(1));
```

In the following code, because `a(1)` is assigned using non-scalar indexing, the code generation software does not recognize that `a(1)` is constant.

```
function y = myarray()  
a = zeros(1,3);  
a(1:2) = 20;  
y = coder.const(a(1));
```

A function returns a structure or array that has constant and nonconstant elements

For an output structure that has both constant and nonconstant fields, the code generation software does not recognize the constant fields. This limitation also applies to arrays that have constant and nonconstant elements. Consider the following code:

```
function y = mystruct_out(x)  
s = create_structure(x);  
y = coder.const(s.a);  
  
function s = create_structure(x)  
s.a = 10;  
s.b = x;
```

Because `create_structure` returns a structure `s` that has one constant field and one nonconstant field, the code generation software does not recognize that `s.a` is constant. The `coder.const` call fails because `s.a` is not constant.

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 7-22
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 7-26
- “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 7-35

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” on page 21-107.

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” on page 21-110. 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 7-24

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

The second argument of `coder.versize` 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.versize` assumes dimensions of 1 are fixed size. For more information, see the `coder.versize` 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"> • <code>ones</code> • <code>zeros</code> • <code>repmat</code> 	“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 7-20

C Code Interface for Statically Allocated Arrays

For statically allocated arrays, the generated code contains the definition of the array and the size of the array.

For example, consider the MATLAB function `myunique_tol`.

```
function B = myunique_tol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 100], [0 1]);
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 statement `coder.varsize('B', [1 100], [0 1])` specifies that `B` is a variable-size array whose first dimension is fixed at 1 and second dimension can vary up to 100 elements. Without this statement, `B` is a dynamically allocated array.

Generate code for `myunique_tol` 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 myunique_tol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

In the generated code, the function declaration is:

```
extern void myunique_tol(const double A_data[], const int A_size[2], double tol,
    double B_data[], int B_size[2])
```

The function signature declares the input argument `A` and the output argument `B`. `A_size` contains the size of `A`. `B_size` contains the size of `B` after the call to `myunique1`. Use `B_size` to determine the number of elements of `B` that you can access after the call to `myunique1`. `B_size[0]` contains the size of the first dimension. `B_size[1]` contains the size of the second dimension. Therefore, the number of elements of `B` is `B_size[0]*B_size[1]`. Even though `B` has 100 elements in the C code, only `B_size[0]*B_size[1]` elements contain valid data.

The following C main function shows how to call `myunique1`.

```
void main()
{
    double A[100], B[100];
    int A_size[2] = { 1, 100 };
    int B_size[2];
    int i;
    for (i = 0; i < 100; i++) {
        A[i] = (double)1/i;
    }
    myunique1(A, A_size, 0.1, B, B_size);
}
```

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>;
```

`baseTypedef` is the predefined type in `rtwtypes.h` corresponding to `baseType`. For example, here is 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;
```

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” on page 27-9.

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><baseType></code> .
<code>*size</code>	Pointer to first element of size vector. Length of the vector equals the number of dimensions.
<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"> <code>true</code> – MATLAB deallocates memory automatically <code>false</code> – Calling program determines when to deallocate memory

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” on page 21-116.

Note: The code generation software exports `emxArray` utility functions only for variable-size arrays that are entry-point function arguments or that are used by functions called by `coder.ceval`.

Function	Arguments	Description
<code>emxArray_<baseType></code> <code>*emxCreateWrapper_<baseType></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 <code>false</code> so it does not inadvertently free user memory, such as the stack.
<code>emxArray_<baseType></code> <code>*emxCreateWrapperND_<baseType></code> <code>(...)</code>	<code>*data</code> <code>numDimensions</code> <code>*size</code>	Same as <code>emxCreateWrapper_<baseType></code> , except it creates a new N-dimensional <code>emxArray</code> .
<code>emxArray_<baseType></code> <code>*emxCreate_<baseType> (...)</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

Function	Arguments	Description
		data type specified by <code><baseType></code> .
<code>emxArray_<baseType></code> <code>*emxCreateND_<baseType> (...)</code>	<code>numDimensions</code> <code>*size</code>	Same as <code>emxCreate_<baseType></code> , except it creates a new N-dimensional <code>emxArray</code> on the heap.
<code>void emxInitArray_<baseType></code> <code>(...)</code>	<code>**emxArray</code> <code>numDimensions</code>	Creates a new empty <code>emxArray</code> on the heap. All data elements have the data type specified by <code><baseType></code> .
<code>void emxInitArray_<structType></code> <code>(...)</code>	<code>*structure</code>	Creates empty <code>emxArrays</code> in a structure.
<code>void emxDestroyArray_<baseType></code> <code>(...)</code>	<code>*emxArray</code>	Frees dynamic memory allocated by <code>emxCreate_<baseType></code> , <code>emxCreateND_<baseType></code> , and <code>emxInitArray_<baseType></code> functions.
<code>void emxDestroyArray_<structType></code> <code>(...)</code>	<code>*structure</code>	Frees dynamic memory allocated by <code>emxInitArray_<structType></code> functions.

By default, when you generate C/C++ source code, static libraries, dynamic libraries, and executables, MATLAB Coder generates an example C/C++ main function. The example main function is a template that can help you to incorporate generated C/C++ code into your application. If you generate code that uses dynamically allocated data, the example main function includes calls to `emxArray` utility functions that create `emxArrays` required for this data. The example main function also initializes `emxArray` data to zero values. For more information, see “Incorporate Generated Code Using an Example Main Function” on page 25-18.

Diagnose and Fix Variable-Size Data Errors

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 7-22

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 7-24

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 $Y = \text{zeros}(1,0)$ so it matches the `coder.ysize` 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 7-26

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 7-28

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

“Incompatibility with MATLAB in Determining Class of Empty Arrays” on page 7-30

“Incompatibility with MATLAB in Vector-Vector Indexing” on page 7-31

“Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-31

“Incompatibility with MATLAB in Concatenating Variable-Size Matrices” on page 7-32

“Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements” on page 7-33

“Dynamic Memory Allocation Not Supported for MATLAB Function Blocks” on page 7-34

Incompatibility with MATLAB for Scalar Expansion

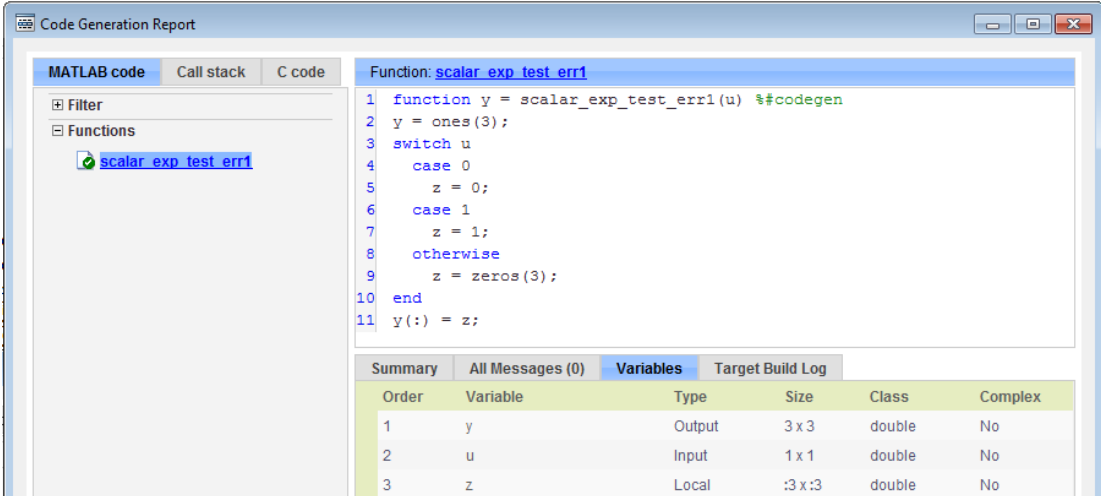
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size 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.



The screenshot shows the 'Code Generation Report' window for the function `scalar_exp_test_err1`. The MATLAB code is displayed in the main pane, and the 'Variables' tab in the summary table is selected.

Order	Variable	Type	Size	Class	Complex
1	y	Output	3 x 3	double	No
2	u	Input	1 x 1	double	No
3	z	Local	:3 x:3	double	No

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 this 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:

```
...
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-size arrays, the dimensions that are not being concatenated must match exactly.

Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements

Suppose that:

- `c` is a variable-size cell array.
- You access the contents of `c` by using curly braces. For example, `c{2:4}`.
- You include the results in concatenation. For example, `[a c{2:4} b]`.
- `c{I}` returns no elements. Either `c` is empty or the indexing inside the curly braces produces an empty result.

For these conditions, MATLAB omits `c{I}` from the concatenation. For example, `[a c{I} b]` becomes `[a b]`. The code generation software treats `c{I}` as the empty array `[c{I}]`. The concatenation becomes `[... [c{i}] ...]`. This concatenation then omits the array `[c{I}]`. So that the properties of `[c{I}]` are compatible with the concatenation `[... [c{i}] ...]`, the code generation software assigns the class, size, and complexity of `[c{I}]` according to these rules:

- The class and complexity are the same as the base type of the cell array.
- The size of the second dimension is always 0.
- For the rest of the dimensions, the size of `Ni` depends on whether the corresponding dimension in the base type is fixed or variable size.
 - If the corresponding dimension in the base type is variable size, the dimension has size 0 in the result.
 - If the corresponding dimension in the base type is fixed size, the dimension has that size in the result.

Suppose that `c` has a base type with class `int8` and size `10x7x8x?`. In the generated code, the class of `[c{I}]` is `int8`. The size of `[c{I}]` is `0x0x8x0`. The second dimension is 0. The first and last dimensions are 0 because those dimensions are variable size in the base type. The third dimension is 8 because the size of the third dimension of the base type is a fixed size 8.

Inside concatenation, if curly-brace indexing of a variable-size cell array returns no elements, the generated code can have the following differences from MATLAB:

- The class of `[...c{i}...]` in the generated code can differ from the class in MATLAB.

When `c{I}` returns no elements, MATLAB removes `c{I}` from the concatenation. Therefore, `c{I}` does not affect the class of the result. MATLAB determines the class of the result based on the classes of the remaining arrays, according to a precedence of classes. See “Valid Combinations of Unlike Classes”. In the generated code, the class of `[c{I}]` affects the class of the result of the overall concatenation `[... [c{I}]...]` because the code generation software treats `c{I}` as `[c{I}]`. The previously described rules determine the class of `[c{I}]`.

- In the generated code, the size of `[c{I}]` can differ from the size in MATLAB.

In MATLAB, the concatenation `[c{I}]` is a 0x0 double. In the generated code, the previously described rules determine the size of `[c{I}]`.

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 7-35

“Toolbox Functions with Variable Sizing Restrictions” on page 7-36

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 7-36.

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"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
any	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
bsxfun	<ul style="list-style-type: none"> • Dimensions expand only where one input array or the other has a fixed length of 1.
cat	<ul style="list-style-type: none"> • Dimension argument must be a constant. • An error occurs if variable-size inputs are empty at run time.
conv	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-35. • Input vectors must have the same orientation, either both row vectors or both column vectors.
cov	<ul style="list-style-type: none"> • For <code>cov(X)</code>, see “Array-to-vector restriction” on page 7-35.
cross	<ul style="list-style-type: none"> • Variable-size array inputs that become vectors at run time must have the same orientation.
deconv	<ul style="list-style-type: none"> • For both arguments, see “Variable-length vector restriction” on page 7-35.
detrend	<ul style="list-style-type: none"> • For first argument for row vectors only, see “Array-to-vector restriction” on page 7-35 .
diag	<ul style="list-style-type: none"> • See “Array-to-vector restriction” on page 7-35 .
diff	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35.

Function	Restrictions with Variable-Size Data
	<ul style="list-style-type: none"> 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.
fft	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 7-35.
filter	<ul style="list-style-type: none"> For first and second arguments, see “Variable-length vector restriction” on page 7-35. See “Automatic dimension restriction” on page 7-35.
hist	<ul style="list-style-type: none"> For second argument, see “Variable-length vector restriction” on page 7-35. For second input argument, see “Array-to-scalar restriction” on page 7-36.
histc	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 7-35.
ifft	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 7-35.
ind2sub	<ul style="list-style-type: none"> First input (the size vector input) must be fixed size.
interp1	<ul style="list-style-type: none"> For the Y input and xi input, see “Array-to-vector restriction” on page 7-35. Y input can become a column vector dynamically. A run-time error occurs if Y input is not a variable-length vector and becomes a row vector at run time.
ipermute	<ul style="list-style-type: none"> Order input must be fixed size.
issorted	<ul style="list-style-type: none"> For optional rows input, see “Variable-length vector restriction” on page 7-35.
magic	<ul style="list-style-type: none"> Argument must be a constant. Output can be fixed-size matrices only.
max	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 7-35.
mean	<ul style="list-style-type: none"> See “Automatic dimension restriction” on page 7-35. An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
median	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
min	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35.
mode	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
mtimes	<ul style="list-style-type: none"> • 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.
nchoosek	<ul style="list-style-type: none"> • The second input, k, must be a fixed-size scalar. • The second input, k, must be a constant for static allocation. If you enable dynamic allocation, the second input can be a variable. • You cannot create a variable-size array by passing in a variable, k, unless you enable dynamic allocation.
permute	<ul style="list-style-type: none"> • Order input must be fixed-size.
planerot	<ul style="list-style-type: none"> • 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.
poly	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-35.
polyfit	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 7-35.
prod	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.

Function	Restrictions with Variable-Size Data
rand	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>rand(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>rand([1 N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
randi	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>randi(imax,1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>randi(imax,[1 N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
randn	<ul style="list-style-type: none"> • For an upper-bounded variable N, <code>randn(1,N)</code> produces a variable-length vector of $1 \times M$ where M is the upper bound on N. • For an upper-bounded variable N, <code>randn([1 N])</code> may produce a variable-length vector of $:1 \times M$ where M is the upper bound on N.
reshape	<ul style="list-style-type: none"> • 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. • 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.
roots	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-35.

Function	Restrictions with Variable-Size Data
shiftdim	<ul style="list-style-type: none"> • 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. • 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). • First input argument must have the same number of dimensions when you supply a positive number of shifts.
std	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
sub2ind	<ul style="list-style-type: none"> • First input (the size vector input) must be fixed size.
sum	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
trapz	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
typecast	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 7-35 on first argument.
var	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 7-35. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

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 8-6
- “Make Structures Persistent” on page 8-8
- “Index Substructures and Fields” on page 8-9
- “Assign Values to Structures and Fields” on page 8-11
- “Pass Structure Arguments by Reference or by Value” on page 8-13

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:

What's Different	More Information
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 8-6
Reference structure fields individually during indexing.	“Index Substructures and Fields” on page 8-9
Avoid type mismatch when assigning values to structures and fields.	“Assign Values to Structures and Fields” on page 8-11

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...

“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

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 8-6

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 8-6

“Defining an Array of Structures by Using struct” on page 8-7

“Defining an Array of Structures Using Concatenation” on page 8-7

Ensuring Consistency of Fields

For code generation, when you create an array of MATLAB structures, corresponding fields in the array elements must have 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 by Using `struct`

To create an array of structures using the `struct` function, specify the field value arguments as cell arrays. Each cell array element is the value of the field in the corresponding structure array element. For code generation, corresponding fields in the structures must have the same type. Therefore, the elements in a cell array of field values must have the same type.

For example, the following code creates a 1-by-3 structure array. For each structure in the array of structures, `a` has type `double` and `b` has type `char`.

```
s = struct('a', {1 2 3}, 'b', {'a' 'b' 'c'});
```

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 8-6 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

When assigning values to a structure, substructure, or field for code generation, use these guidelines:

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.

For structures with constant fields, do not assign field values inside control flow constructs

In the following code, the code generation software recognizes that the structure fields `s.a` and `s.b` are constants.

```
function y = mystruct()
s.a = 3;
s.b = 5;
y = zeros(s.a,s.b);
```

If a field of a structure is assigned inside a control flow construct, the code generation software does not recognize that `s.a` and `s.b` are constant. Consider the following code:

```
function y = mystruct(x)
s.a = 3;
if x > 1
    s.b = 4;
else
    s.b = 5;
end
y = zeros(s.a,s.b);
```

If variable-sizing is enabled, `y` is treated as a variable-size array. If variable-sizing is disabled, `y`, the code generation software reports an error.

Do not assign mxArray to structures

You cannot assign mxArray to structure elements; convert mxArray to known types before code generation (see “Working with mxArray” on page 14-17).

Do not assign classes to structure fields

You cannot assign classes to structure fields.

Do not assign cell arrays to structure fields

You cannot assign cell arrays to structure fields.

Pass Structure Arguments by Reference or by Value

In this section...

“Specify Pass by Reference or by Value Using the MATLAB Coder App” on page 8-13

“Specify Pass by Reference or by Value Using the Command-Line Interface” on page 8-14

“Pass Input Structure Argument by Reference” on page 8-14

“Pass Input Structure Argument by Value” on page 8-15

“Pass Output Structure Argument by Reference” on page 8-16

“Pass Output Structure Argument by Value” on page 8-16


“Pass Input and Output Structure Argument by Reference” on page 8-17

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 the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to one of the following:
 - Source Code
 - Static Library
 - Dynamic Library
 - Executable
- 3 Click **More Settings**.

- 4 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 (default)
false	Pass by value

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;
```

- 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;
```

- 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;
```

- 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 Cell Arrays

- “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2
- “Control Whether a Cell Array is Homogeneous or Heterogeneous” on page 9-4
- “Define Cell Array Inputs” on page 9-5
- “Make a Cell Array Variable Size” on page 9-6
- “Name the Structure Type That Represents a Cell Array” on page 9-8
- “Cell Array Requirements and Limitations for Code Generation” on page 9-10
- “Cell Arrays in Code Generation Reports” on page 9-13

Homogeneous vs. Heterogeneous Cell Arrays

The code generation software classifies a cell array as *homogeneous* or *heterogeneous*. This classification determines how a cell array is represented in the generated C/C++ code. It also determines how you can use the cell array in MATLAB code from which you generate C/C++ code.

A homogeneous cell array has the following characteristics:

- The cell array is represented as a C array in the generated code.
- All elements have the same properties. The type associated with the cell array specifies the properties of all elements rather than the properties of individual elements.
- The cell array can be variable size.
- You can index into the cell array using an index whose value is determined at run time.

A heterogeneous cell array has the following characteristics:

- The cell array is represented as a C structure in the generated code. Each element is represented as a field of the structure.
- The elements can have different properties. The type associated with the cell array specifies the properties of each element individually.
- The cell array cannot be variable size.
- You must index into the cell array using a constant index or using `for`-loops with constant bounds.

The code generation software uses heuristics to determine the classification of a cell array as homogeneous or heterogeneous. It considers the properties (class, size, complexity) of the elements and other factors such as how you use the cell array in your program. A particular cell array can be homogeneous in one case and heterogeneous in another case. For example, consider the cell array `{1 [2 3]}`. This cell array can be a heterogeneous 1x2 cell array whose first element is double scalar and whose second element is a 1x2 array of doubles. However, if you index into this cell array with an index whose value is determined at run time, the cell array is a homogeneous cell array. The elements are variable-size arrays of doubles with an upper bound of 2.

To see whether a cell array is homogeneous or heterogeneous, view the variable in the code generation report. See “Cell Arrays in Code Generation Reports” on page 9-13.

For more information, see “Control Whether a Cell Array is Homogeneous or Heterogeneous” on page 9-4.

More About

- “Cell Array Requirements and Limitations for Code Generation” on page 9-10

Control Whether a Cell Array is Homogeneous or Heterogeneous

As long as you do not specify conflicting requirements, you can control whether a cell array is homogeneous or heterogeneous.

If a cell array is an entry-point function input:

- At the command line, use the `coder.typeof` function or the `coder.newtype` function to create a `coder.CellType` object. To create homogeneous or heterogeneous cell array types, use `coder.CellType` methods.
- In the MATLAB Coder app, select **Homogeneous cell array** or **Heterogeneous cell array** as the type of the input.

If a cell array is not an entry-point function input:

- If you use `coder.ysize` with the cell array, the cell array is homogeneous.
- If you index the cell array with an index whose value is determined at run time, the cell array is homogeneous.
- If you use `coder.cstructname` with the cell array, the cell array is heterogeneous.
- If the elements have different classes, the cell array is heterogeneous.

If the code generation software detects conflicting requirements for a cell array, the code generation fails. For example, you cannot use `coder.ysize` with a cell array whose elements have different classes.

See Also

`coder.CellType` | `coder.cstructname` | `coder.ysize`

Related Examples

- “Define Cell Array Inputs” on page 9-5
- “Make a Cell Array Variable Size” on page 9-6
- “Name the Structure Type That Represents a Cell Array” on page 9-8

More About

- “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2

Define Cell Array Inputs

To define entry-point cell array inputs	See
At the command line	“Specify Cell Array Inputs at the Command Line” on page 21-56
Programmatically in the MATLAB file	“Define Input Properties Programmatically in the MATLAB File” on page 21-67
Using the MATLAB Coder app	“Automatically Define Input Types Using the App” on page 18-4 “Define Input Parameters by Example Using the App” on page 18-5 “Define or Edit Input Parameter Type Using the App” on page 18-12

Make a Cell Array Variable Size

This example shows how to specify that a cell array that is not an entry-point input is variable size. To specify a variable-size cell array input, see “Define Cell Array Inputs” on page 9-5.

Define a function `mycellfun` that defines the cell array `c`.

```
function y = mycellfun()
c = {1, [2 3]};
y = c;
end
```

Generate code for `mycellfun`. For example, generate a static library.

```
codegen -config:lib mycellfun -report
```

To see whether the code generation software classifies `c` as homogeneous or heterogeneous:

- 1 Click the **View report** link.
- 2 Click the **MATLAB code** tab. Under **Functions**, click `mycellfun`.
- 3 On the **Variables** tab, expand the information for `c`. The report specifies the properties for `c{1}` and `c{2}` individually, indicating that `c` is heterogeneous.

Summary	All Messages (0)	Variables	Target Build Log			
Order	Variable	Type	Size	Class	Complex	
1	<code>y</code>	Output	1 x 2	cell	-	
2	<code>c</code>	Local	1 x 2	cell	-	
2.1	<code>c{1}</code>	Element	1 x 1	double	No	
2.2	<code>c{2}</code>	Element	1 x 2	double	No	

To specify that `c` is variable size with an upper bound of 10 for the second dimension, add the `coder. varsize` call.

```
function y = mycellfun()
```

```
c = {1, [2 3]};
coder.varsize('c', [1 10]);
y = c;
end
```

Generate the code again.

View **c** in the report. **c** is a variable-size cell array with an upper bound of 10. The report indicates that **c** is homogeneous by using the notation `{:}`. This notation indicates that all elements of the cell array have the same properties. To make **c** homogeneous, the code generation software uses a size `1x:2` that can apply to all of the elements.

Summary	All Messages (0)	Variables	Target Build Log			
Order	Variable	Type	Size	Class	Complex	
1	y	Output	1 x 2 *	cell	-	
2	c	Local	1 x :10	cell	-	
2.1	c{:}	Element	1 x :2	double	No	

See Also

`coder.varsize`

Related Examples

- “Name the Structure Type That Represents a Cell Array” on page 9-8

More About

- “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2
- “Cell Arrays in Code Generation Reports” on page 9-13
- “Cell Array Requirements and Limitations for Code Generation” on page 9-10

Name the Structure Type That Represents a Cell Array

This example shows how to name the structure type that represents a cell array in the generated code. If a cell array is an entry-point function input, to name the structure that represents the cell array, see `coder.cstructname` and “Define Cell Array Inputs” on page 9-5.

Define a function `mycellfun` that defines the cell array `c`. To name the structure type that represents `c` in the generated code, use `coder.cstructname`.

```
function y = mycellfun()
c = {1, [2 3]};
coder.cstructname(c, 'myname');
y = c;
end
```

Generate a static library for `mycellfun`.

```
codegen -config:lib mycellfun -report
```

To see the types that the code generation software specified for `c`:

- 1 Click the **View report** link.
- 2 Click the **MATLAB code** tab. Under **Functions**, click `mycellfun`.
- 3 On the **Variables** tab, expand the information for `c`, `c` is a 1x2 heterogeneous cell array.

Summary	All Messages (0)	Variables	Target Build Log				
Order	Variable	Type	Size	Class	Complex		
1	y	Output	1 x 2	cell	-		
2	c	Local	1 x 2	cell	-		
2.1	c{1}	Element	1 x 1	double	No		
2.2	c{2}	Element	1 x 2	double	No		

To see the types generated for `c` in the C code:

- 1 On the **C code** tab, under **Target Source Files**, click `mycellfun.c`.
- 2 Under **Target Source Files**, click `mycellfun_types.h`.

The definition for structure type `myname` is:

```
typedef struct {  
    double f1;  
    double f2[2];  
} myname;
```

See Also

`coder.cstructname`

Related Examples

- “Make a Cell Array Variable Size” on page 9-6

More About

- “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2

Cell Array Requirements and Limitations for Code Generation

Cell Array Definition

- You must completely assign cell arrays at compile time. You must assign each cell array element on all execution paths.

If you use `cell` to create a cell array, you must assign values to all elements of the cell array. The following code is not allowed because `y{2}` is not assigned.

```
function y = foo()
y = cell(1,3);
y{1} = 1;
y{3} = 3;
end
```

Even if the element is not used, you must assign a value so that the code generation software can determine whether the cell array is homogeneous or heterogeneous. Assign a value that is consistent with how you plan to use the cell array. See “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2.

In the preceding example, if `y{2}` is double, `y` is homogeneous. If `y{2}` is empty, its size is `0x0` not `1x1` like the other elements. In this case, `y` is heterogeneous.

- The assignment must be consistent on all execution paths. The following code is not allowed because `y{2}` is double on one execution path and char on the other execution path.

```
function y = foo(n)
y = cell(1,3)
if n > 1;
    y{1} = 1
    y{2} = 2;
    y{3} = 3;
else
    y{1} = 10;
    y{2} = 'a';
    y{3} = 30;
end
```

Cell Array Indexing

- You cannot index cell arrays by using smooth parentheses (). Instead, consider indexing cell arrays by using curly braces { } to access the contents of the cell.
- You must index into heterogeneous cell arrays by using constant indexes or by using for-loops with constant bounds.

For example, the following code is not allowed:

```
x = {1, 'mystring'};  
disp(x{randi});
```

If the index A and B are constant, the following code is allowed:

```
x = {1, 'mystring'};  
for i = A:B  
    disp(x{i});
```

When the for-loop has constant bounds, it is unrolled. For large cell arrays, the unrolling can increase compile time and generate inefficient code.

Variable-Size Cell Arrays

- Heterogeneous cell arrays cannot be variable size. See “Make a Cell Array Variable Size” on page 9-6.
- You cannot use the cell function to create a variable-size cell array. For example, the following code is not allowed:

```
function y = foo(n)  
y = cell(1,n);
```

Instead, use repmat. For example:

```
function y = foo(n)  
y = repmat({1}, [1 n]);
```

- If you use the cell function to create a cell array, you cannot use coder. varsize with that cell array.

Cell Array Contents

- Cell arrays cannot contain `mxarrays`. In a cell array, you cannot store a value that an extrinsic function returns.
- Cell arrays cannot contain classes or `System` objects.
- Cell arrays cannot contain function handles.

Cell Arrays in Structures

Structures cannot contain cell arrays.

Passing to External C/C++ Functions

You cannot pass a cell array to `coder.ceval`. If a variable is an input argument to `coder.ceval`, define the variable as an array or structure instead of as a cell array.

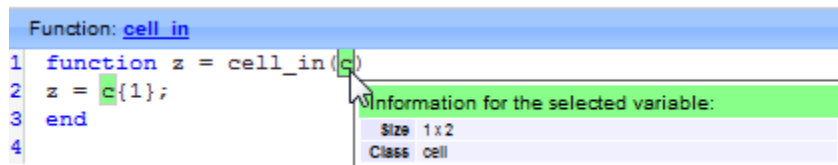
More About

- “Differences in Behavior After Compiling MATLAB Code” on page 2-8

Cell Arrays in Code Generation Reports

Cell Array Variable in the MATLAB Code Pane

When you place your cursor over a cell array variable in the **MATLAB code** pane, the report shows that the variable has class `cell`.



If the cell array has all constant elements, or some constant and some nonconstant elements, the variable name is orange. When you place your cursor over the variable, the report shows the values of the elements. The report displays a nonconstant element as an empty array. If you export the cell array variable to the base workspace, a nonconstant element is an empty array in the base workspace. See “Viewing Variables in Your MATLAB Code” on page 22-18.

Cell Array Variable on the Variables Tab

On the **Variables** tab, for a cell array, the report shows class `cell`. By default, the report collapses the list of cell array elements. To see the properties of the elements, expand the list.

For a homogeneous cell array, the report has one entry that specifies the properties of all elements. The notation `{:}` indicates that all elements of the cell array have the same properties.

Summary	All Messages (0)	Variables	Target Build Log		
Order	Variable	Type	Size	Class	Complex
1	z	Output	1 x 1	double	No
2	c	Input	1 x 3	cell	-
2.1	c{:}	Element	1 x 1	double	No

For a heterogeneous cell array, the report has an entry for each element. For example, for a heterogeneous cell array `c` with two elements, the entry for `c{1}` shows the properties for the first element. The entry for `c{2}` shows the properties for the second element.

Summary	All Messages (0)	Variables	Target Build Log			
Order	Variable	Type	Size	Class	Complex	
1	<code>z</code>	Output	1 x 1	char	-	
2	<code>c</code>	Input	1 x 2	cell	-	
2.1	<code>c{1}</code>	Element	1 x 1	char	-	
2.2	<code>c{2}</code>	Element	1 x 1	double	No	

More About

- “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2
- “Code Generation Reports” on page 22-10

Code Generation for Enumerated Data

- “Enumerated Data Definition for Code Generation” on page 10-2
- “Enumerated Types Supported for Code Generation” on page 10-3
- “When to Use Enumerated Data for Code Generation” on page 10-6
- “Generate Code for Enumerated Data from MATLAB Algorithms” on page 10-7
- “Define Enumerated Data for Code Generation” on page 10-8
- “Operations on Enumerated Data for Code Generation” on page 10-10
- “Include Enumerated Data in Control Flow Statements” on page 10-13
- “Customize Enumerated Types for Code Generation” on page 10-19
- “Use Enumerated Types in LED Control Function” on page 10-24
- “Control Names of Enumerated Type Values in Generated Code” on page 10-27
- “Change and Reload Enumerated Data Types” on page 10-30
- “Restrictions on Use of Enumerated Data in `for`-Loops” on page 10-31
- “Toolbox Functions That Support Enumerated Types for Code Generation” on page 10-32

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 10-3
Name of each enumerated data type must be unique	“Naming Enumerated Types for Code Generation” on page 10-9
Each enumerated data type must be defined in a separate file on the MATLAB path	“Define Enumerated Data for Code Generation” on page 10-8 and “Generate Code for Enumerated Data from MATLAB Algorithms” on page 10-7
Restricted set of operations	“Operations on Enumerated Data for Code Generation” on page 10-10
Restricted use in <code>for</code> -loops	“Restrictions on Use of Enumerated Data in <code>for</code> -Loops” on page 10-31

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 PrimaryColors < int32
    enumeration
        Red(1),
        Blue(2),
        Yellow(4)
    end
end
```

In this example, the statement `classdef 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 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 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 Types Used in Generated Code” on page 21-39 and “How MATLAB Coder Infers C/C++ Data Types” on page 27-9 .

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” on page 10-3.
- 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” on page 10-24
- “Define Enumerated Data for Code Generation” on page 10-8
- “Specify an Enumerated Type Input Parameter by Example” on page 18-9
- “Specify an Enumerated Type Input Parameter by Type” on page 18-13

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 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 sysMode < int32
    ...
end
```

- 3 Define enumerated values in an enumeration section:

```
classdef EnumTypeName < BaseType
    enumeration
        EnumName(N)
        ...
    end
end
```

For example, the following code defines a set of two values for enumerated type `sysMode`:

```
classdef 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 10-27.

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 10-13.

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” on page 4-2.

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 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 <= xoff</pre>	<pre>ans = 1</pre>
<pre>xon > 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 10-13
switch	<pre>switch button case VCRButton.Stop state = VCRState.Stop;</pre>	“switch Statement with Enumerated

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>	“Data Types” on page 10-14
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>	“while Statement with Enumerated Data Types” on page 10-16

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 10-3.

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 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 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 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 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 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
addClassNameToEnumNames	Specifies whether the class name becomes a prefix in the generated code.	false — 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 10-27.
getDefaultValue	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 Enumerated Value” on page 10-21.

Method	Description	Default Value Returned or Specified	When to Use
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 10-22

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 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 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 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 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 10-7
- “Customize Enumerated Types for Code Generation” on page 10-19

More About

- “Enumerated Data Definition for Code Generation” on page 10-2
- “Enumerated Types Supported for Code Generation” on page 10-3

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 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 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 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">• Locate and delete specific obsolete instances.• Delete the classes from the workspace by using the <code>clear classes</code> command. For more information, see <code>clear</code>.	<ul style="list-style-type: none">• Clear MEX functions that are caching instances of the class.

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 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`
- `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 11-2
- “Classes That Support Code Generation” on page 11-8
- “Generate Code for MATLAB Value Classes” on page 11-9
- “Generate Code for MATLAB Handle Classes and System Objects” on page 11-14
- “MATLAB Classes in Code Generation Reports” on page 11-16
- “Troubleshooting Issues with MATLAB Classes” on page 11-19
- “Handle Object Limitations for Code Generation” on page 11-21
- “System Objects Requirements and Limitations for Code Generation” on page 11-24

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.	"Create a Single, Self-Contained Class Definition File"
Restricted set of language features.	"Language Limitations" on page 11-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 11-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 11-4
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 11-14 "Handle Object Limitations for Code Generation" on page 11-21
Calls to base class constructor.	"Calls to Base Class Constructor" on page 11-5
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 11-6

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 `foo` takes one input, `a`, that is a MATLAB object, you cannot generate code for `foo` by executing:

```
codegen foo -args {a}
```

- Code generation does not support classes in matrices or structures.
- 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
        end
    end
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 11-9
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 11-14

For more information, see:

- “Role of Classes in MATLAB”
- “MATLAB Classes Definition for Code Generation” on page 11-2

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
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

Filter

Functions

- use_shape

Classes

- Rhombus
 - Rhombus
 - getarea
- Shape
 - Shape.1
 - Shape.2
 - distanceBetweenShapes
- Square
 - Square
 - getarea

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

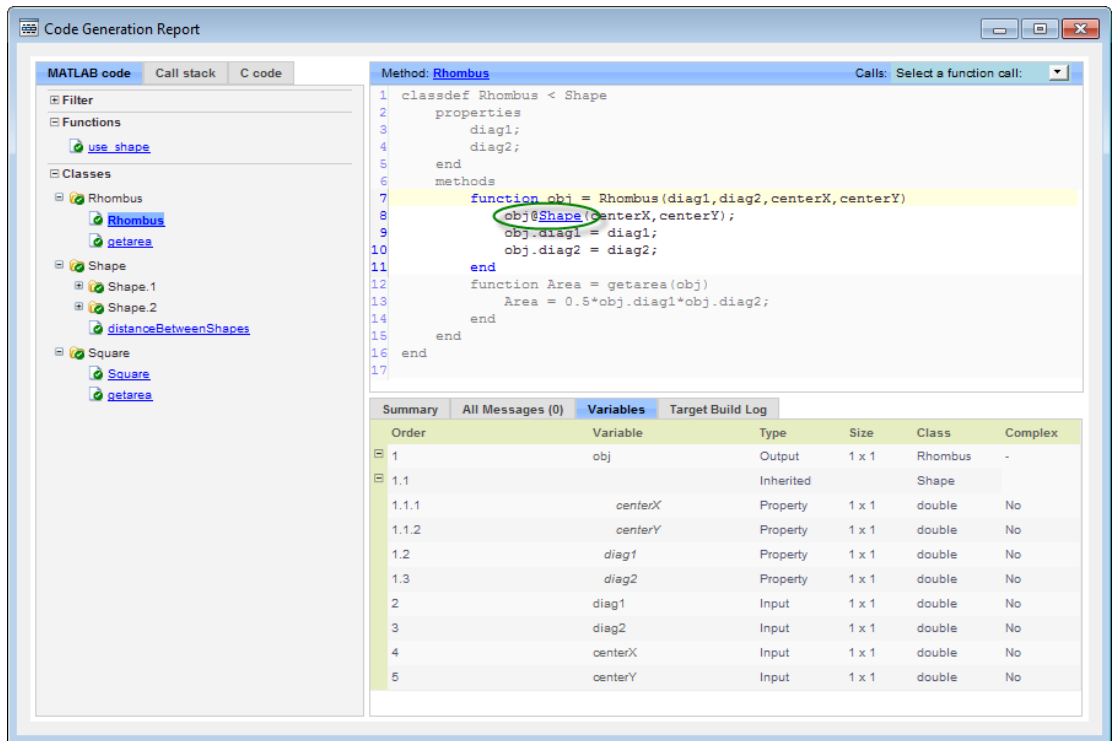
- 8 At the top right side of the report, expand the **Calls** list.

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 'MATLAB code' tab selected. On the left, a tree view shows the class hierarchy: use_shape, Rhombus, Shape, and Square. The main area displays the MATLAB code for the Rhombus class, including its constructor. A 'Calls' dropdown menu is open, showing a call from 'use_shape line 4' to the Rhombus constructor. Below the code, a table summarizes the variables used in the function.

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 screenshot displays the Code Generation Report window, which is divided into several sections. On the left, there is a navigation pane with tabs for 'MATLAB code', 'Call stack', and 'C code'. Under 'MATLAB code', there are sections for 'Filter', 'Functions' (containing a link to 'use_shape'), and 'Classes'. The 'Classes' section is expanded to show 'Rhombus', 'Shape', and 'Square'. The 'Rhombus' class is selected, and its methods 'Rhombus' and 'getarea' are listed. The 'Rhombus' method is highlighted, and a link to 'obj@Shape' is visible in the code editor.

The main area of the window shows the MATLAB code for the 'Rhombus' class. The code is as follows:

```
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
```

The code editor shows a link to 'obj@Shape' in the constructor function, which is highlighted with a green circle. This link is used to navigate to the 'Shape' class definition.

Below the code editor, there is a 'Summary' table with the following columns: 'Order', 'Variable', 'Type', 'Size', 'Class', and 'Complex'. The table contains the following data:

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

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
```

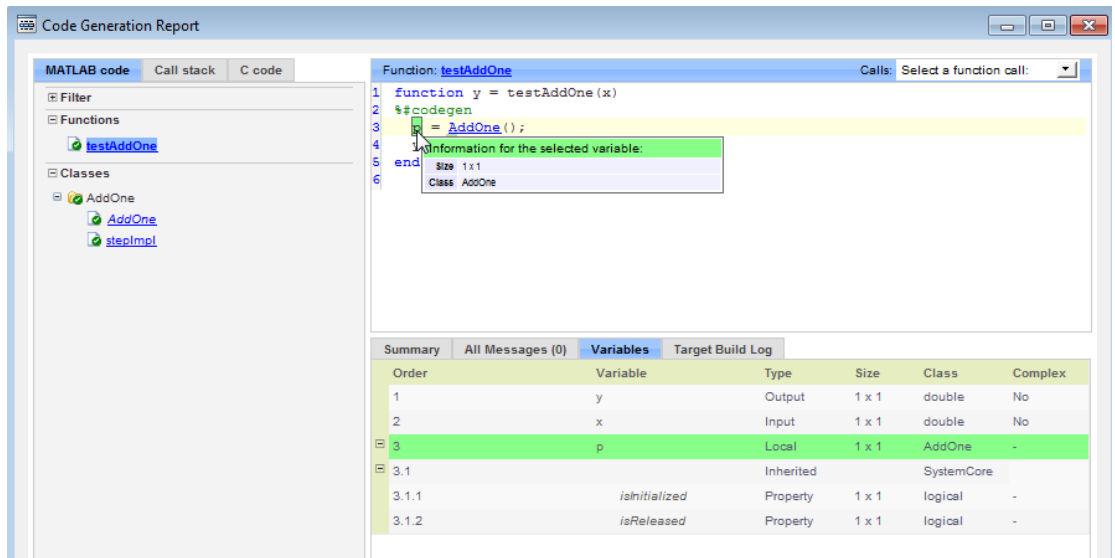
- 3 Generate a MEX function for this code.

```
codegen -report testAddOne -args {0}
```

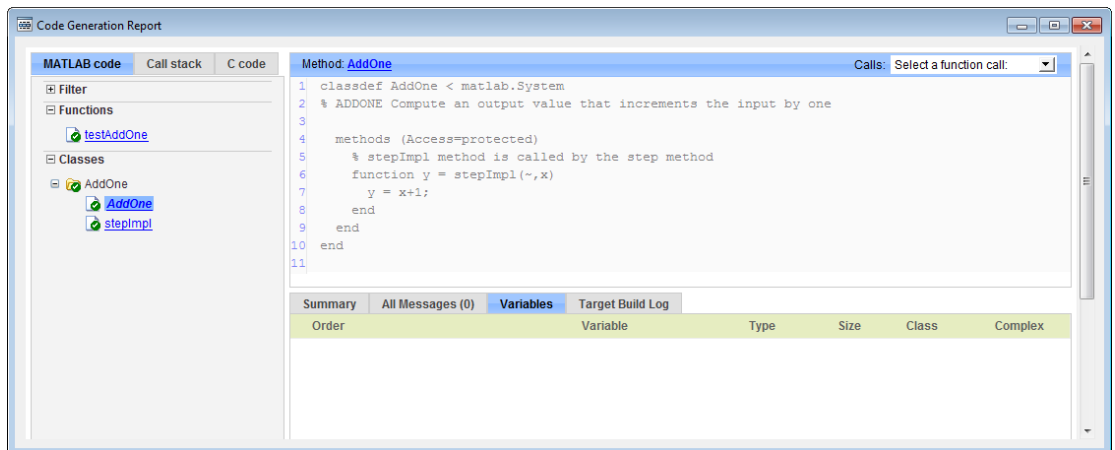
The `-report` option instructs `codegen` to generate a code generation report, even if no errors or warnings occur. The `-args` option specifies that the `testAddOne` function takes one scalar double input.

```
>> 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

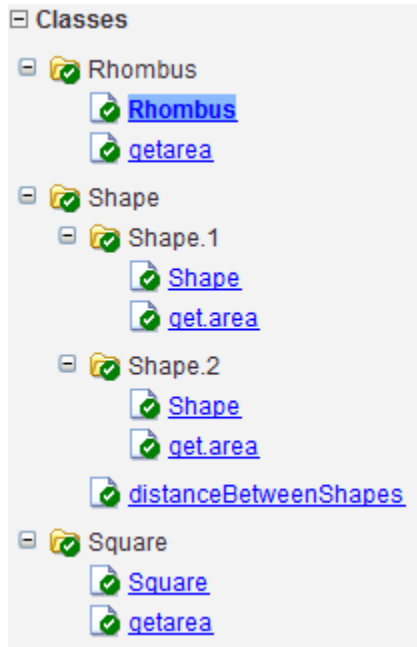
If a class has a default constructor, the report displays the constructor in italics.

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;
```

Handle Object Limitations for Code Generation

The code generation software statically determines the lifetimes of handle objects. It can reuse memory rather than rely on a dynamic memory management scheme such as reference counting or garbage collection. It generates code that does not use dynamic memory allocation or incur the overhead of run-time automatic memory management. These characteristics of the generated code are important for some safety-critical and real-time applications.

When you use handle objects, the static analysis that the code generation software uses requires that you adhere to the following restrictions:

- “A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop” on page 11-21
- “A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object” on page 11-21

A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop

Consider the handle class `mycls` and the function `usehandle1`. The code generation software reports an error because `p`, which is outside the loop, has a property that refers to a `mycls` object created inside the loop.

```
classdef mycls < handle
    properties
        prop
    end
end

function usehandle1
    p = mycls;
    for i = 1:10
        p.prop = mycls;
    end
end
```

A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object

If a persistent variable refers to a handle object, the code generation software allows only one instance of the object during the program’s lifetime. The object must be a *singleton*

object. To create a singleton handle object, enclose statements that create the object in the `if isempty()` guard for the persistent variable.

For example, consider the class `mycls` and the function `usehandle2`. The code generation software reports an error for `usehandle2` because `p.prop` refers to the `mycls` object that the statement `inner = mycls` creates. This statement creates a `mycls` object for each invocation of `usehandle2`.

```
classdef mycls < handle
    properties
        prop
    end
end

function usehandle2(x)
    assert(isa(x, 'double'));
    persistent p;
    inner = mycls;
    inner.prop = x;
    if isempty(p)
        p = mycls;
        p.prop = inner;
    end
end
```

If you move the statements `inner = mycls` and `inner.prop = x` inside the `if isempty()` guard, code generation succeeds. The statement `inner = mycls` executes only once during the program's lifetime.

```
function usehandle2(x)
    assert(isa(x, 'double'));
    persistent p;
    if isempty(p)
        inner = mycls;
        inner.prop = x;
        p = mycls;
        p.prop = inner;
    end
end
```

Consider the function `usehandle3`. The code generation software reports an error for `usehandle3` because the persistent variable `p` refers to the `mycls` object that the statement `myobj = mycls` creates. This statement creates a `mycls` object for each invocation of `usehandle3`.

```
function usehandle3(x)
```

```
assert(isa(x, 'double'));
myobj = mycls;
myobj.prop = x;
doinit(myobj);
disp(myobj.prop);
function doinit(obj)
persistent p;
if isempty(p)
    p = obj;
end
```

If you make `myobj` persistent and enclose the statement `myobj = mycls` inside an `if isempty()` guard, code generation succeeds. The statement `myobj = mycls` executes only once during the program's lifetime.

```
function usehandle3(x)
assert(isa(x, 'double'));
persistent myobj;
if isempty(myobj)
    myobj = mycls;
end

doinit(myobj);

function doinit(obj)
persistent p;
if isempty(p)
    p = obj;
end
```

System Objects Requirements and Limitations for Code Generation

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty()`.
- Set arguments to System object constructors as compile-time constants.
- You cannot initialize System objects properties with other MATLAB class objects as default values in code generation. You must initialize these properties in the constructor.

Inputs and Outputs

- System objects accept a maximum of 32 inputs. A maximum of 8 dimensions per input is supported.
- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

Note: Variable-size properties in MATLAB Function block in Simulink are not supported. System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.

- Do not set System objects to become outputs from the MATLAB Function block.
- Do not use the Save and Restore Simulation State as SimState option for any System object in a MATLAB Function block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.
 - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
 - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation. For `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.
- Objects cannot be used as default values for properties.
- In MATLAB simulations, default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Cell Arrays and Global Variables

- System objects can contain cell arrays, but cell arrays cannot contain System objects.
- Global variables are allowed in a System object, unless you will be using that System object in Simulink via the MATLAB System block. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

```
f = coder.MEXConfig;  
f.GlobalSyncMethod = 'NoSync'  
Then, include '-config f' in your codegen command.
```

Methods

- Code generation support is available only for these System object methods:
 - `get`
 - `getNumInputs`
 - `getNumOutputs`
 - `isDone` (for sources only)
 - `isLocked`
 - `release`

- `reset`
- `set` (for tunable properties)
- `step`
- For System objects that you define,

Code generation support is available only for these methods:

- `getDiscreteStateImpl`
- `getNumInputsImpl`
- `getNumOutputsImpl`
- `infoImpl`
- `isDoneImpl`
- `isInputDirectFeedThroughImpl`
- `outputImpl`
- `processTunedPropertiesImpl`
- `releaseImpl` — Code is not generated automatically for the this method. To release an object, you must explicitly call the `release` method in your code.
- `resetImpl`
- `setupImpl`
- `stepImpl`
- `updateImpl`
- `validateInputsImpl`
- `validatePropertiesImpl`
- Code generation support for using dot notation depends on whether the System object is predefined in the software or is one that you defined.
 - For System objects that are predefined in the software, you cannot use dot notation to call methods.
 - For System objects that you define, you can use dot notation or function call notation, with the System object as first argument, to call methods.

Code Generation for Function Handles

- “Function Handle Definition for Code Generation” on page 12-2
- “Define and Pass Function Handles for Code Generation” on page 12-3
- “Function Handle Limitations for Code Generation” on page 12-5

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” on page 4-2)

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 function handles, 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 12-5

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 14-12.

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 13-2
- “Supported Index Expressions” on page 13-3
- “Apply Operations to a Variable Number of Arguments” on page 13-4
- “Implement Wrapper Functions” on page 13-6
- “Pass Property/Value Pairs” on page 13-7
- “Variable Length Argument Lists for Code Generation” on page 13-9

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 13-4
- “Implement Wrapper Functions” on page 13-6
- “Pass Property/Value Pairs” on page 13-7

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 13-9).

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 13-9.

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 13-9.

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 `varargout` and `varargin` 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 13-3.

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 13-4.

Do not write to varargin

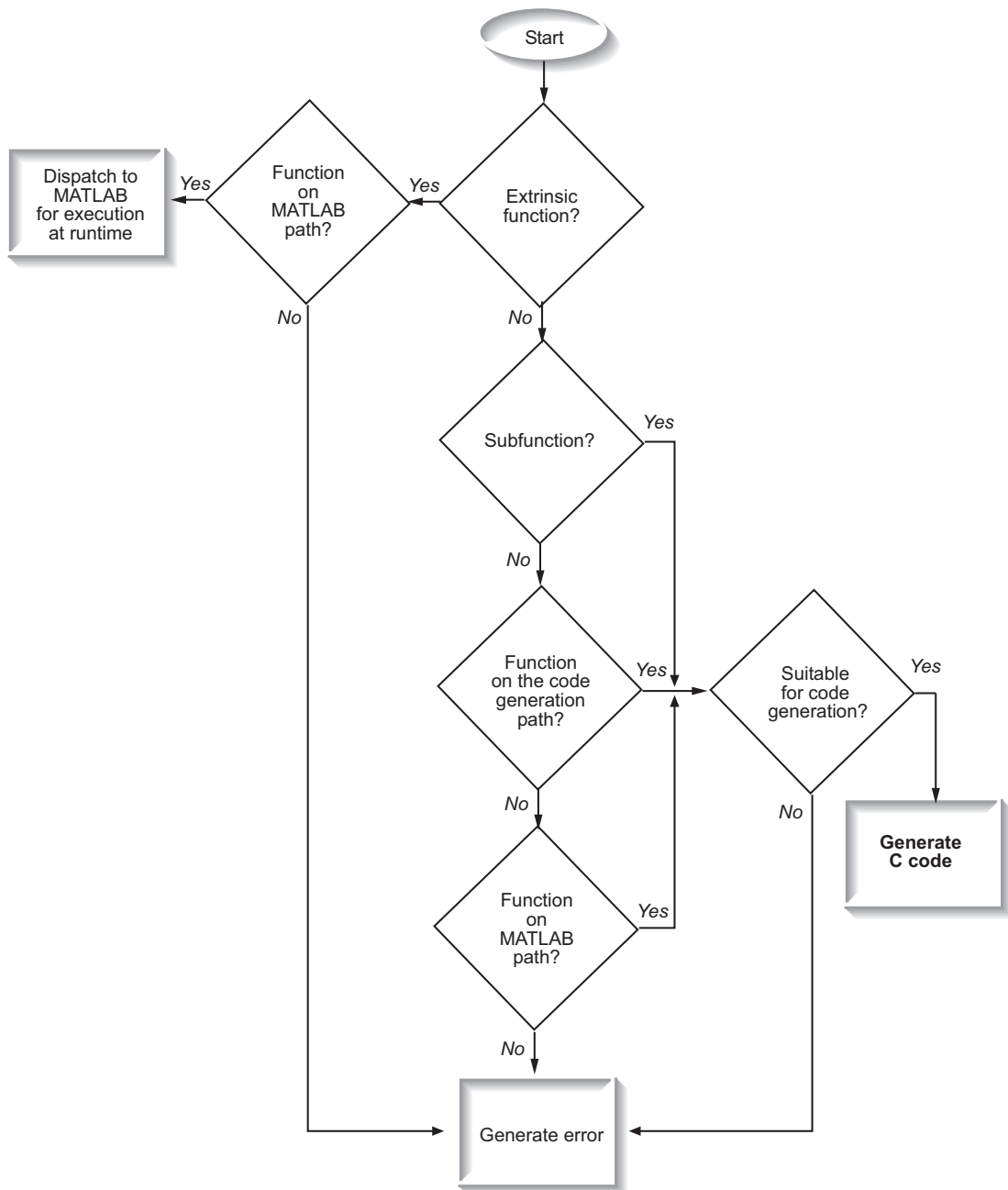
Generated code treats `varargin` as a read-only variable. If you want to write to input arguments, copy the values into a local variable.

Calling Functions for Code Generation

- “Resolution of Function Calls for Code Generation” on page 14-2
- “Resolution of File Types on Code Generation Path” on page 14-6
- “Compilation Directive `%#codegen`” on page 14-8
- “Call Local Functions” on page 14-9
- “Call Supported Toolbox Functions” on page 14-10
- “Call MATLAB Functions” on page 14-11

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 14-4.

- 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” on page 14-12. 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 14-6

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.

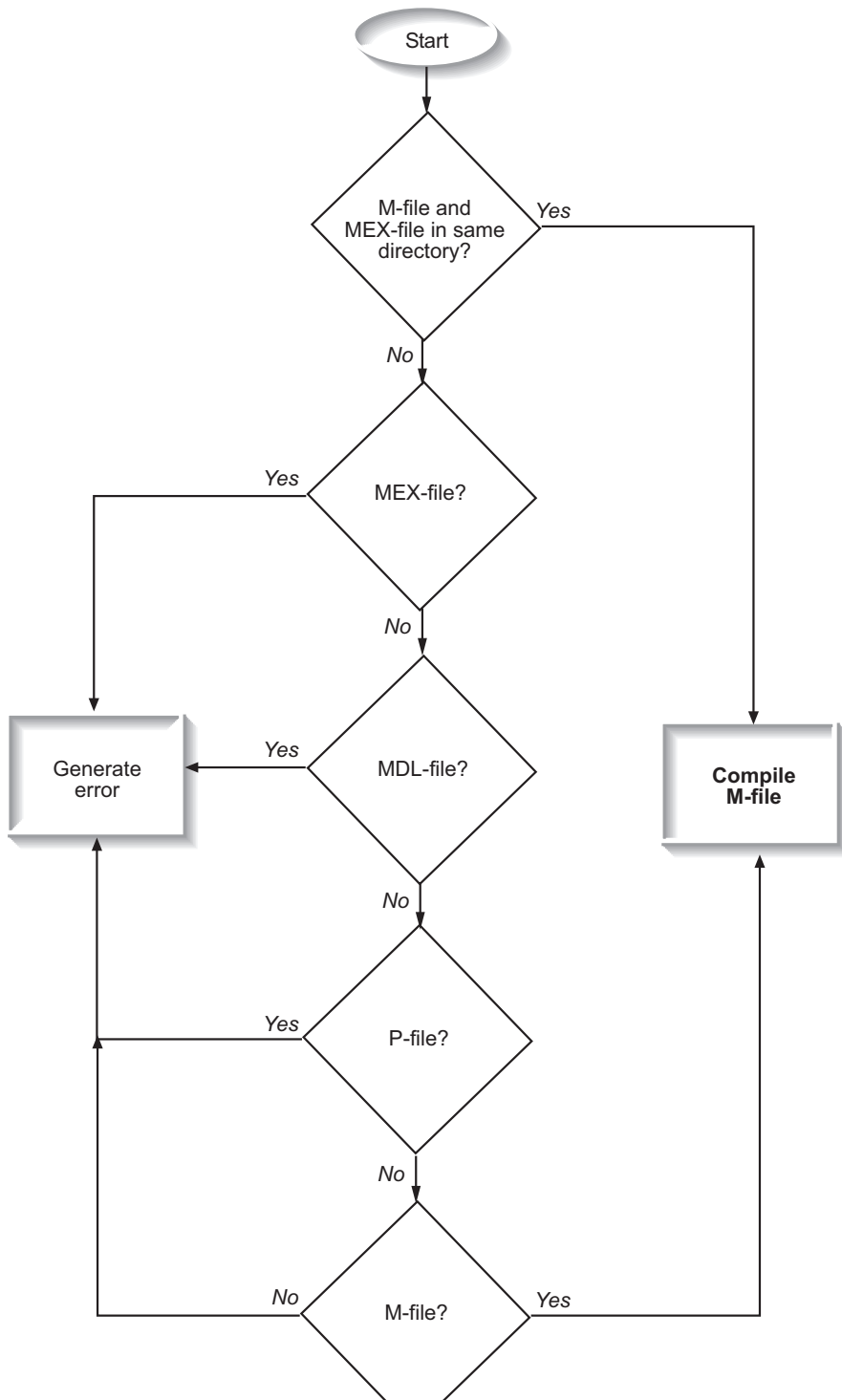
MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order”).

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 for the function 'stats':

```

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` 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 14-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 14-16).

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 14-12).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 14-16).

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');

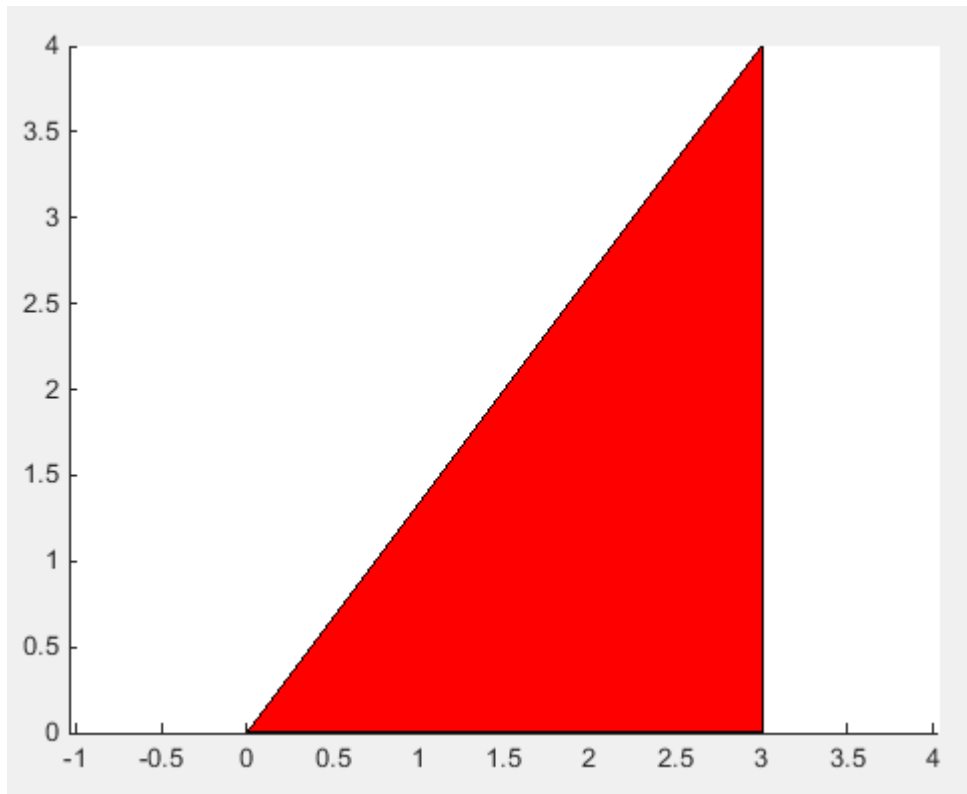
```

A tooltip points to the `axis('equal');` line, 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 14-16).
- 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 14-17).

- 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 14-15).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 14-15). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 14-16).

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 14-16.

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.

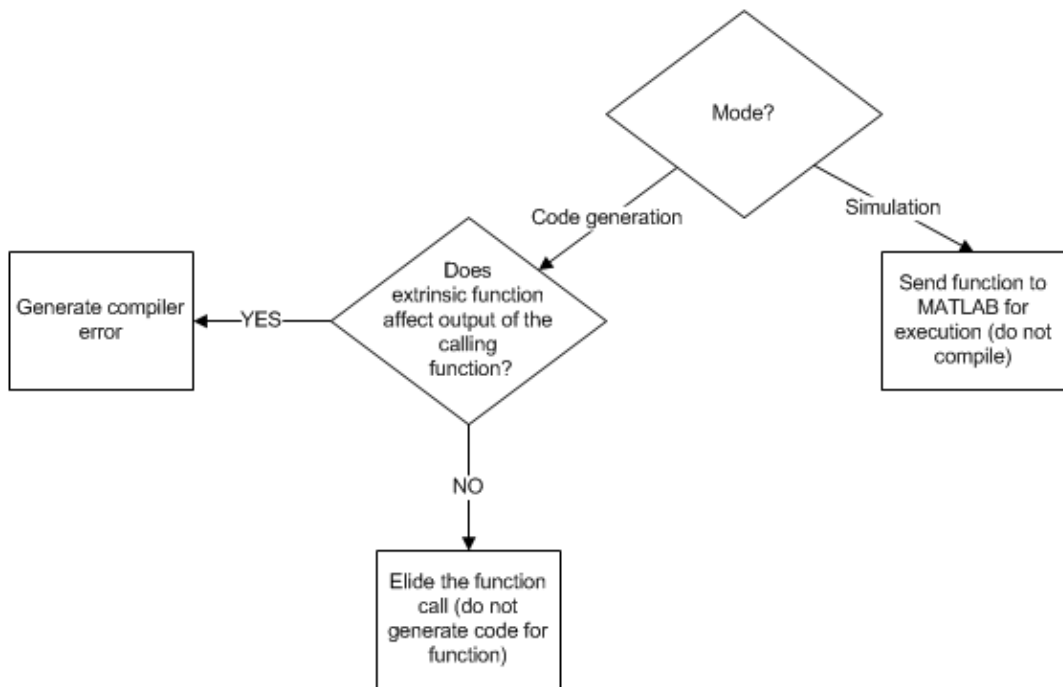
Consider the following example:

```
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 14-17). 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 14-18.

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

- “Detect Dead and Constant-Folded Code” on page 15-2
- “Convert MATLAB Code to Fixed-Point C Code” on page 15-7
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 15-9
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-23
- “Specify Type Proposal Options” on page 15-38
- “Detect Overflows” on page 15-42
- “Replace the exp Function with a Lookup Table” on page 15-53
- “Replace a Custom Function with a Lookup Table” on page 15-62
- “Enable Plotting Using the Simulation Data Inspector” on page 15-71
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 15-72
- “Log Data for Histogram” on page 15-83
- “View and Modify Variable Information” on page 15-85
- “Automated Fixed-Point Conversion” on page 15-89
- “Convert Fixed-Point Conversion Project to MATLAB Scripts” on page 15-110
- “Generated Fixed-Point Code” on page 15-113
- “Fixed-Point Code for MATLAB Classes” on page 15-119
- “Automated Fixed-Point Conversion Best Practices” on page 15-122
- “Replacing Functions Using Lookup Table Approximations” on page 15-130
- “MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 15-131
- “Inspecting Data Using the Simulation Data Inspector” on page 15-133
- “Custom Plot Functions” on page 15-136
- “Data Type Issues in Generated Code” on page 15-138

Detect Dead and Constant-Folded Code

The MATLAB Coder app detects code that is constant folded and dead code during the simulation of your test file. The app uses the code coverage information when translating your code from floating-point MATLAB code to fixed-point MATLAB code. Reviewing code coverage results helps you verify that your test file is exercising the algorithm adequately.

The app inserts inline comments in the fixed-point code to mark the dead and untranslated regions and includes the code coverage information in the generated fixed-point conversion HTML report. The app also displays a color-coded coverage bar to the left of the code within its editor.

Coverage Bar Color	How Often Code Is Executed During Test File Simulation
Dark green	Always
Light green	Sometimes
Orange	Once
Red	Never

What Is Dead Code?

Dead code is code that does not execute during simulation. Dead code can result from these scenarios:

- Defensive code containing intended corner cases that are never reached
- Human error in the code, resulting in code that cannot be reached by any execution path
- Inadequate test bench range
- Constant folding

Detect Dead Code

This example shows how to detect dead code in your algorithm using the MATLAB Coder app.

- 1 In a local writable folder, create the function `myFunction.m`.

```
function y = myFunction(u,v)
```



```

    %#codegen
    for i = 1:length(u)
        if u(i) > v(i)
            y=bar(u,v);
        else
            tmp = u;
            v = tmp;
            y = baz(u,v);
        end
    end
end

function y = bar(u,v)
    y = u+v;
end

function y = baz(u,v)
    y = u-v;
end

```

- 2 In the same folder, create a test file, myFunction_tb.

```

u = 1:100;
v = 101:200;

```

```

myFunction(u,v);

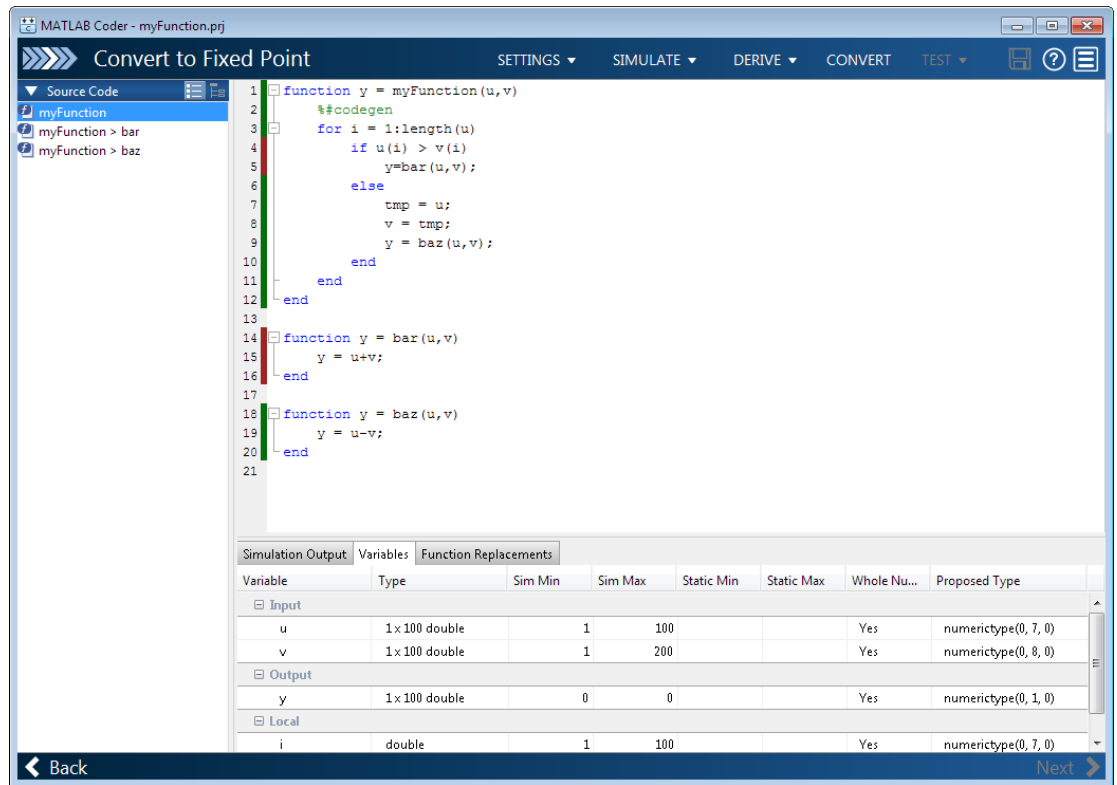
```

- 3 From the apps gallery, open the MATLAB Coder app.
- 4 Set **Numeric Conversion** to **Convert to fixed point**.
- 5 On the **Select Source Files** page, browse to the myFunction file, and click **Open**.
- 6 Click **Next**. On the **Define Input Types** page, browse to select the test file you created, myFunction_tb. Click **Autodefine Input Types**.
- 7 Click **Next**. On the **Check for Run-Time Issues** page, click **Check for Issues**.

The app runs the myFunction_tb test file and detects no issues.

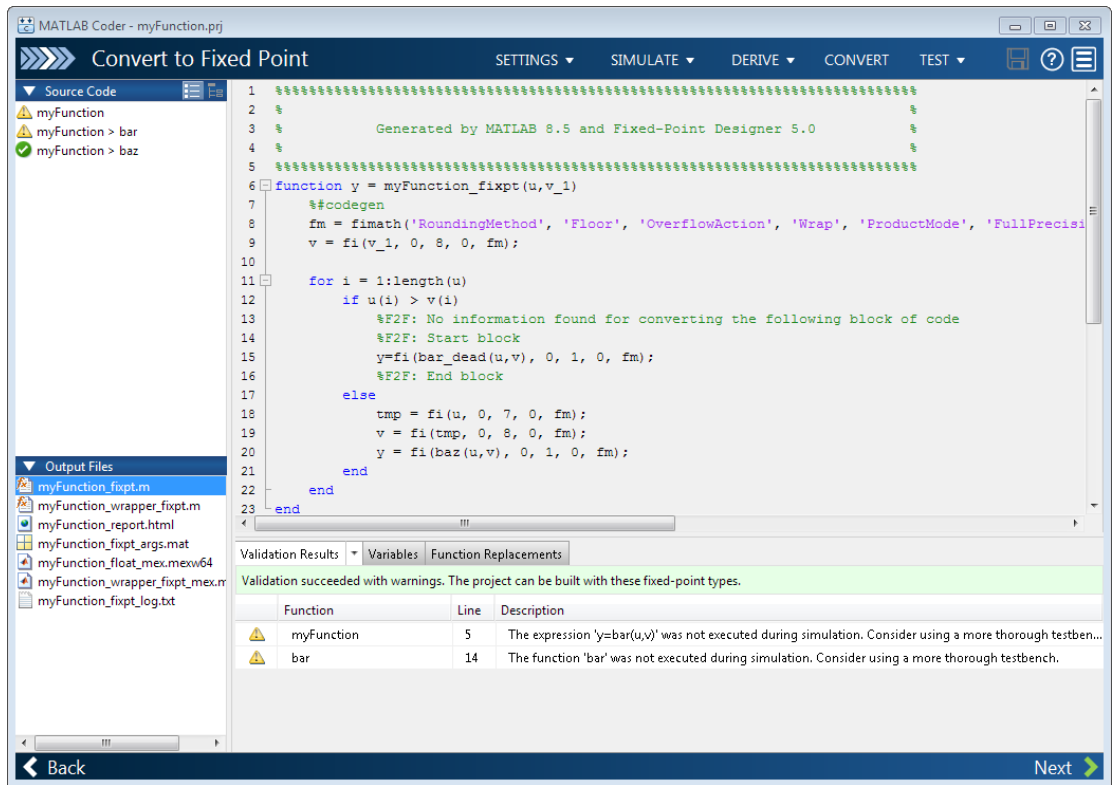
- 8 Click **Next**. On the **Convert to Fixed-Point** page, click **Simulate** to simulate the entry-point functions, gather range information, and get proposed data types.

The code coverage bar on the left-hand side of the edit window indicates if the code executed. The code in the first condition of the if-statement did not execute during simulation because u is never greater than v . Because the if-statement never executed, the `bar` function also never executed. These parts of the algorithm are marked with a red bar, indicating that they are dead code.



- 9 Click **Convert** to apply the proposed data types to the function.

The MATLAB Coder app generates a fixed-point function, `myFunction_fixpt`. The generated fixed-point code contains comments around the pieces of code identified as dead code, and the **Validation Results** pane suggests that you use a more thorough test bench.



When the MATLAB Coder app detects dead code, consider editing your test file so your algorithm is exercised over its full range. If your test file already reflects the full range of the input variables, consider editing your algorithm to eliminate the dead code.

10 Close the MATLAB Coder app.

Fix Dead Code

1 Edit the test file `myFunction_tb.m` to include a wider range of inputs.

```
u = 1:100;
v = -50:2:149;

myFunction(u,v);
```

- 2 Open the MATLAB Coder app again.
- 3 Using the same function and the edited test file, go through the conversion process again.
- 4 After you click **Simulate**, this time the code coverage bar shows that all parts of the algorithm were executed with the new test file input ranges.

The screenshot shows the MATLAB Coder interface for converting code to fixed point. The code editor displays the following functions:

```

1 function y = myFunction(u,v)
2     %%codegen
3     for i = 1:length(u)
4         if u(i) > v(i)
5             y=bar(u,v);
6         else
7             tmp = u;
8             v = tmp;
9             y = baz(u,v);
10        end
11    end
12 end
13
14 function y = bar(u,v)
15     y = u+v;
16 end
17
18 function y = baz(u,v)
19     y = u-v;
20 end
21

```

Below the code editor is a table showing simulation results for variables:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
u	1x100 double	1	100			Yes	numerictype(0, 7, 0)
v	1x100 double	-50	148			Yes	numerictype(1, 9, 0)
Output							
y	1x100 double	-49	248			Yes	numerictype(1, 9, 0)
Local							
i	double	1	100			Yes	numerictype(0, 7, 0)

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

- 5 Click **Convert** to finish the conversion process and convert the function to fixed point.

Convert MATLAB Code to Fixed-Point C Code

To convert MATLAB Code to fixed-point C Code using the MATLAB Coder app:

- 1 Open the MATLAB Coder app.
- 2 Set **Fixed-Point Conversion** to Enabled.
- 3 On the **Select Source Files** page, add the entry-point function from which you want to generate code.
- 4 Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. If the app does not find issues, it opens the **Define Input Types** page.
- 5 On the **Define Input Types** page, specify a test file that the app can use to define the input types.
- 6 Click **Next** to go to the **Check for Run-Time Issues** step.
- 7 On the **Check for Run-Time Issues** page, specify a test file that calls your entry-point function. Alternatively, at the prompt, enter code that calls your entry-point function. The app generates instrumented MEX. It runs the test file or code that you specified, replacing calls to your entry-point function with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. Click a message to highlight the problematic code in a window where you can edit the code.
- 8 Click **Next** to go to the **Convert to Fixed Point** step.
- 9 Propose data types based on simulation range data, derived (also known as static) range data, or both. See “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 15-9 and “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-23.
- 10 To convert the floating-point MATLAB code to fixed-point MATLAB code, click **Convert**. During fixed-point conversion, the app validates the build using the proposed fixed-point data types. See “Validating Types” on page 15-108.
- 11 Verify the behavior of the fixed-point MATLAB code. See “Testing Numerics” on page 15-108.
- 12 Click **Next** to go to the **Generate Code** step.
- 13 In the **Generate** dialog box, set **Build source** to **Fixed-Point**. Set the **Build type** to build a static or dynamic library, or executable. Set **Language** to **C**. Click **Generate**.

MATLAB Coder generates fixed-point C code for your entry-point MATLAB function.

Related Examples

- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 15-9
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-23

Propose Fixed-Point Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data using the MATLAB Coder app.

Prerequisites

This example requires the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). See http://www.mathworks.com/support/compilers/current_release/

You can use `mex -setup` to change the default compiler. See “Change 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)
```

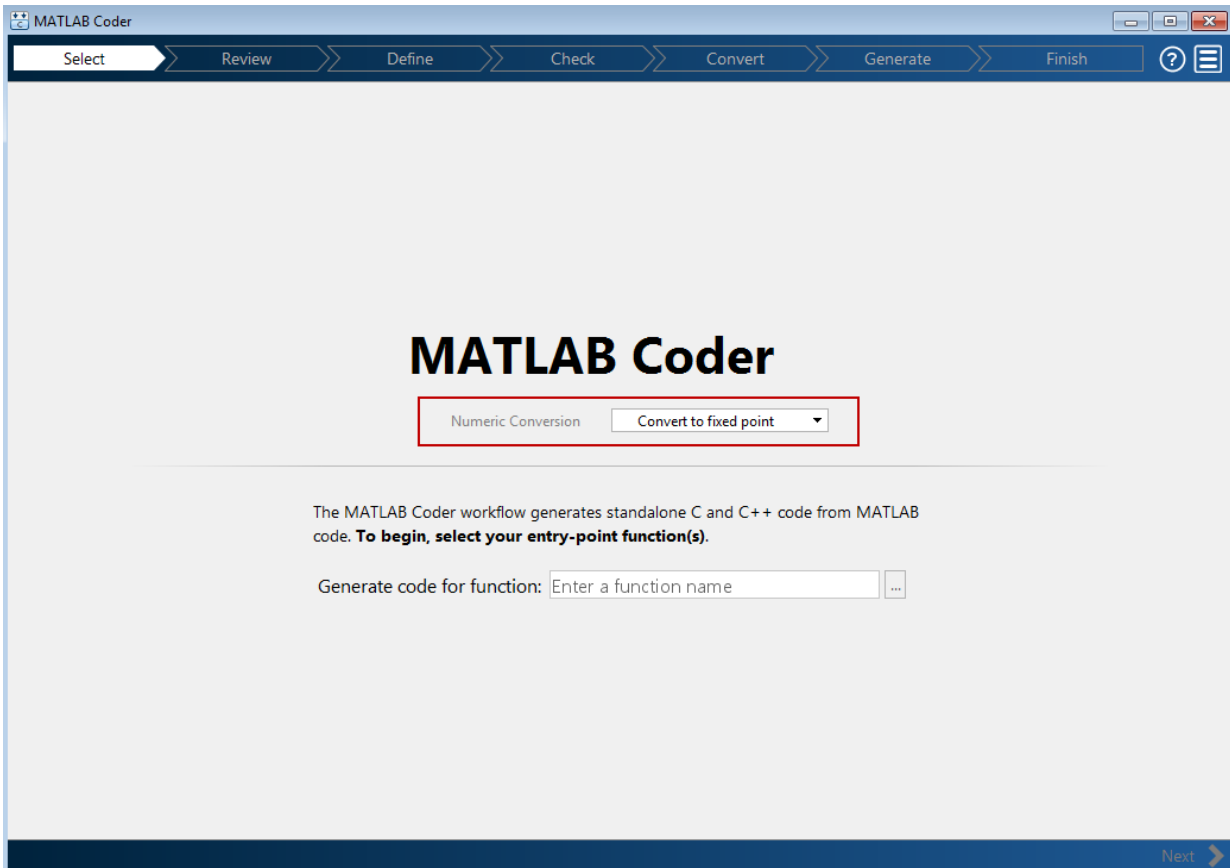
```
disp('Test complete.')
```

Open the MATLAB Coder App

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Fixed-Point Conversion

Set **Numeric Conversion** to **Convert to fixed point**.



Select Source Files

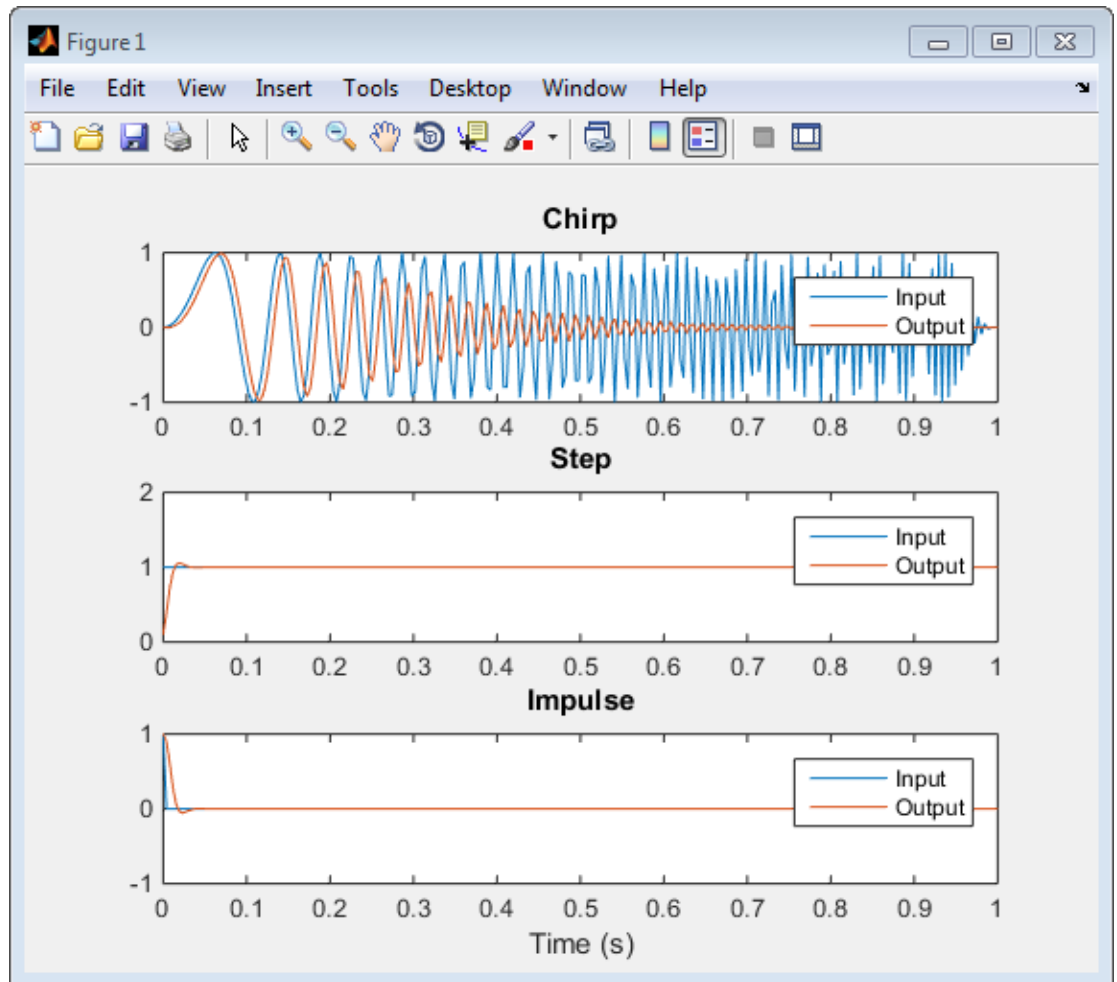
- 1 To add the entry-point function `fun_with_matlab` to the project, browse to the file `fun_with_matlab.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `fun_with_matlab.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `fun_with_matlab.m` for code violations and code generation readiness issues. The app does not find issues in `fun_with_matlab.m`.

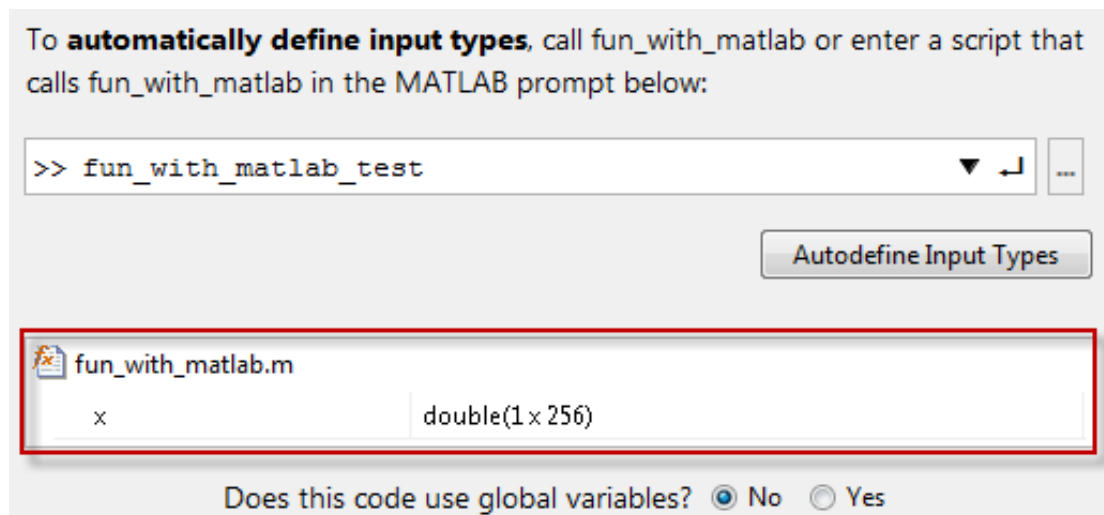
Define Input Types

- 1 On the **Define Input Types** page, to add `fun_with_matlab_test` as a test file, browse to `fun_with_matlab_test`, and then click **Open**.
- 2 Click **Autodefine Input Types**.

The test file runs and displays the outputs of the filter for each of the input signals.



The app determines from the test file that the input type of x is `double(1x256)`.



- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `fun_with_matlab_test` replacing calls to `fun_with_matlab` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a window where you can edit the code.

- 1 On the **Check for Run-Time Issues** page, the app populates the test file field with `fun_with_matlab_test`, the test file that you used to define the input types.
- 2 Click **Check for Issues**.

The app does not detect issues.

- 3 Click **Next** to go to the **Convert to Fixed Point** step.

Convert to Fixed Point

- 1 The app displays compiled information—type, size, and complexity—for variables in your code. See “View and Modify Variable Information” on page 15-85.


```

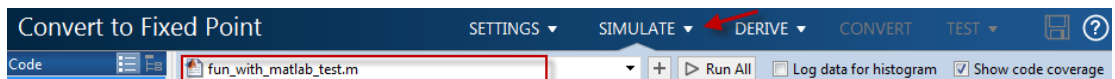
1 function y = fun_with_matlab(x) %#codegen
2     persistent z
3     if isempty(z)
4         z = zeros(2,1);
5     end
6     % [b,a] = butter(2, 0.25)
7     b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
8     a = [
9         1, -0.942809041582063, 0.333333333333333];
10
11    y = zeros(size(x));
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

```

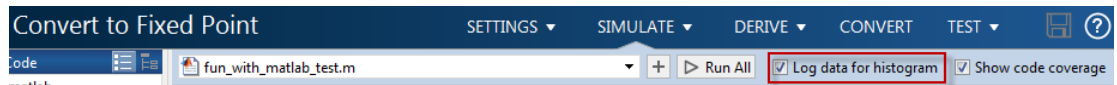
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	1x256 double					No	
Output							
y	1x256 double					No	
Persistent							
z	2x1 double					No	
Local							

On the **Function Replacements** tab, the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 15-94.

- Click the **Simulate** arrow . Verify that the test file is `fun_with_matlab_test`. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the app merges the simulation results.



- 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.

4 Click **Simulate**.

The simulation runs and the app 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. The dark green line to the left of the code indicates that the code runs every time the algorithm executes. The orange bar indicates that the code next to it executes only once. This behavior is expected for this example because the code initializes a persistent variable. If your test file does not cover all of your code, update the test or add more test files.

Simulation Output

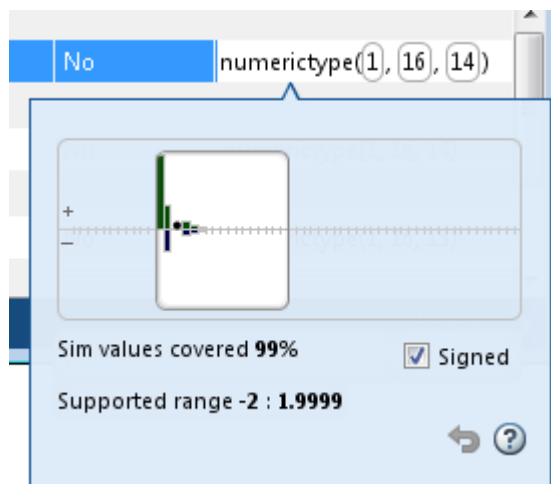
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	1 x 256 double	-1...	1			No	numericType(1, 16, 14)
Output							
y	1 x 256 double	-0.97...	1.06...			No	numericType(1, 16, 14)
Persistent							
z	2 x 1 double	-0.89...	0.96...			No	numericType(1, 16, 15)
Local							

If a value has . . . next to it, the value is rounded. Place your cursor over the . . . to view the actual value.

The app 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 app enables the **Convert** option.

Note: You can manually enter static ranges. These manually entered ranges take precedence over simulation ranges. The app uses the manually entered ranges to propose data types. You can also modify and lock the proposed type.

- 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.



To modify the proposed data types, either enter the required type into the **Proposed Type** field or use the histogram controls. For more information about the histogram, see “Log Data for Histogram” on page 15-105.

- To convert the floating-point algorithm to fixed point, click **Convert**.

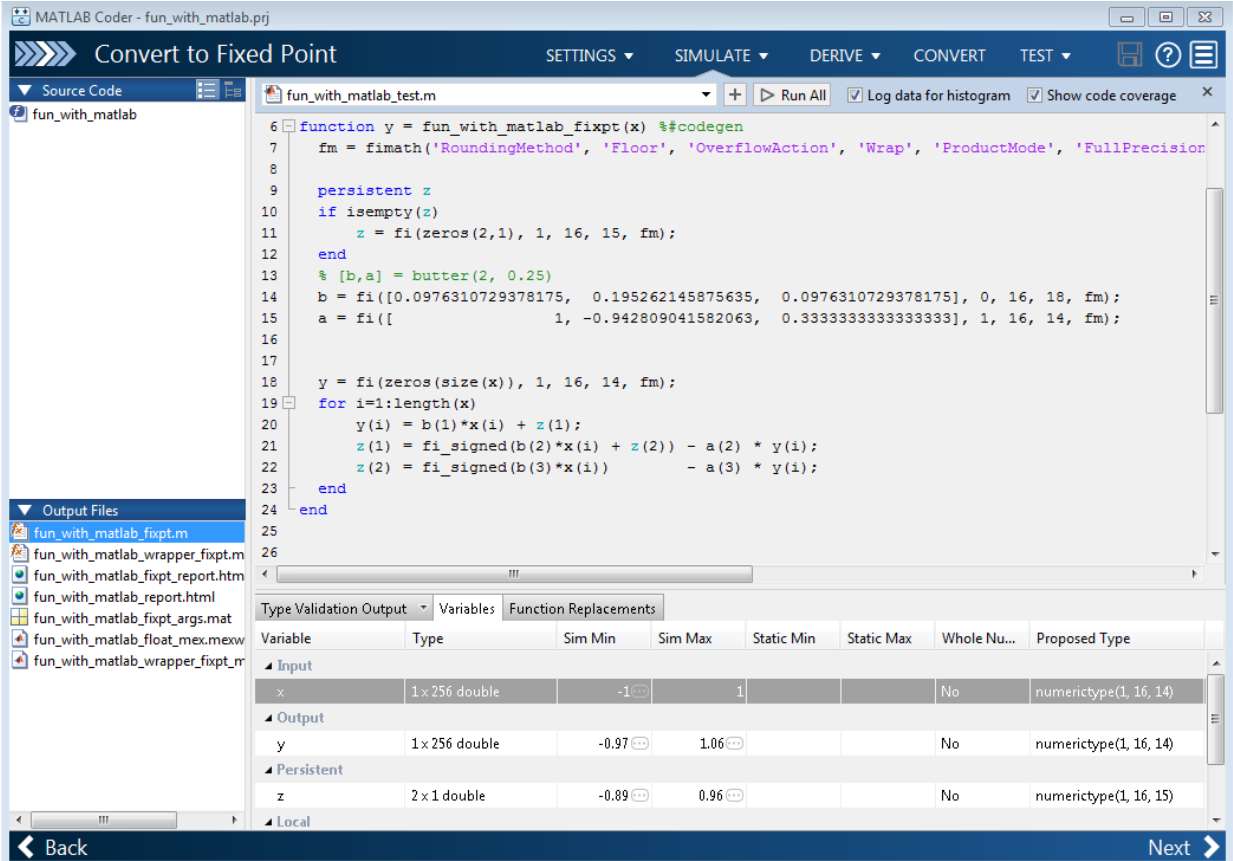
During the fixed-point conversion process, the software validates the proposed types and generates the following files in the `codegen\fun_with_matlab\fixpt` folder in your local working folder.

- `fun_with_matlab_fixpt.m` — the fixed-point version of `fun_with_matlab.m`.
- `fun_with_matlab_wrapper_fixpt.m` — this file converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during conversion. These fixed-point values are fed into the converted fixed-point design, `fun_with_matlab_fixpt.m`.
- `fun_with_matlab_fixpt_report.html` — this report shows the generated fixed-point code and the fixed-point instrumentation results.
- `fun_with_matlab_report.html` — this report shows the original algorithm and the fixed-point instrumentation results.

- `fun_with_matlab_fixpt_exVal.mat` — MAT-file containing a structure for the input arguments, a structure for the output arguments and the name of the fixed-point file.

If errors or warnings occur during validation, you see them on the **Type Validation Output** tab. See “Validating Types” on page 15-108.

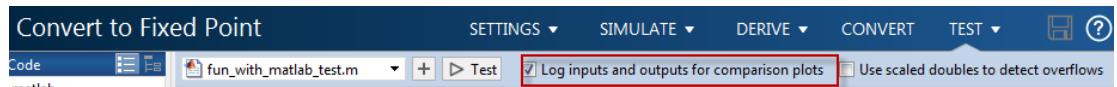
- 7 In the **Output Files** list, select `fun_with_matlab_fixpt.m`. The app displays the generated fixed-point code.



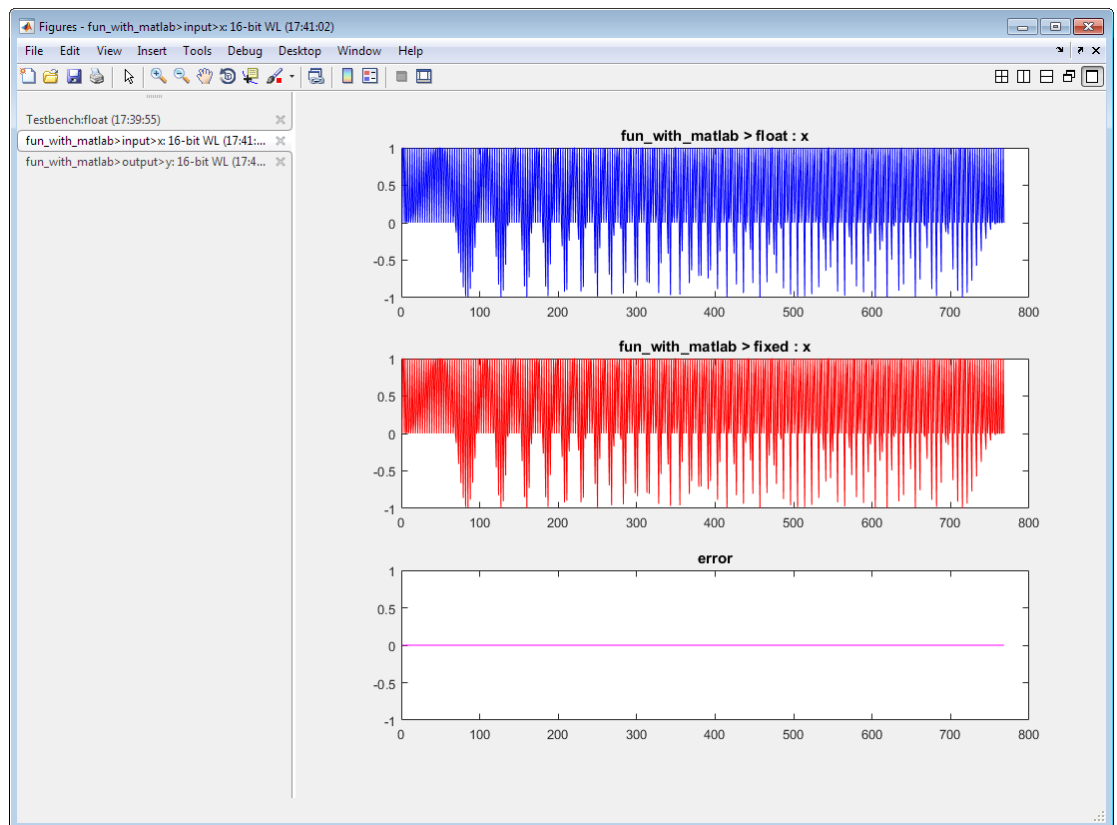
The screenshot shows the MATLAB Code Generator interface for the 'Convert to Fixed Point' app. The main window displays the generated fixed-point code for a Butterworth filter. The 'Output Files' list on the left shows the generated files, with 'fun_with_matlab_fixpt.m' selected. The 'Type Validation Output' table at the bottom shows the proposed fixed-point types for variables x, y, and z.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	1x256 double	-1	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							

- 8 Click the **Test** arrow . Select **Log inputs and outputs for comparison plots**, and then click **Test**.



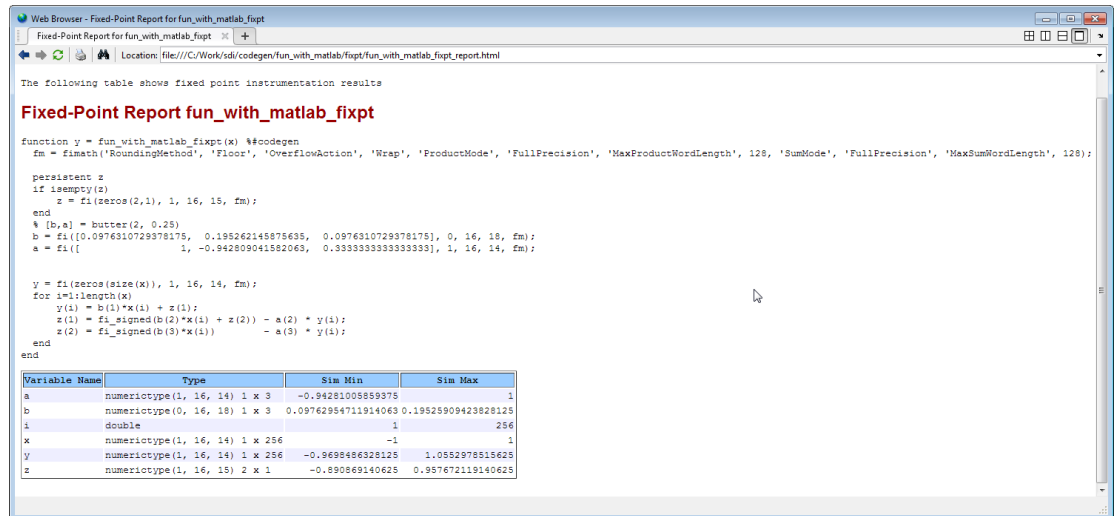
To test the fixed-point MATLAB code, the app runs the test file that you used to define input types. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable y . Because you selected to log inputs and outputs for comparison plots, the app generates a plot for each input and output. The app docks these plots in a single figure window.



The app also reports error information on the **Verification Output** tab. The maximum error is less than 0.03%. For this example, this margin of error is acceptable.

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics” on page 15-108.

- 9 On the **Verification Output** tab, the app provides a link to a type proposal report. The report displays the generated fixed-point code and the proposed type information.



The following table shows fixed point instrumentation results

Fixed-Point Report fun_with_matlab_fixpt

```
function y = fun_with_matlab_fixpt(x) %%codegen
    fm = fimmath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

    persistent z
    if isempty(z)
        z = fi(zeros(2,1), 1, 16, 16, fm);
    end
    % [b,a] = butter(2, 0.25)
    b = fi([0.0976310729378175, 0.195262145875635, 0.0976310729378175], 0, 16, 16, fm);
    a = fi([1, -0.942809041582063, 0.333333333333333], 1, 16, 16, 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
```

Variable Name	Type	Sim Min	Sim Max
a	numerictype(1, 16, 14) 1 x 3	-0.94281005859375	1
b	numerictype(0, 16, 18) 1 x 3	0.09762954711914063	0.19525909423828125
i	double	1	256
x	numerictype(1, 16, 14) 1 x 256	-1	-1
y	numerictype(1, 16, 14) 1 x 256	-0.9698466328125	1.0552975115625
z	numerictype(1, 16, 15) 2 x 1	-0.830869140625	0.357672119140625

- 10 Click **Next** to go to the **Generate Code** page.

Generate Fixed-Point C Code

- 1 In the **Generate** dialog box, set **Build source** to **Fixed-Point** and **Build type** to **Static Library**.
- 2 Set **Language** to **C**.
- 3 Click **Generate** 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`.

- 4 The app displays the generated code for `fun_with_matlab.c`. In the generated C code, variables are assigned fixed-point data types.
- 5 Click **Next** to go to the **Finish Workflow** page.

On the **Finish Workflow** page, the app displays a project summary and links to generated output files.

Propose Fixed-Point Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges using the MATLAB Coder app. When you propose data types based on derived ranges you, do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a long time. You can save time by deriving ranges instead.

Prerequisites

This example requires the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). See http://www.mathworks.com/support/compilers/current_release/.

You can use `mex -setup` to change the default compiler. See “Change 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;
```

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)')

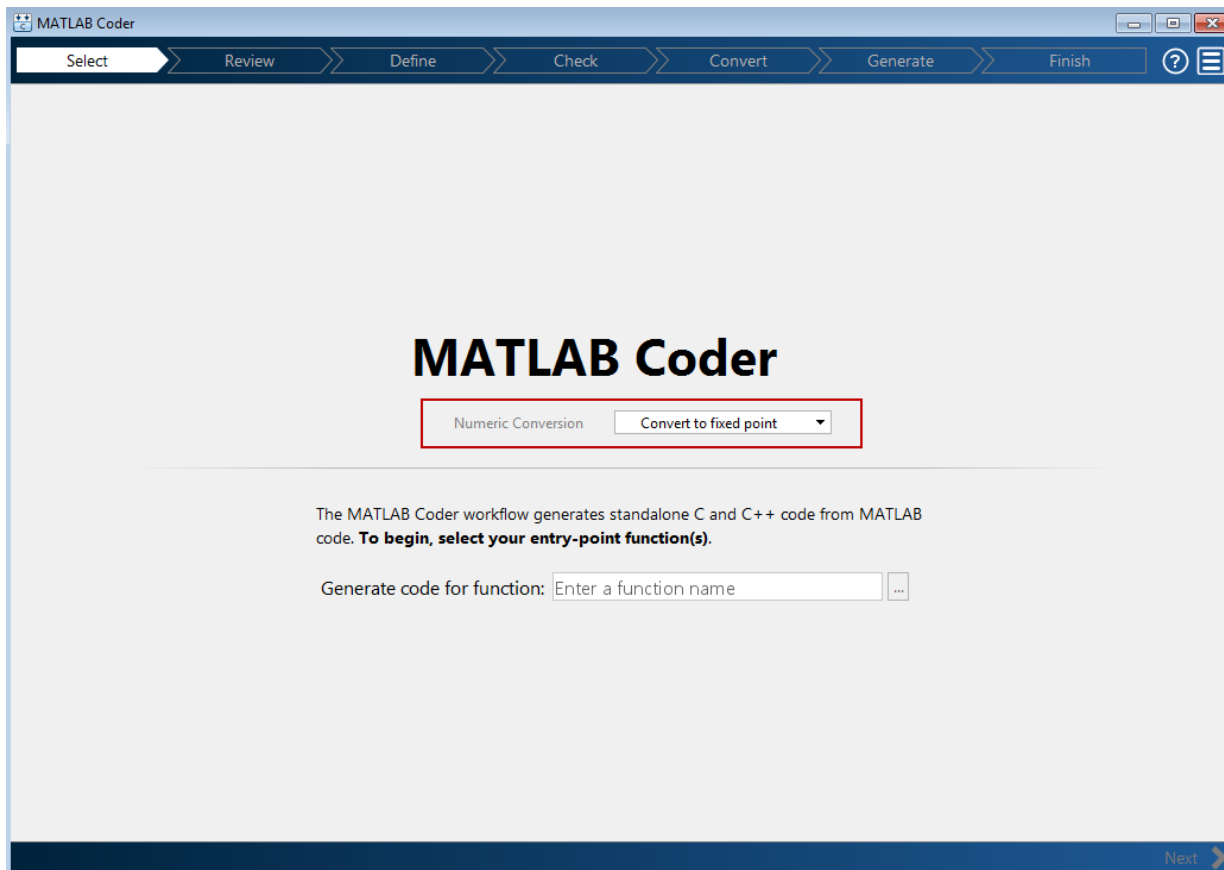
disp('Test complete.');
```

Open the MATLAB Coder App

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Fixed-Point Conversion

Set **Numeric Conversion** to **Convert to fixed point**.



Select Source Files

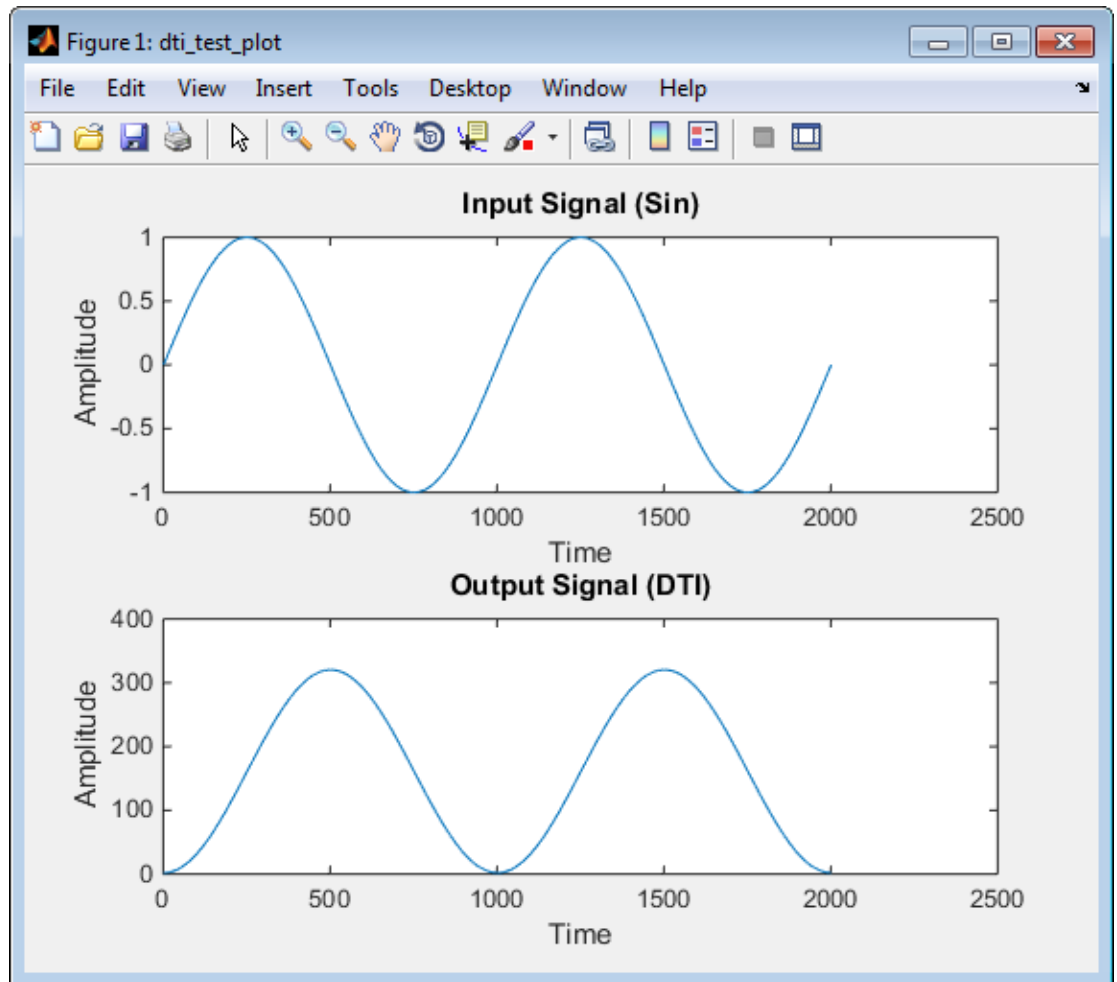
- 1 To add the entry-point function `dti` to the project, browse to the file `dti.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `dti.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `dti.m` for code violations and code generation readiness issues. The app does not find issues in `dti.m`.

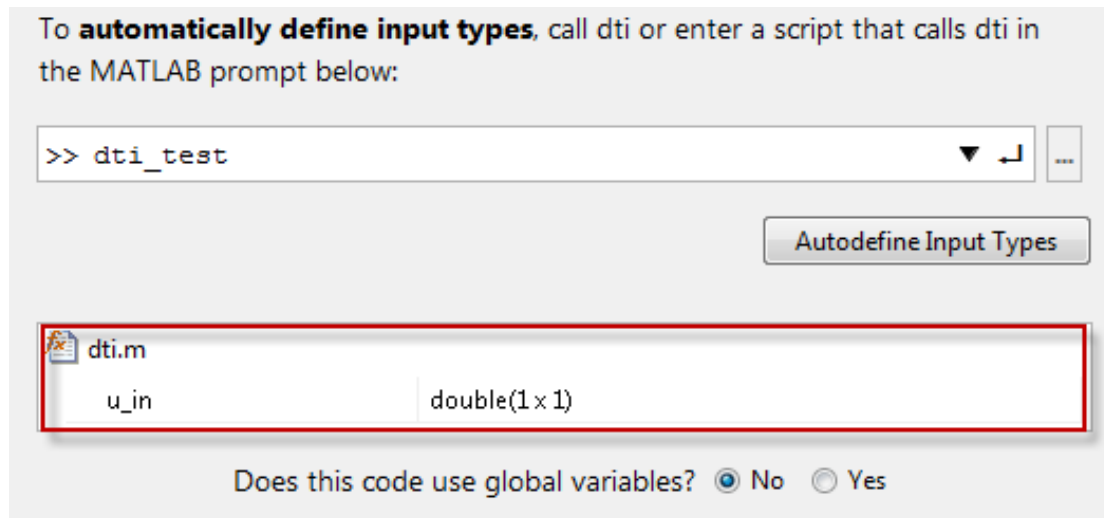
Define Input Types

- 1 On the **Define Input Types** page, to add `dti_test` as a test file, browse to `dti_test.m`, and then click **Open**.
- 2 Click **Autodefine Input Types**.

The test file runs and displays the outputs of the filter for each of the input signals.



The app determines from the test file that the input type of `u_in` is `double(1x1)`.



- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `dti_test` replacing calls to `dti` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a window where you can edit the code.

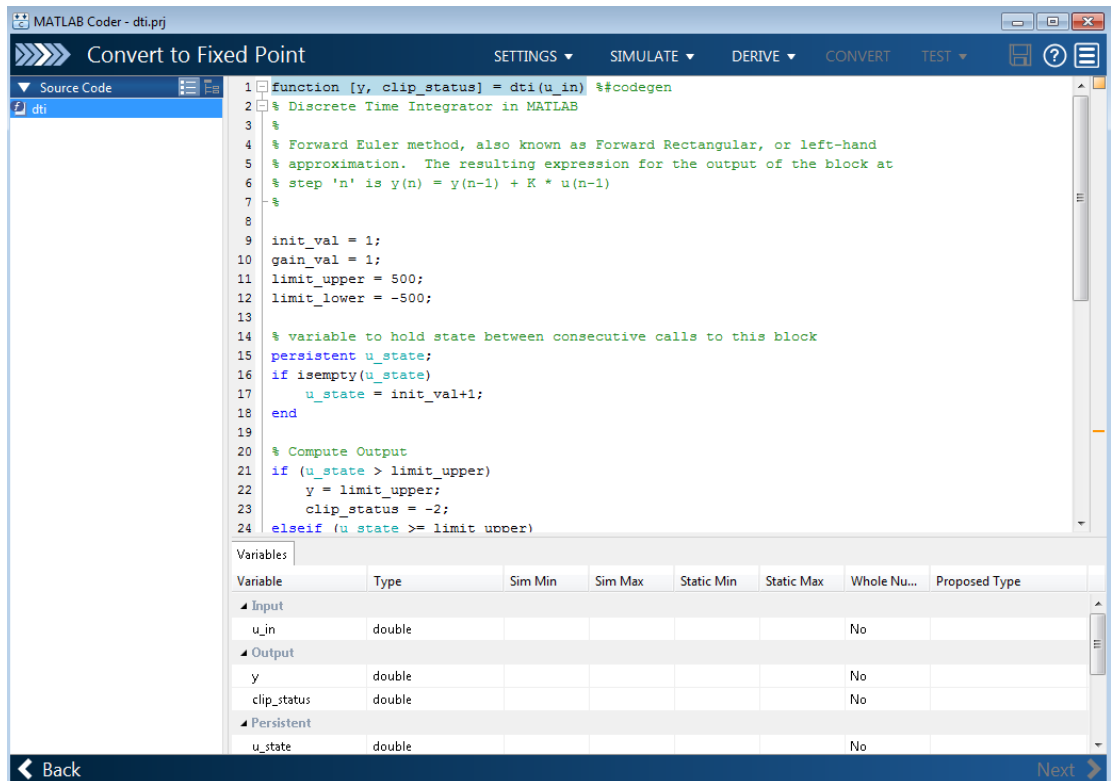
- 1 On the **Check for Run-Time Issues** page, the app populates the test file field with `dti_test`, the test file that you used to define the input types.
- 2 Click **Check for Issues**.

the app does not detect issues.

- 3 Click **Next** to go to the **Convert to Fixed Point** step.

Convert to Fixed Point

- 1 The app displays compiled information—type, size, and complexity—for variables in your code. For more information, see “View and Modify Variable Information” on page 15-85.



If functions are not supported for fixed-point conversion, the app displays them on the **Function Replacements** tab.

- 2 On the **Convert to Fixed Point** page, on the **Variables** tab, for input `u_in`, select **Static Min** and set it to -1. 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. The app uses the manually entered ranges to propose data types. You can also modify and lock the proposed type.

- 3 Click **Derive**.

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. The app displays them in the **Proposed Type** column.

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 app analyzes the function and computes these minimum and maximum values for `clip_status`.

```

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
9 init_val = 1;
10 gain_val = 1;
11 limit_upper = 500;
12 limit_lower = -500;
13
14 % variable to hold state between consecutive calls to this block
15 persistent u_state;
16 if isempty(u_state)

```

Static Analysis Output		Variables					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
▲ Input							
u_in	double			-1	1	No	numerictype(1, 16, 14)
▲ Output							
y	double			-500	500	No	numerictype(1, 16, 6)
clip_status	double			-2	2	No	numerictype(1, 16, 13)
▲ Persistent							
u_state	double			-501	501	No	numerictype(1, 16, 6)
▲ Local							
init_val	double			1	1	Yes	numerictype(0, 1, 0)
gain_val	double			1	1	Yes	numerictype(0, 1, 0)
limit_upper	double			500	500	Yes	numerictype(0, 9, 0)
limit_lower	double			-500	-500	Yes	numerictype(1, 10, 0)
tprod	double			-1	1	No	numerictype(1, 16, 14)

The app provides a **Quick derived range analysis** option and the option to specify a timeout in case the analysis takes a long time. See “Computing Derived Ranges” on page 15-94.


- 4 To convert the floating-point algorithm to fixed point, click **Convert**.

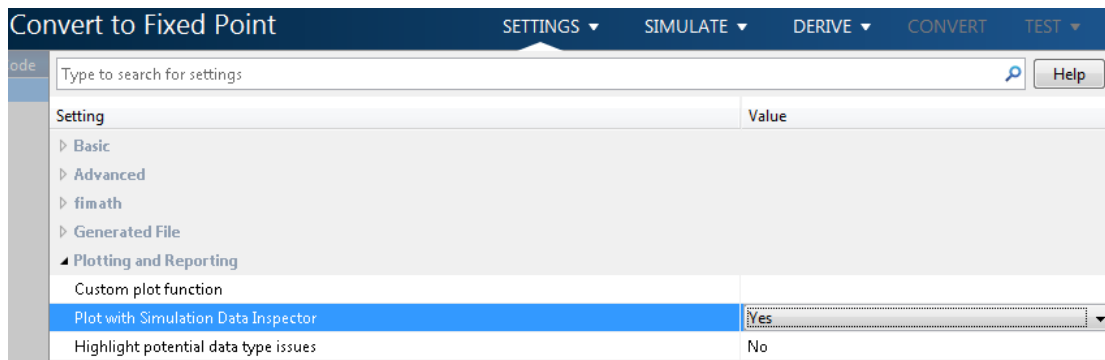
During the fixed-point conversion process, the software validates the proposed types and generates the following files in the `codegen\dti\fixpt` folder in your local working folder:

- `dti_fixpt.m` — the fixed-point version of `dti.m`.

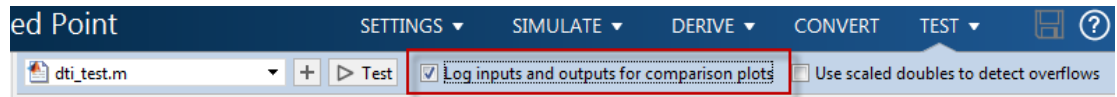
- `dti_wrapper_fixpt.m` — this file converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during conversion. The app feeds these fixed-point values into the converted fixed-point design, `dti_fixpt.m`.
- `dti_fixpt_report.html` — this report shows the generated fixed-point code and the fixed-point instrumentation results.
- `dti_report.html` — this report shows the original algorithm and the fixed-point instrumentation results.
- `dti_fixpt_exVal.mat` — MAT-file containing a structure for the input arguments, a structure for the output arguments and the name of the fixed-point file.

If errors or warnings occur during validation, they show on the **Type Validation Output** tab. See “Validating Types” on page 15-108.

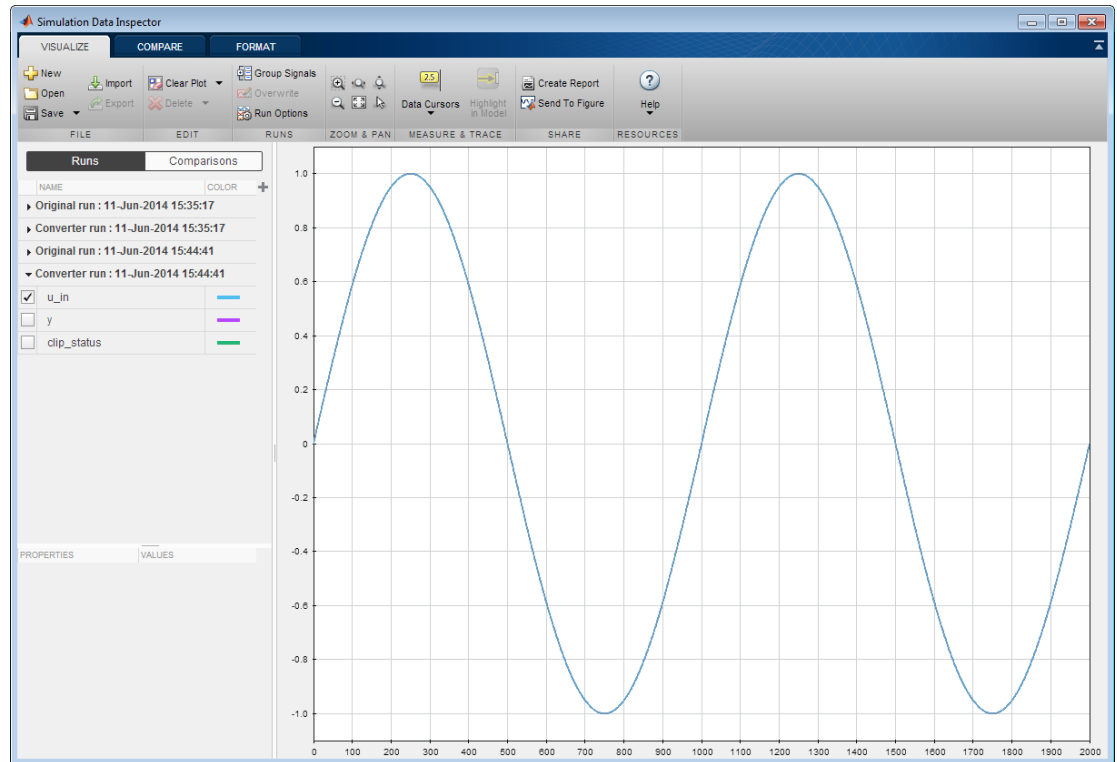
- 5 In the **Output Files** list, select `dti_fixpt.m`. The app displays the generated fixed-point code.
- 6 Use the Simulation Data Inspector to plot the floating-point and fixed-point results.
 - a Click the **Settings** arrow .
 - b Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to Yes.



- c Click the **Test** arrow . Select **Log inputs and outputs for comparison plots**. Click **Test**.



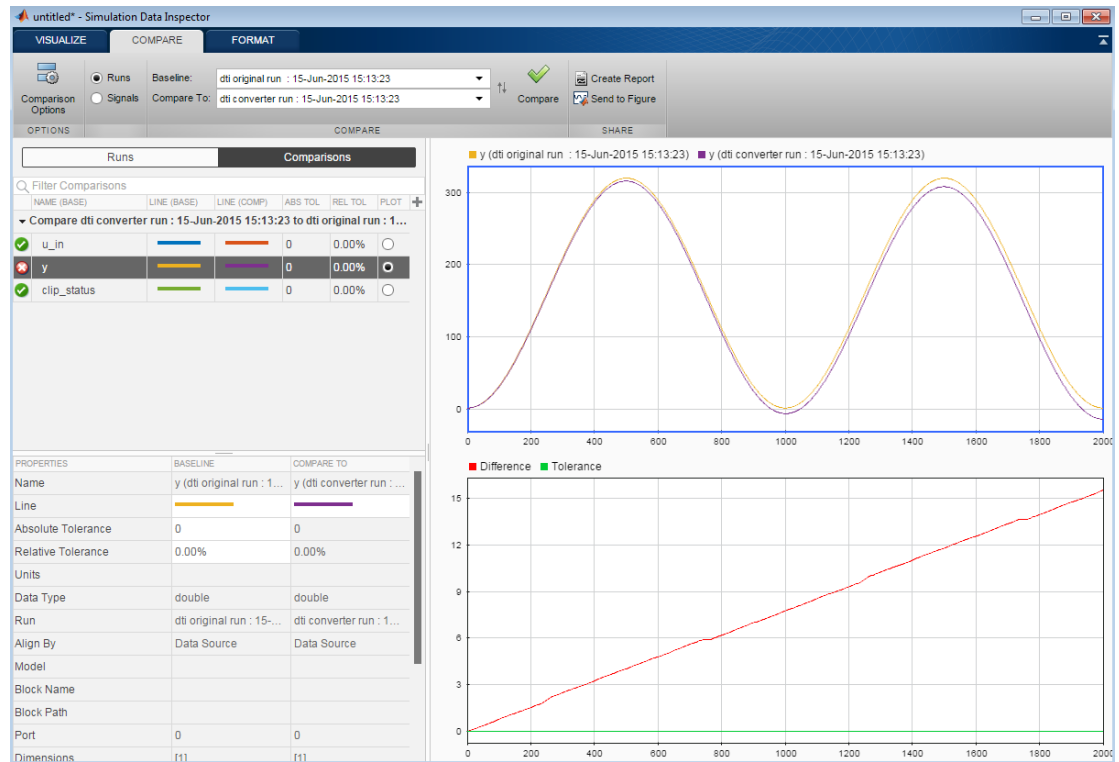
The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the 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.



- d** 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 . Select **Runs** and then click **Compare**.

The Simulation Data Inspector displays a plot of the baseline floating-point run against the fixed-point run and the difference between them.



- 7 On the **Verification Output** tab, the app provides a link to a type proposal report.

The screenshot displays the MATLAB IDE interface. The top pane shows the source code for the function `dti_fixpt`. The code implements a discrete-time integrator using the Forward Euler method. It includes comments explaining the method and the resulting expression for the output of the block at step 'n'. The code also sets up fixed-point data types using `fi` and `fimath` for various parameters and state variables. A persistent state variable `u_state` is used to hold state between consecutive calls to the block.

```

6 function [y, clip_status] = dti_fixpt(u_in) %#codegen
7 %# Discrete Time Integrator in MATLAB
8 %
9 % Forward Euler method, also known as Forward Rectangular, or left-hand
10 % approximation. The resulting expression for the output of the block at
11 % step 'n' is y(n) = y(n-1) + K * u(n-1)
12 %
13
14 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision',
15
16 init_val = fi(1, 0, 1, 0, fm);
17 gain_val = fi(1, 0, 1, 0, fm);
18 limit_upper = fi(500, 0, 9, 0, fm);
19 limit_lower = fi(-500, 1, 10, 0, fm);
20
21 % variable to hold state between consecutive calls to this block
22 persistent u_state;
23 if isempty(u_state)
24     u_state = fi(init_val+fi(1, 0, 1, 0, fm), 1, 16, 6, fm);
25 end
26

```

The bottom pane shows the verification output, which includes the following text:

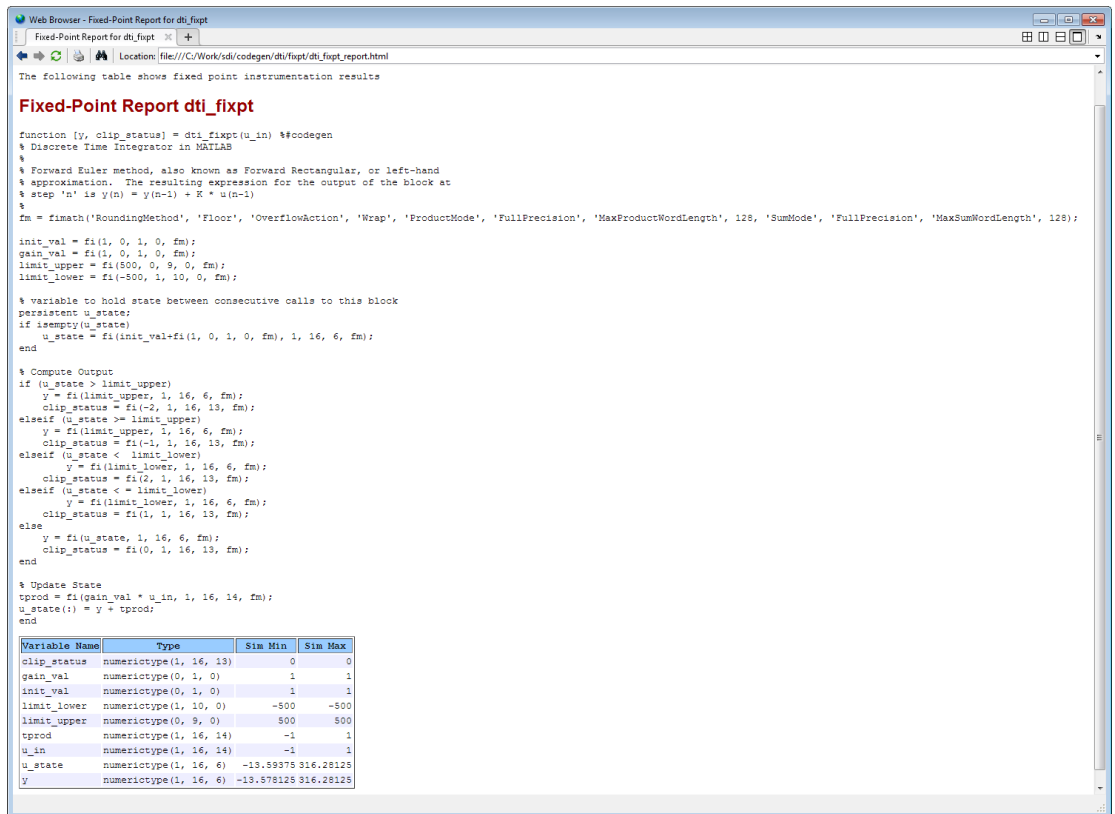
```

Verification Output | Variables | Function Replacements
Generating comparison plot...
----- Output variable : y -----
Generating comparison plot...
----- Output variable : clip_status -----
Generating comparison plot...

### Fixed Point Simulation Completed in 31.1900 sec(s)
### Generating Fixed-point Types Report for 'dti_fixpt' dti\_fixpt\_report.html
### Elapsed Time: 42.4876 sec(s)

```

To open the report, click the `dti_fixpt_report.html` link.



The following table shows fixed point instrumentation results

Fixed-Point Report dti_fixpt

```

function [y, clip_status] = dti_fixpt(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)
%
fm = fimach('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

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(1) = 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

8 Click **Next** to go to the **Generate Code** step.

Generate Fixed-Point C Code

- 1 In the **Generate** dialog box, set **Build source** to **Fixed-Point** and **Build type** to **Source Code**.
- 2 Set **Language** to **C**.
- 3 Click **Generate** 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 The app displays the generated code for `dti_fixpt.c`. In the generated C code, variables are assigned fixed-point data types.

- 5 Click **Next** to go to the **Finish Workflow** page.

On the **Finish Workflow** page, the app displays a project summary and links to generated output files.

Specify Type Proposal Options

To view type proposal options, in the MATLAB Coder app, on the **Convert to Fixed Point** page, click the **Settings** arrow .

The following options are available.

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.
Note: Manually-entered static ranges always take precedence over simulation ranges.	ignore derived ranges	Propose data types based on simulation ranges.

Advanced Type Proposal Settings	Values	Description
	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 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.

Advanced Type Proposal Settings	Values	Description
Safety margin for sim min/max (%)	0 (default)	<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>

fimath Settings	Values	Description
Rounding method	Ceiling	<p>Specify the <code>fimath</code> properties for the generated fixed-point data types.</p> <p>The default fixed-point math properties use the <code>Floor</code> rounding and <code>Wrap</code> overflow because they are the default actions in C. These settings generate the most efficient code but might cause problems with overflow.</p> <p>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.</p>
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	

fimath Settings	Values	Description
		For more information on <code>fimath</code> properties, see “ <code>fimath Object Properties</code> ”.

Generated File Settings	Value	Description
Generated fixed-point file name suffix	<code>_fixpt</code> (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 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 MATLAB Coder app. At the numerical testing stage in the conversion process, you choose to simulate the fixed-point code using scaled doubles. The app then reports which expressions in the generated code produce values that overflow the fixed-point data type.

Prerequisites

This example requires the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB) See http://www.mathworks.com/support/compilers/current_release/.

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\overflow`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
```

- 3 Copy the `overflow.m` and `overflow_test.m` files to your local working folder.

Type	Name	Description
Function code	<code>overflow.m</code>	Entry-point MATLAB function
Test file	<code>overflow_test.m</code>	MATLAB script that tests <code>overflow.m</code>

The overflow Function

```
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

```

The overflow_test Function

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);
end
```

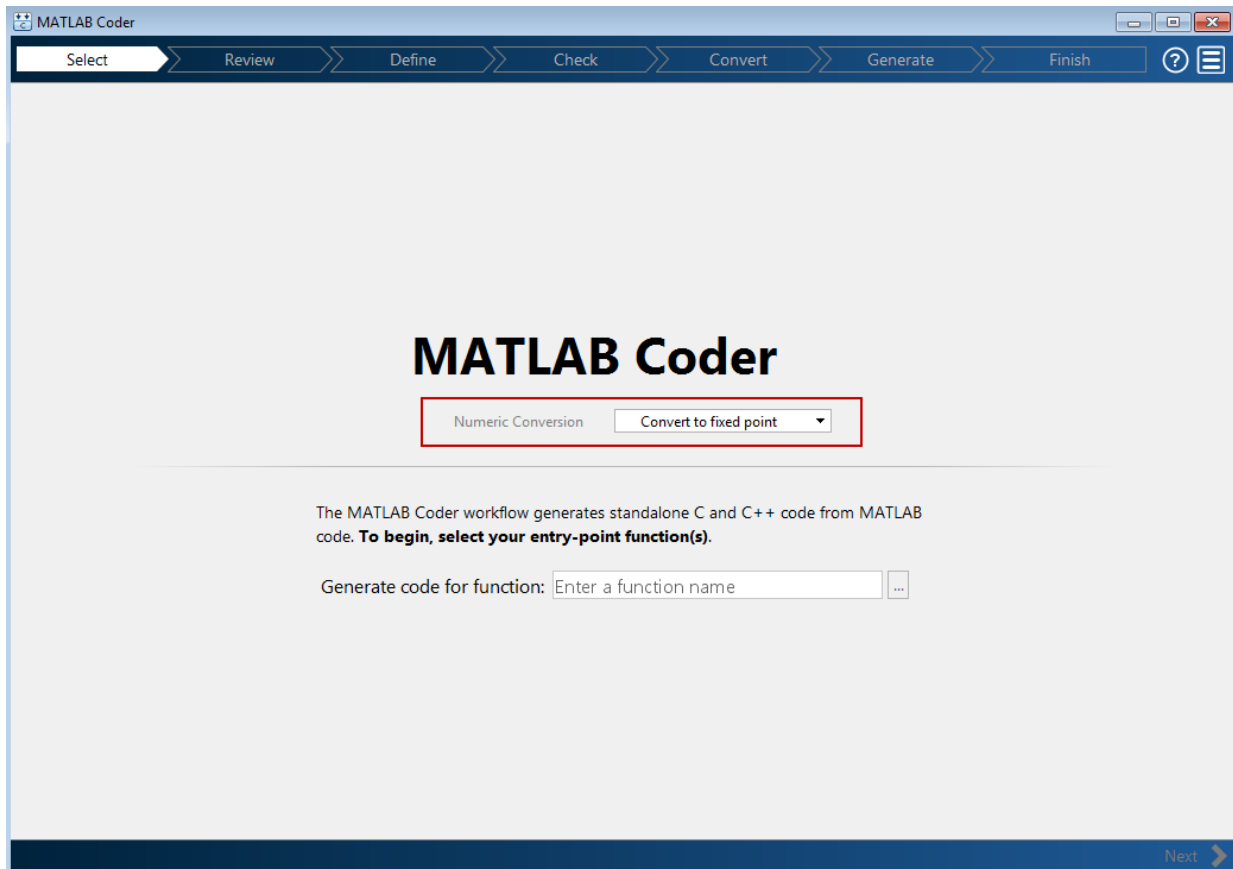
```
        title(titles{i}, 'FontSize', font_size);  
        ax = gca;  
        ax.FontSize = 10;  
    end  
    figure(gcf)  
end
```

Open the MATLAB Coder App

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Fixed-Point Conversion

Set **Numeric Conversion** to **Convert to fixed point**.



Select Source Files

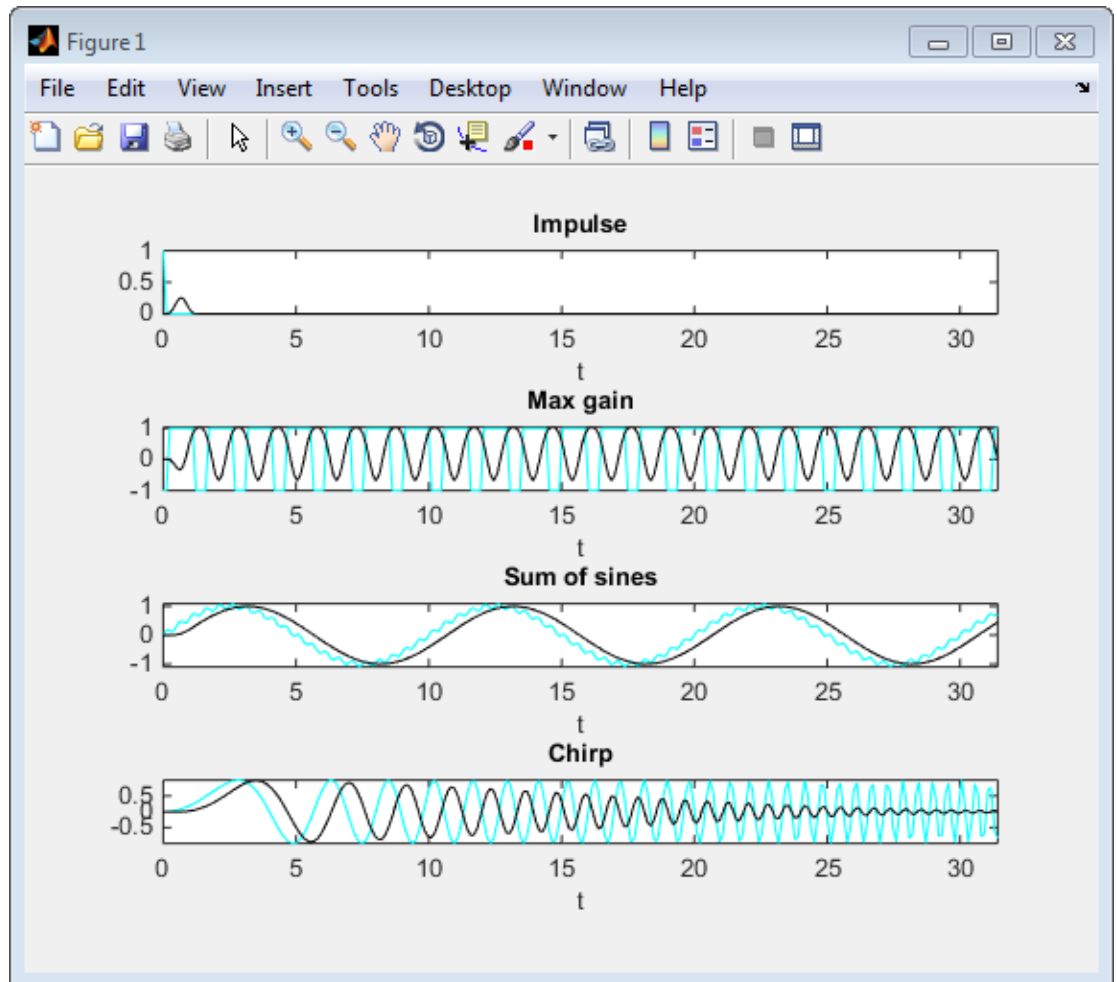
- 1 To add the entry-point function `overflow` to the project, browse to the file `overflow.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `overflow.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `overflow.m` for code violations and code generation readiness issues. The app does not find issues in `overflow.m`.

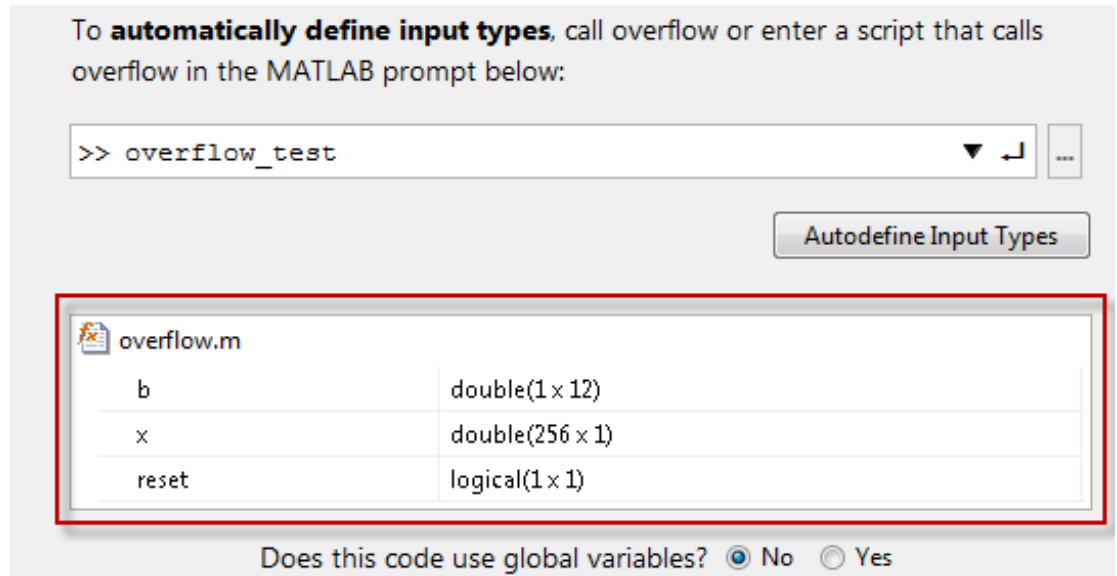
Define Input Types

- 1 On the **Define Input Types** page, to add `overflow_test` as a test file, browse to `overflow_test.m`, and then click **Open**.
- 2 Click **Autodefine Input Types**.

The test file runs and displays the outputs of the filter for each of the input signals.



The app determines from the test file that the input type of `b` is `double(1x12)`, `x` is `double(256x1)`, and `reset` is `logical(1x1)`.



- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `overflow_test` replacing calls to `overflow` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a pane where you can edit the code.

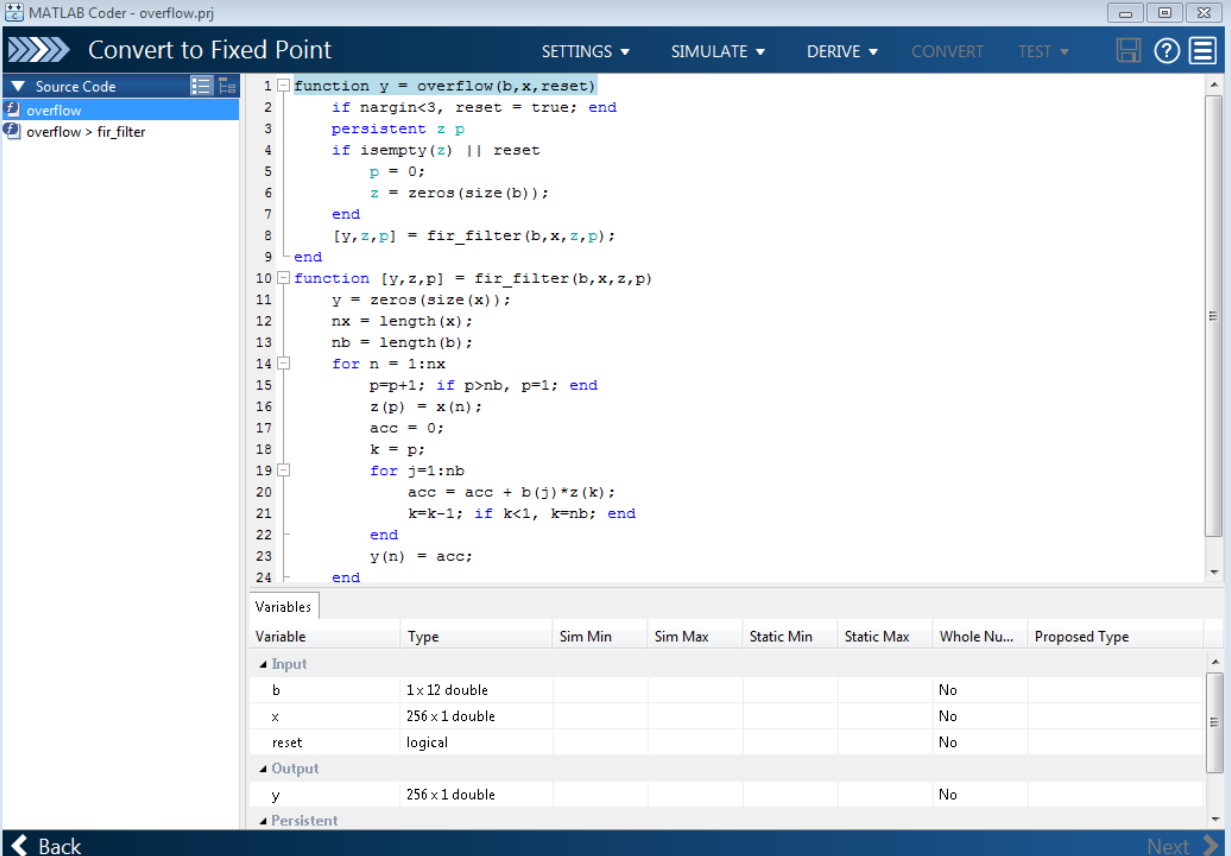
- 1 On the **Check for Run-Time Issues** page, the app populates the test file field with `overflow_test`, the test file that you used to define the input types.
- 2 Click **Check for Issues**.

The app does not detect issues.

- 3 Click **Next** to go to the **Convert to Fixed Point** step.

Convert to Fixed Point


- 1 The app displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information” on page 15-85.

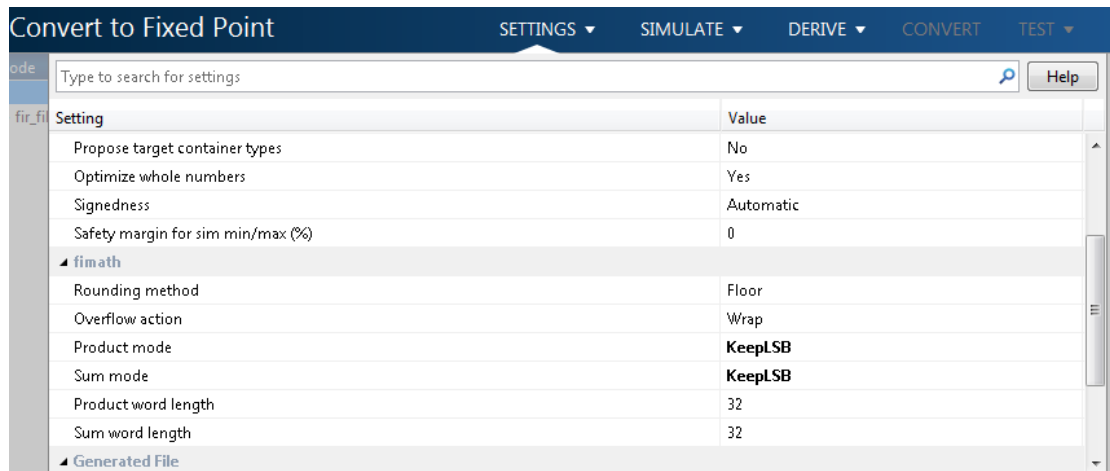


The screenshot shows the MATLAB Coder interface for the 'Convert to Fixed Point' app. The main window displays the source code for two functions: 'overflow' and 'fir_filter'. The 'Variables' table at the bottom provides detailed information about the variables used in the code.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
b	1 x 12 double					No	
x	256 x 1 double					No	
reset	logical					No	
Output							
y	256 x 1 double					No	
Persistent							

On the **Function Replacements** tab the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 15-94.

- 2 To view the advanced settings, click the **Settings** arrow . Set the fimath **Product mode** and **Sum mode** to KeepLSB. These settings model the behavior of integer operations in the C language.



3 Click **Simulate**.

The test file, `overflow_test`, runs. The app 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.


Simulation Output							
Variables		Function Replacements					
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
[-] Input							
b	1 x 12 double	0	0.25			No	numerictype(1, 16, 17)
x	256 x 1 double	-1.1	1.09			No	numerictype(1, 16, 14)
[-] Output							
y	256 x 1 double	-1	1.04			No	numerictype(1, 16, 14)
[-] Input-Output							
z	1 x 12 double	-1.1	1.09			No	numerictype(1, 16, 14)
p	double	0	13			Yes	numerictype(0, 4, 0)
[-] Local							
nx	double	256	256			Yes	numerictype(0, 9, 0)
nb	double	12	12			Yes	numerictype(0, 4, 0)
n	double	1	256			Yes	numerictype(0, 9, 0)
acc	double	-1	1.04			No	numerictype(1, 16, 14)
k	double	0	12			Yes	numerictype(0, 4, 0)
j	double	1	12			Yes	numerictype(0, 4, 0)

- 4 To convert the floating-point algorithm to fixed point, click **Convert**.

The software validates the proposed types and generates a fixed-point version of the entry-point function.

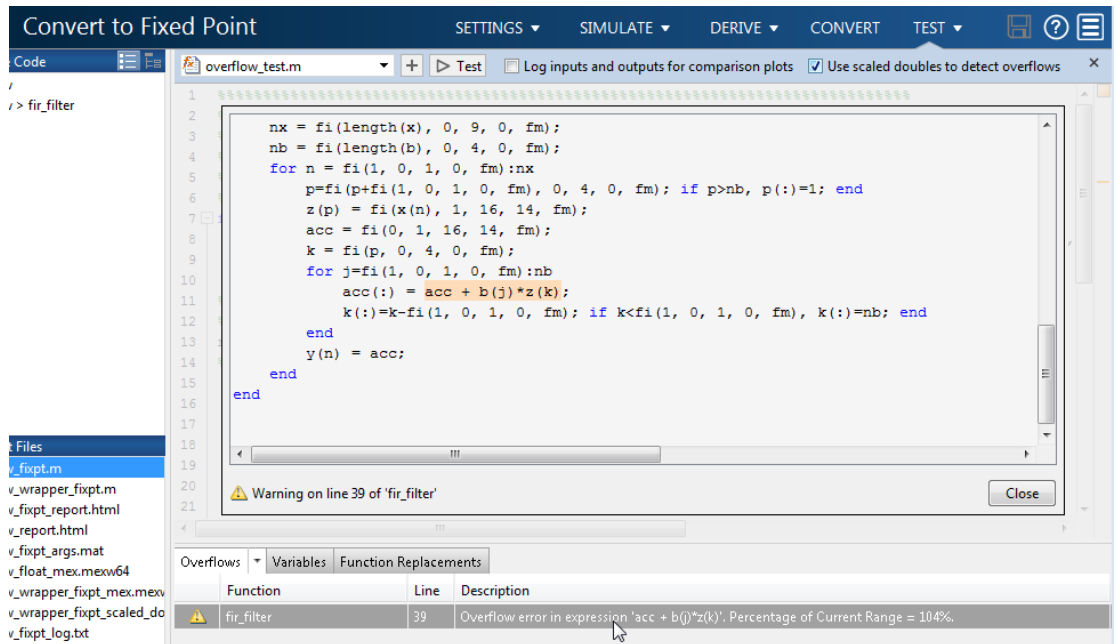
If errors and warnings occur during validation, the app displays them on the **Type Validation Output** tab. See “Validating Types” on page 15-108.

Test Numerics and Check for Overflows

- 1 Click the **Test** arrow . Verify that the test file is `overflow_test.m`. Select **Use scaled doubles to detect overflows**, and then click **Test**.

The app 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. Because they retain their fixed-point settings, they can report when a computation goes out of the range of the fixed-point type.

The simulation runs. The app detects an overflow. The app reports the overflow on the **Overflow** tab. To highlight the expression that overflowed, click the overflow.



- 2 Determine whether it was the sum or the multiplication that overflowed.

In the **Advanced** settings, set **Product mode** to **FullPrecision**, and then repeat the conversion and test the fixed-point code again.

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 MATLAB Coder app.

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). See http://www.mathworks.com/support/compilers/current_release/.

You can use `mex -setup` to change the default compiler. See “Change 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

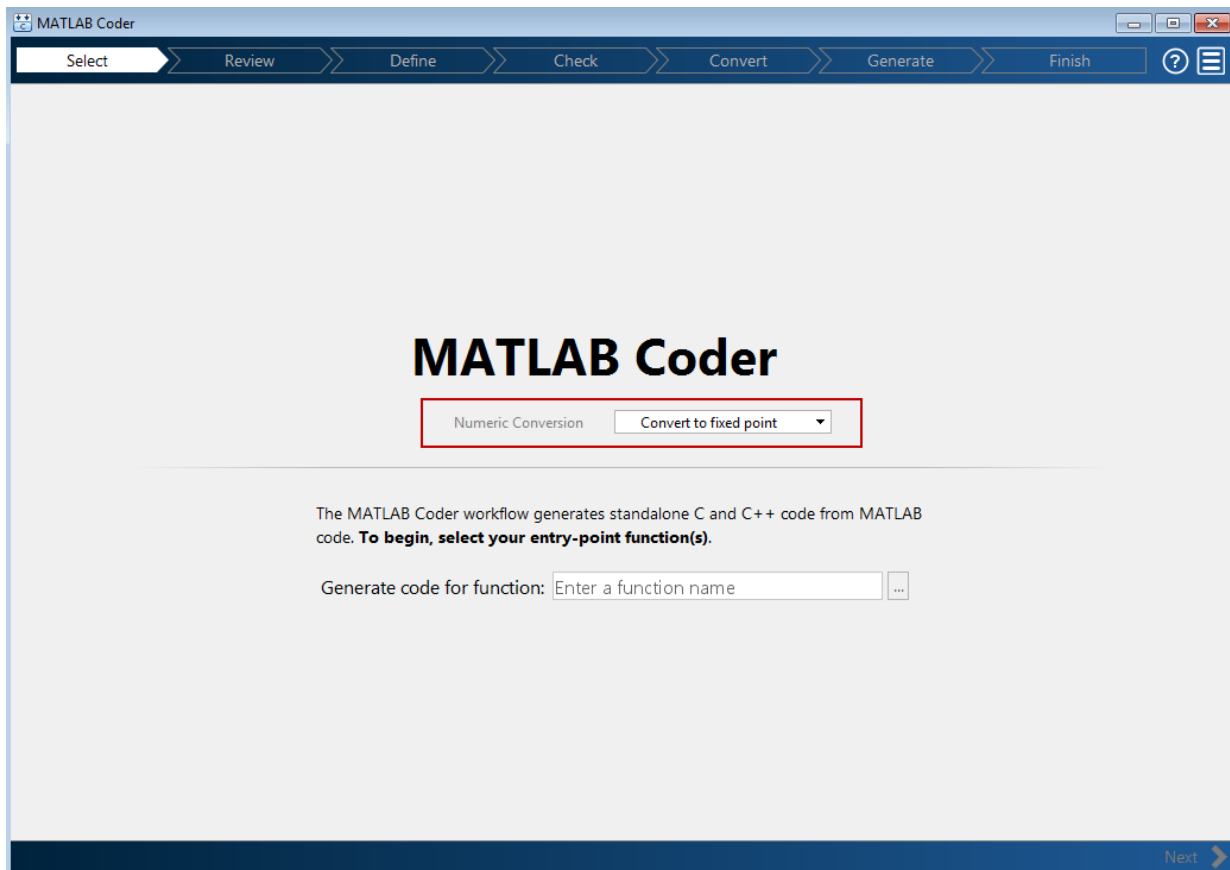
x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Open the MATLAB Coder App

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Fixed-Point Conversion

Set **Numeric Conversion** to **Convert to fixed point**.



Select Source Files

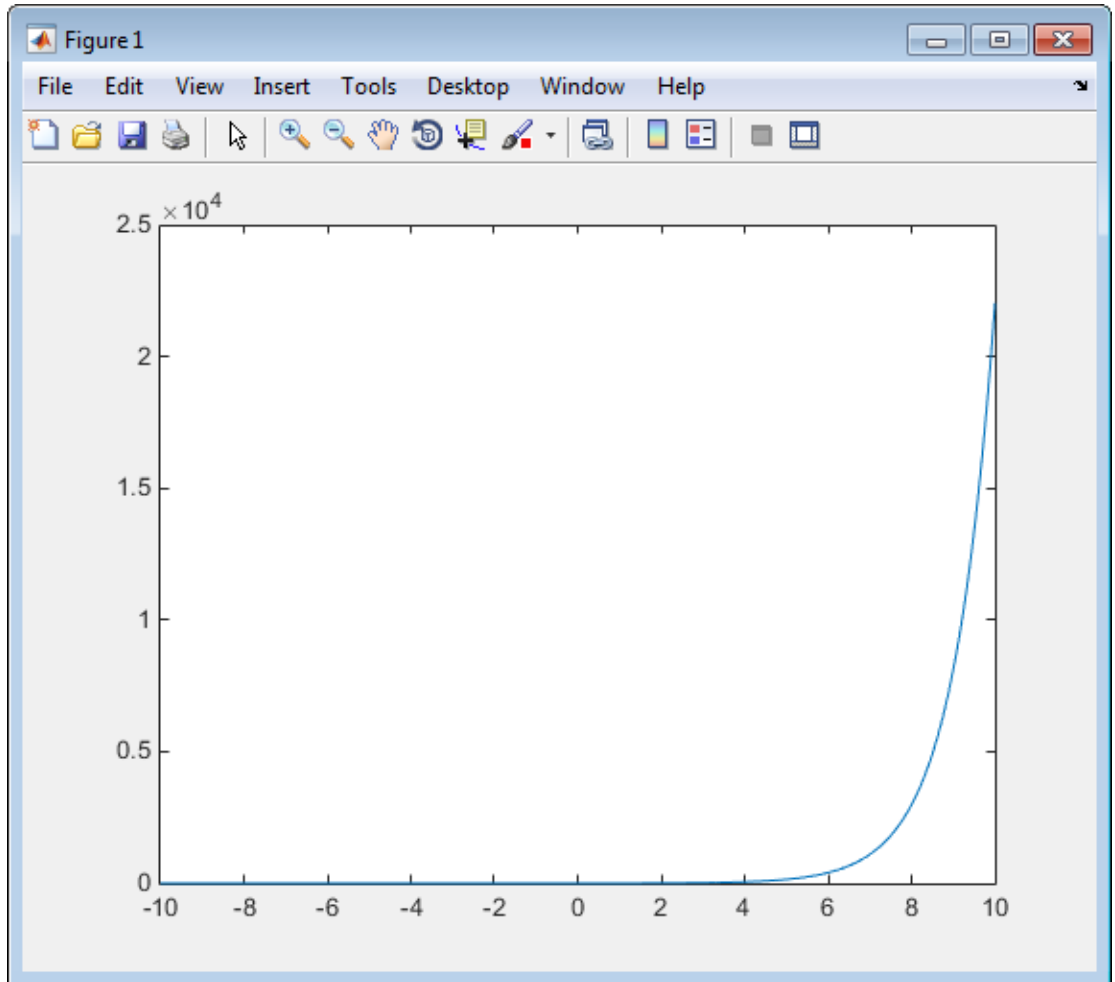
- 1 To add the entry-point function `my_fcn` to the project, browse to the file `my_fcn.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `my_fcn.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `my_fcn.m` for code violations and code generation readiness issues. The app does not find issues.

Define Input Types

- 1 Add `my_fcn_test` as a test file and then click **Autodefine Input Types**.

The test file runs and plots the output. The app determines from the test file that x is a scalar double.



- 2 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates an instrumented MEX function. It runs the test file `my_fcn_test` replacing calls to `my_fcn` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a pane where you can edit the code.

- 1 On the **Check for Run-Time Issues** page, the app populates the test file field with `my_fcn_test`, the test file that you used to define the input types.
- 2 Click **Check for Issues**.

The app does not detect issues.

- 3 Click **Next** to go to the **Convert to Fixed Point** step.

Replace `exp` Function with Lookup Table

- 1 Select the **Function Replacements** tab.

The app indicates that you must replace the `exp` function.

```

1 function y = my_fcn(x)
2     y = exp(x);
3 end
4
5

```

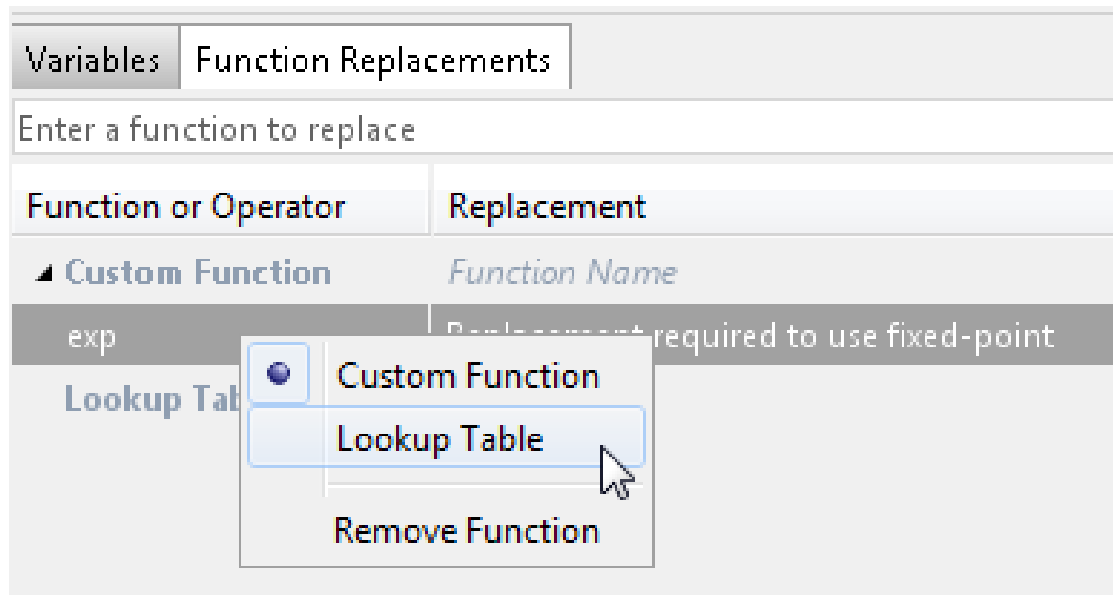
Variables | **Function Replacements**

Enter a function to replace Custom Function ▾ + -

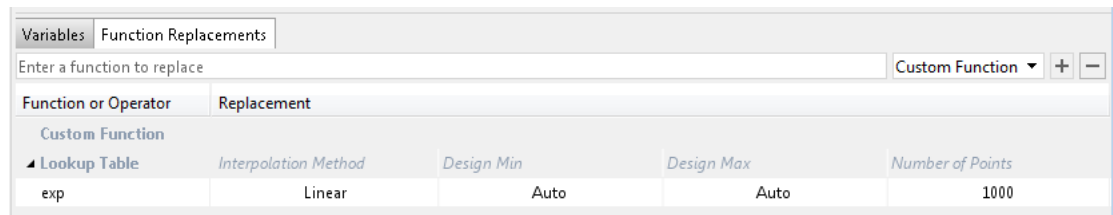
Function or Operator	Replacement
▾ Custom Function exp	Replacement required to use fixed-point
Lookup Table	

Next >

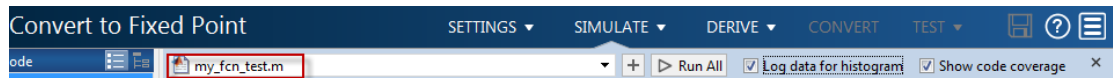
- 2 On the **Function Replacements** tab, right-click the `exp` function and select Lookup Table.



The app 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 and 1000 points. **Design Min** and **Design Max** are set to **Auto** which means that the app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.



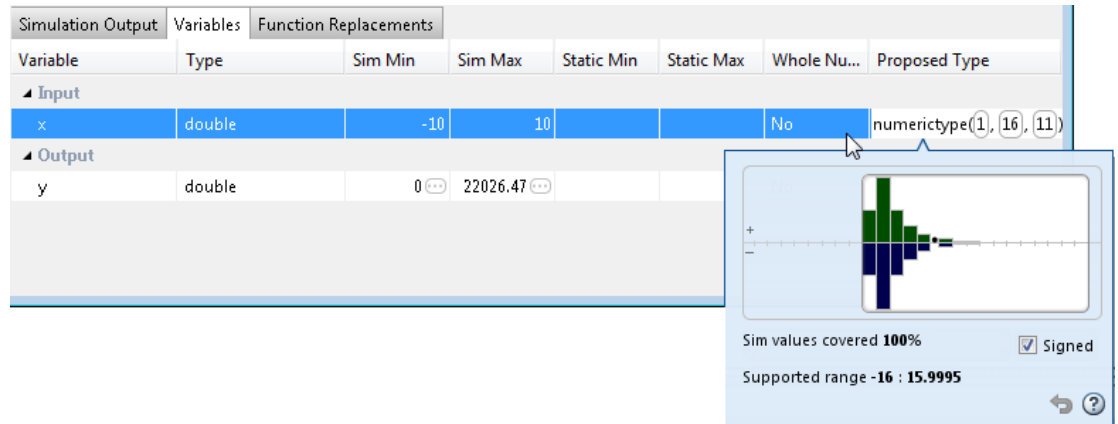
- 3 Click the **Simulate** arrow , select **Log data for histogram**, and verify that the test file is `my_fcn_test`.



4 Click **Simulate**.

The simulation runs. On the **Variables** tab, the app displays simulation minimum and maximum ranges. 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 app enables the **Convert** option.

5 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.



Convert to Fixed Point

1 Click **Convert**.

The app validates the proposed types, and generates a fixed-point version of the entry-point function, `my_fcn_fixpt.m`.

2 In the Output Files list, select `my_fcn_fixpt.m`.

The conversion process generates a lookup table approximation, `replacement_exp`, for the `exp` function.

The screenshot shows the 'Convert to Fixed Point' tool interface. The main window displays the MATLAB code for the function `replacement_exp(x)`. The code defines a persistent LUT, sets the numeric type to `fi(14, -1)`, and uses `fi` to find the minimum and maximum values of the function. It then uses linear interpolation to approximate the function value based on the LUT.

The bottom panel shows the 'Function Replacements' table:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 16, 11)
Output							
y	double	0	22026.47			No	numerictype(0, 16, 1)

The generated fixed-point function, `my_fcn_fixpt.m`, calls this approximation instead of calling `exp`. 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.

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

    y = fi(replacement_exp(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. Then, regenerate the 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 MATLAB Coder app.

Prerequisites

This example requires the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). See http://www.mathworks.com/support/compilers/current_release/.

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

Create Algorithm and Test Files

In a local, writable folder:

- 1 Create a MATLAB function, `custom_fcn.m` 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, `call_custom_fcn.m`, that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

- 3 Create a test file, `custom_test.m`, that uses `call_custom_fcn`.

```
close all
clear all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
```

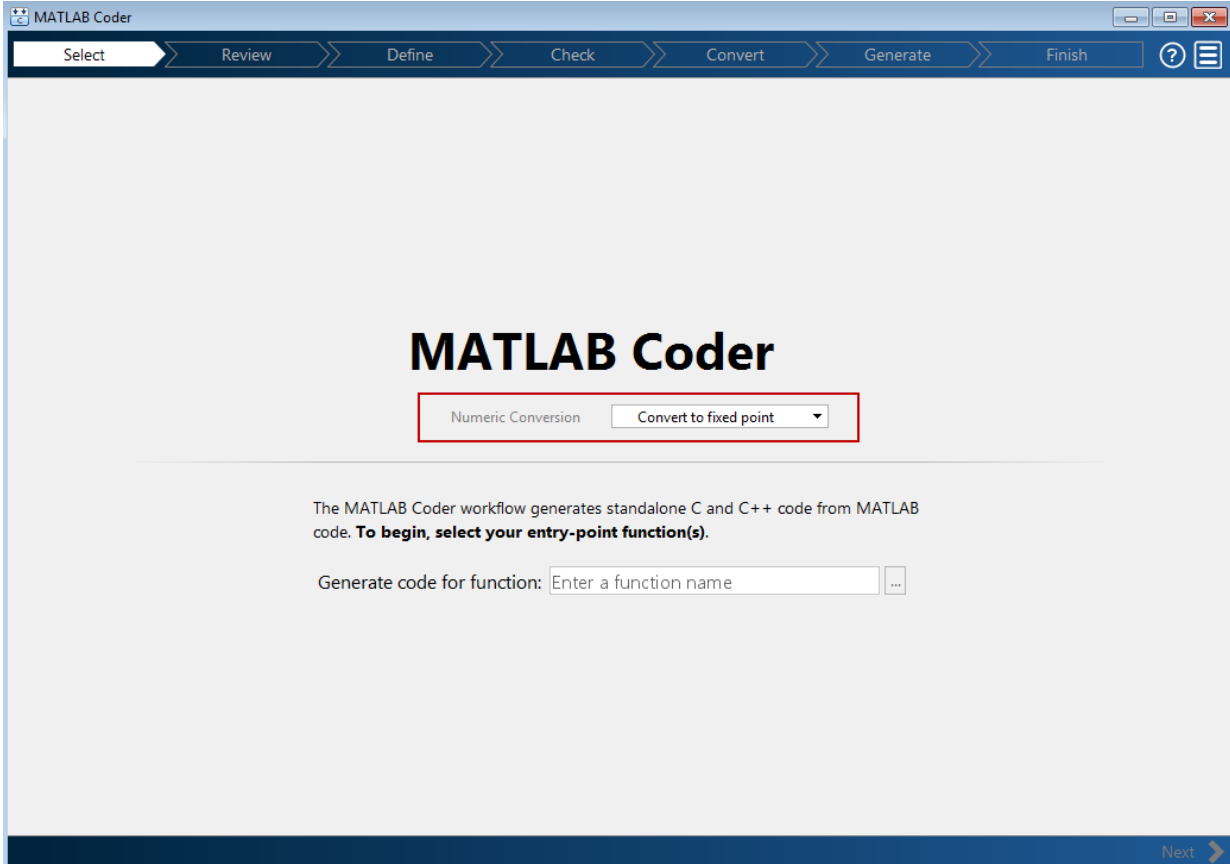
```
end  
plot( x, y );
```

Open the MATLAB Coder App

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Fixed-Point Conversion

Set **Numeric Conversion** to **Convert to fixed point**.



Select Source Files

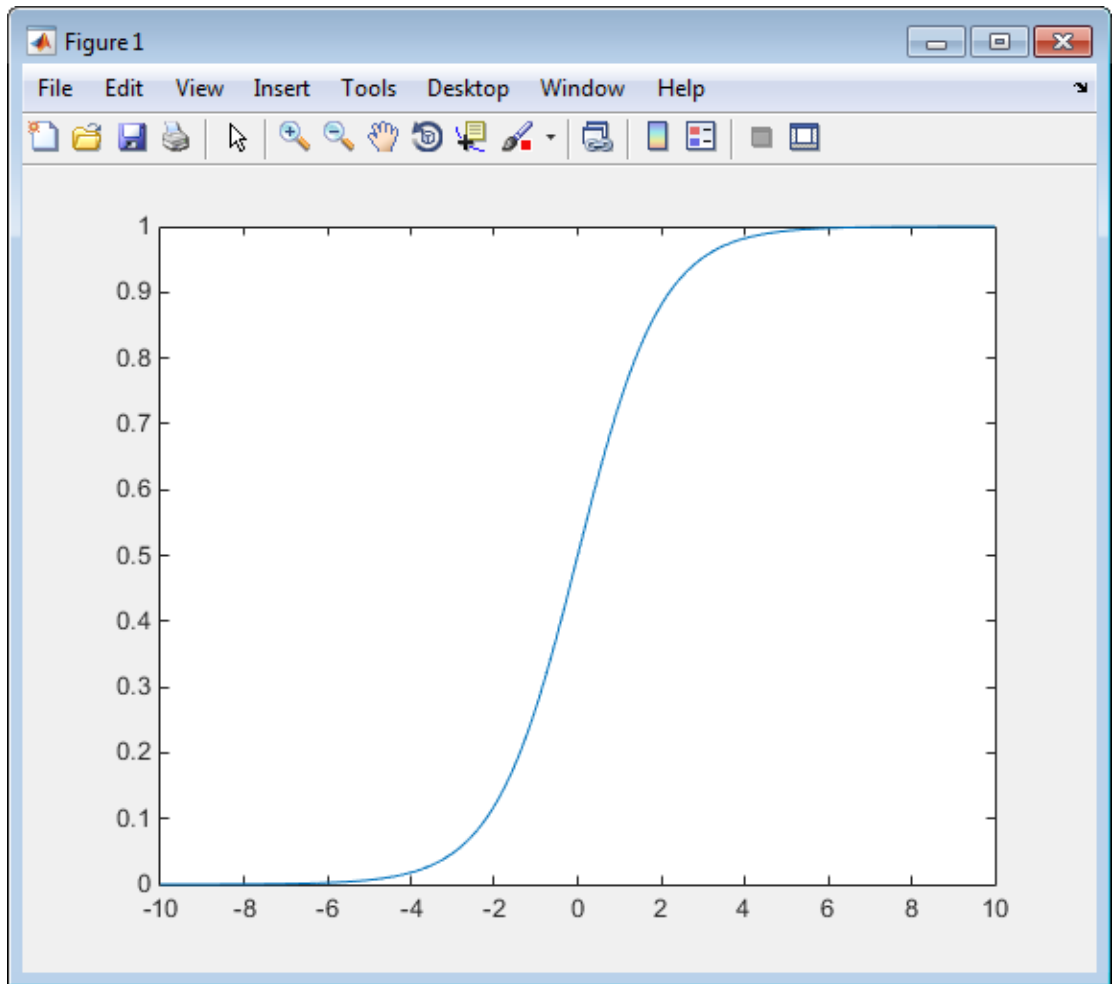
- 1 To add the entry-point function `call_custom_fcn` to the project, browse to the file `call_custom_fcn.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `call_custom_fcn.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `call_custom_fcn.m` for code violations and code generation readiness issues. The app does not find issues.

Define Input Types

- 1 Add `custom_test` as a test file and then click **Autodefine Input Types**.

The test file runs and plots the output. The app determines from the test file that `x` is a scalar double.



- 2 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `custom_test` replacing calls to `call_custom_fcn` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a pane where you can edit the code.

- 1 On the **Check for Run-Time Issues** page, the app populates the test file field with `custom_test`, the test file that you used to define the input types.
- 2 Click **Check for Issues**.

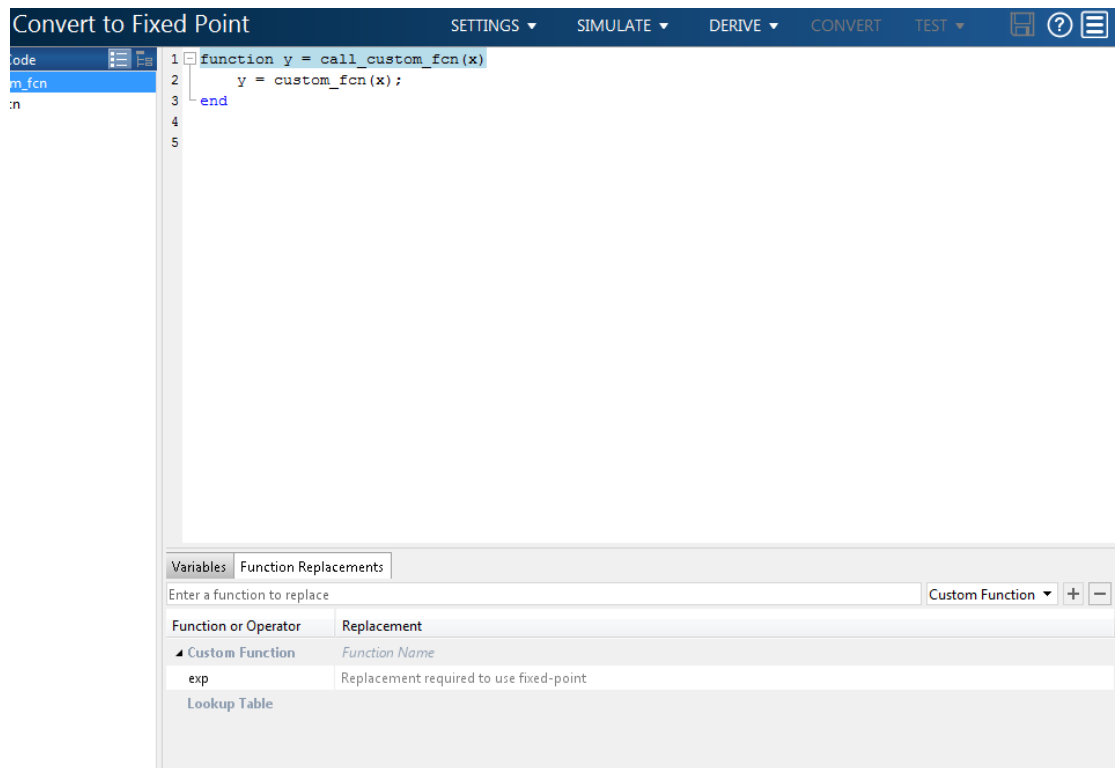
The app does not detect issues.

- 3 Click **Next** to go to the **Convert to Fixed Point** step.

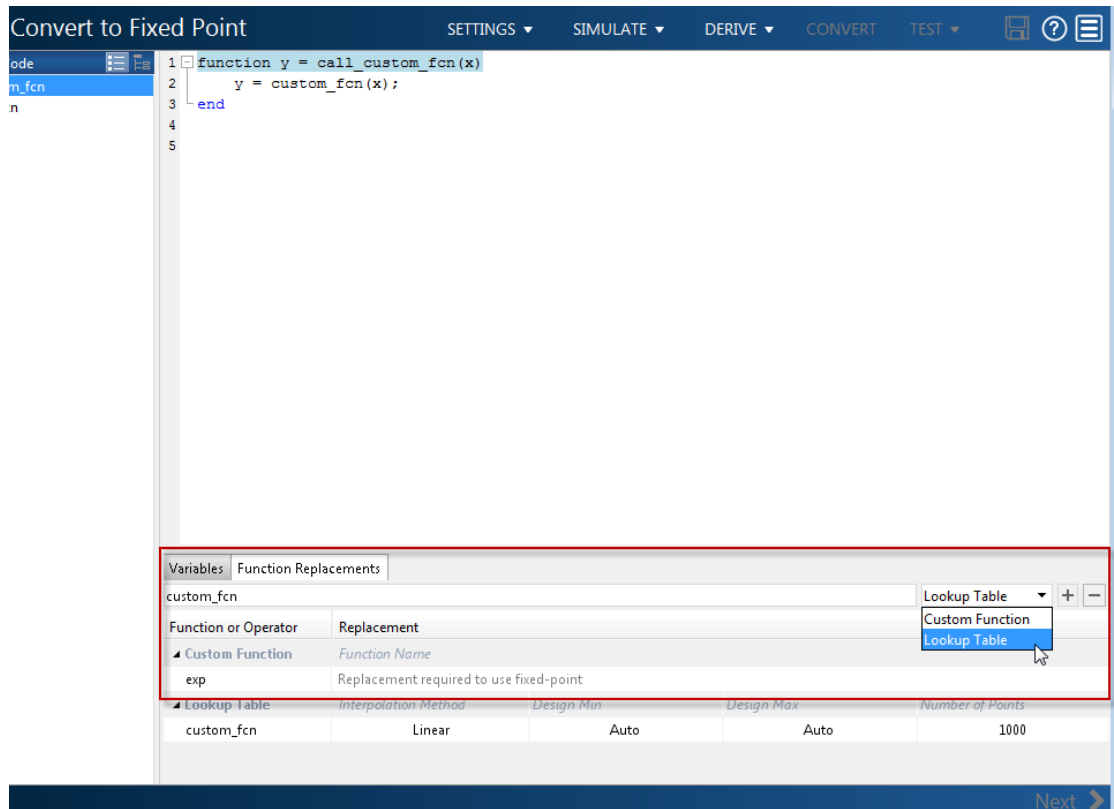
Replace `custom_fcn` with Lookup Table

- 1 Select the **Function Replacements** tab.

The app indicates that you must replace the `exp` function.




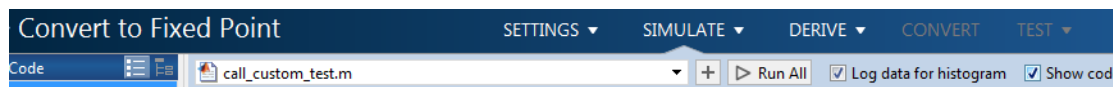
- 2 Enter the name of the function to replace, `custom_fcn`, select **Lookup Table**, and then click **+**.



The app adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. The app sets **Design Min** and **Design Max** to **Auto** which means that app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.

Function Replacements	
custom_fcn	Lookup Table + -
Function or Operator	Replacement
Custom Function <i>Function Name</i>	
exp	Replacement required to use fixed-point
Lookup Table <i>Interpolation Method</i> <i>Design Min</i> <i>Design Max</i> <i>Number of Points</i>	
custom_fcn	Linear Auto Auto 1000

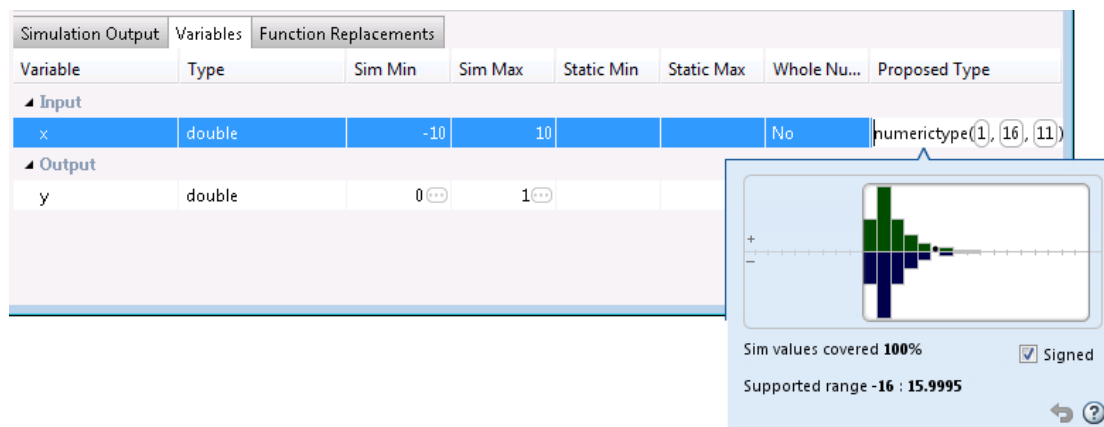
- 3 Click the **Simulate** arrow , select **Log data for histogram**, and verify that the test file is `call_custom_test`.



- 4 Click **Simulate**.

The simulation runs. The app 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 **Convert** option is now enabled.

- 5 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.



Convert to Fixed Point

- 1 Click **Convert**.

The app validates the proposed types and generates a fixed-point version of the entry-point function, `call_custom_fcn_fixpt.m`.

The screenshot displays the MATLAB environment. On the left, the 'Source Code' pane shows files `call_custom_fcn` and `custom_fcn`. Below it, the 'Output Files' list includes `call_custom_fcn_fixpt.m`, which is selected. The main editor shows the source code for `call_custom_fcn_fixpt.m` and `custom_fcn`. The `call_custom_fcn_fixpt.m` function calls `fi` on the output of `replacement_custom_fcn(x)`. The `custom_fcn` function uses `fimath` and `fi` to perform fixed-point arithmetic. Below the code, the 'Type Validation Output' table is visible, showing the inferred types and ranges for variables `x` and `y`.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1,16,11)
Output							
y	double	0	1			No	numerictype(0,16,16)

- 2 In the Output Files list, select `call_custom_fcn_fixpt.m`.

The conversion process generates a lookup table approximation, `replacement_custom_fcn`, 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. 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, `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(replacement_custom_fcn(x), 0, 16, 16, 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.

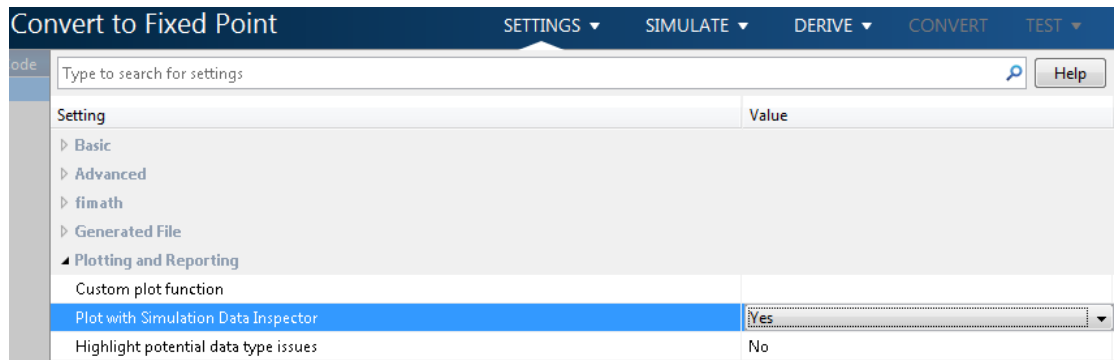
Enable Plotting Using the Simulation Data Inspector


You can use the Simulation Data Inspector with the MATLAB Coder app to inspect and compare floating-point and fixed-point logged input and output data.

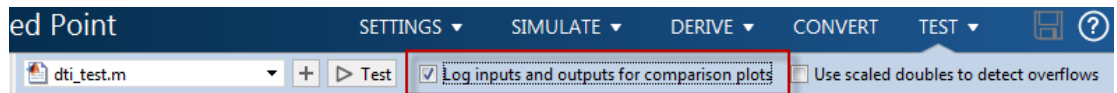
- 1 On the **Convert to Fixed Point** page,

Click the **Settings** arrow .

- 2 Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to **Yes**.



- 3 Click the **Test** arrow . Select **Log inputs and outputs for comparison plots**, and then click **Test**.



For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-23 “Propose Data Types Based on Derived Ranges”.

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the MATLAB Coder app 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 **Log inputs and outputs for comparison plots** 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

This example requires the following products:

- MATLAB
- Fixed-Point Designer
- MATLAB Coder
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). See http://www.mathworks.com/support/compilers/current_release/.

You can use `mex -setup` to change the default compiler. See “Change 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	<code>myFilter.m</code>	Entry-point MATLAB function

Type	Name	Description
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
```

The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%           i) name
```

```
%          ii) functionName
% floatVals - cell array of logged values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after
%          Fixed-Point Conversion.
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');

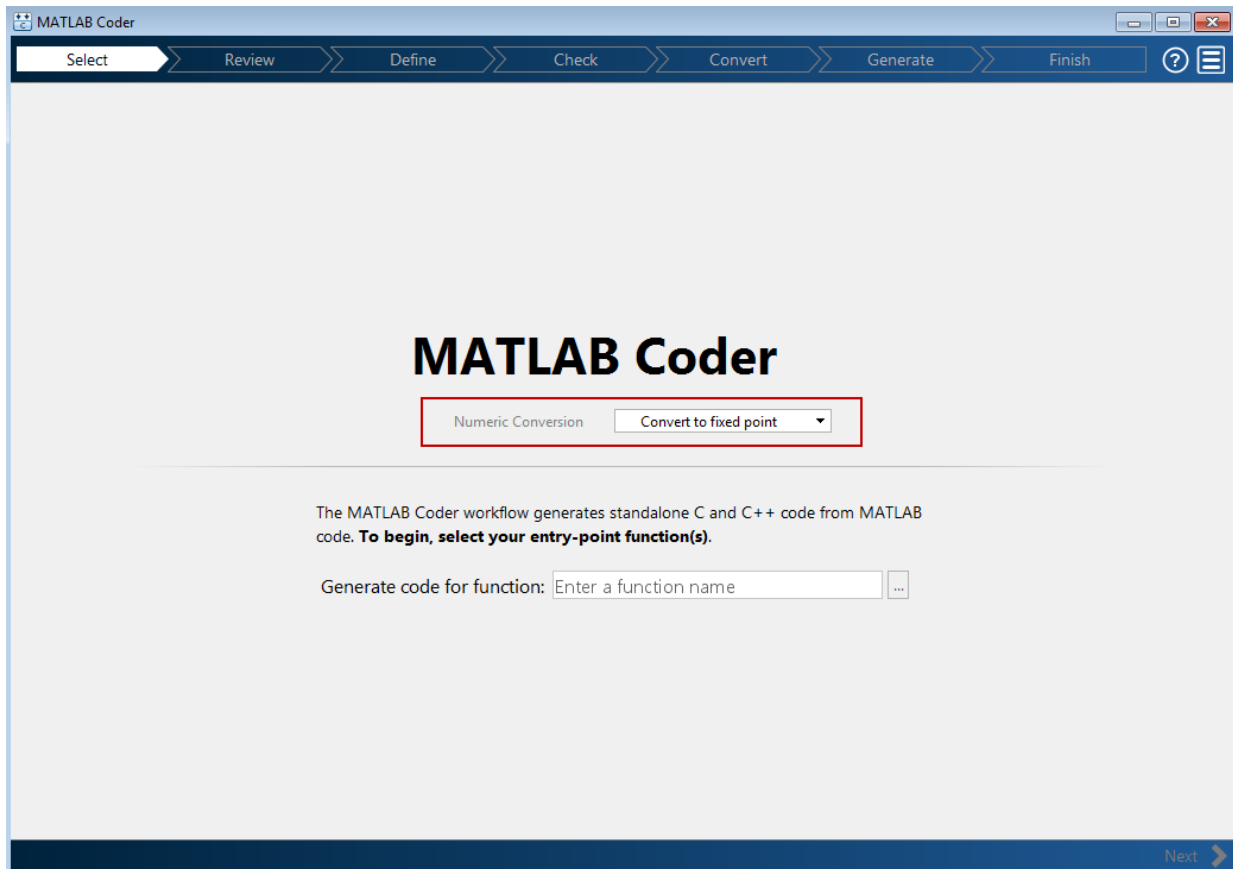
    title(figTitle);
end
```

Open the MATLAB Coder App

- 1 Navigate to the folder that contains the files for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Fixed-Point Conversion

Set **Numeric Conversion** to **Convert to fixed point**.



Select Source Files

- 1 To add the entry-point function `myFilter` to the project, browse to the file `myFilter.m`, and then click **Open**.

By default, the app saves information and settings for this project in the current folder in a file named `myFilter.prj`.

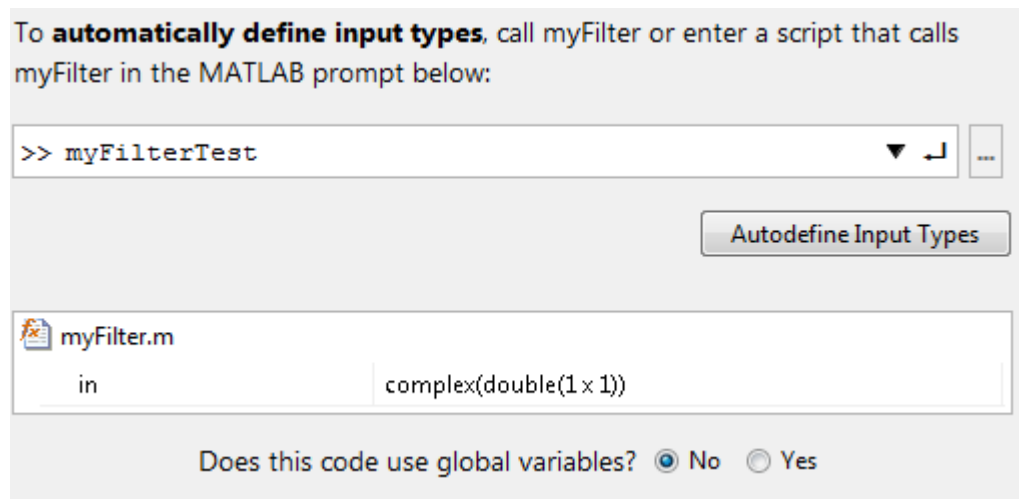
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `myFilter.m` for code violations and code generation readiness issues. The app does not find issues in `myFilter.m`.

Define Input Types

- 1 On the **Define Input Types** page, to add `myFilter_test` as a test file, browse to `myFilter_test.m`, and then click **Open**.
- 2 Click **Autodefine Input Types**.

The app determines from the test file that the input type of `in` is `complex(double(1x1))`.



- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates instrumented MEX, `myFilter`. It runs the test file `myFilterTest` replacing calls to `myFilter` with calls to the generated MEX. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a window where you can edit the code.

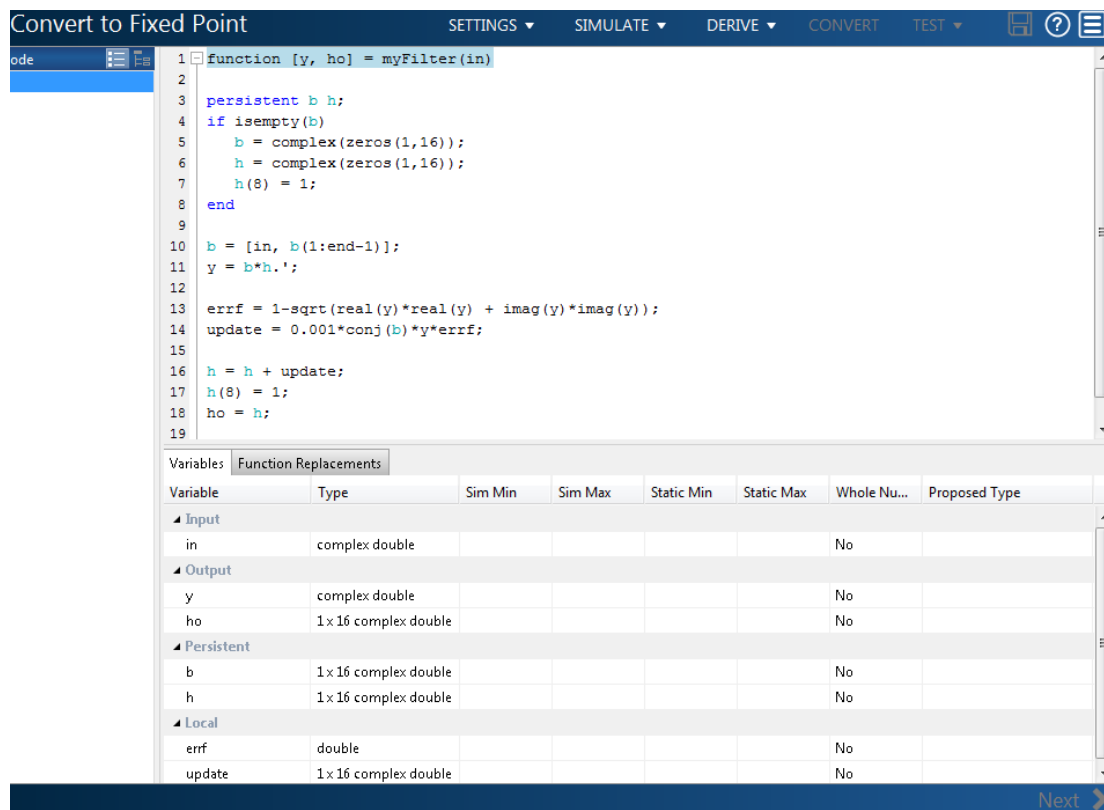
- 1 Browse to the test file `myFilterTest.m`.
- 2 Click **Check for Issues**.

The app does not detect issues.

- 3 Click **Next** to go to the **Convert to Fixed Point** step.

Convert to Fixed Point

- 1 The app displays compiled information for variables in your code. For more information, see “View and Modify Variable Information” on page 15-85 “View and Modify Variable Information”.



The screenshot shows the 'Convert to Fixed Point' dialog box in MATLAB. The top part is a code editor showing the following MATLAB code:

```



1 function [y, ho] = myFilter(in)
2
3 persistent b h;
4 if isempty(b)
5     b = complex(zeros(1,16));
6     h = complex(zeros(1,16));
7     h(8) = 1;
8 end
9
10 b = [in, b(1:end-1)];
11 y = b*h.';
12
13 errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
14 update = 0.001*conj(b)*y*errf;
15
16 h = h + update;
17 h(8) = 1;
18 ho = h;
19

```

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

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
in	complex double					No	
Output							
y	complex double					No	
ho	1x16 complex double					No	
Persistent							
b	1x16 complex double					No	
h	1x16 complex double					No	
Local							
errf	double					No	
update	1x16 complex double					No	

The dialog box has a 'Next' button at the bottom right.

- 2 To open the settings dialog box, click the **Settings** arrow .
 - a Verify that **Default word length** is set to 16.
 - b Under **Advanced**, set **Signedness** to Signed
 - c Under **Plotting and Reporting**, set **Custom plot function** to plotDiff.
- 3 Click the **Simulate** arrow . Verify that the test file is myFilterTest.

4 Click **Simulate**.

The test file, `myFilterTest`, runs and the app 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.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
in	complex double	-0.95	0.95			No	numerictype(1, 16, 15)
Output							
y	complex double	-0.95	0.95			No	numerictype(1, 16, 15)
ho	1x16 complex double	-0.13	1			No	numerictype(1, 16, 14)
Persistent							
b	1x16 complex double	-0.95	0.95			No	numerictype(1, 16, 15)
h	1x16 complex double	-0.13	1			No	numerictype(1, 16, 14)
Local							
errf	double	-0.32	1			No	numerictype(1, 16, 14)
update	1x16 complex double	0	0			No	numerictype(1, 16, 25)

5 To convert the floating-point algorithm to fixed point, click **Convert**.

The software validates the proposed types and generates a fixed-point version of the entry-point function.

The screenshot shows a MATLAB code editor window titled 'myFilterTest.m'. The code defines a function `myFilter_fixpt` that takes an input `in` and returns `y` and `ho`. The function uses fixed-point arithmetic with the `fi` function. It initializes `b` and `h` as fixed-point numbers, then processes the input `in` through a series of operations including multiplication, addition, and error calculation. The code is as follows:

```

7 function [y, ho] = myFilter_fixpt(in)
8
9 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision',
10
11 persistent b h;
12 if isempty(b)
13     b = fi(complex(zeros(1,16)), 1, 16, 15, fm);
14     h = fi(complex(zeros(1,16)), 1, 16, 14, fm);
15     h(8) = 1;
16 end
17
18 b(:) = [fi(in, 1, 16, 15, fm), b(1:end-1)];
19 y = fi(b*h.', 1, 16, 15, fm);
20
21 errf = fi(fi_signed(fi(1, 1, 2, 0, fm))-sqrt(real(y)*real(y) + imag(y)*imag(y)), 1, 16, 14, fm);
22 update = fi(fi(0.001, 1, 16, 24, fm)*conj(b)*y*errf, 1, 16, 25, fm);
23
24 h(:) = h + update;
25 h(8) = 1;
26 ho = fi(h, 1, 16, 14, fm);
27

```

Below the code editor, there are tabs for 'Type Validation Output', 'Variables', and 'Function Replacements'. The 'Type Validation Output' tab is active, showing the following text:


```

### Generating Type Proposal Report for 'myFilter' myFilter\_report.html
### Generating Fixed Point MATLAB Code myFilter\_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper myFilter\_wrapper\_fixpt
### Generating Mex file for ' myFilter_wrapper_fixpt '
Code generation successful: View report

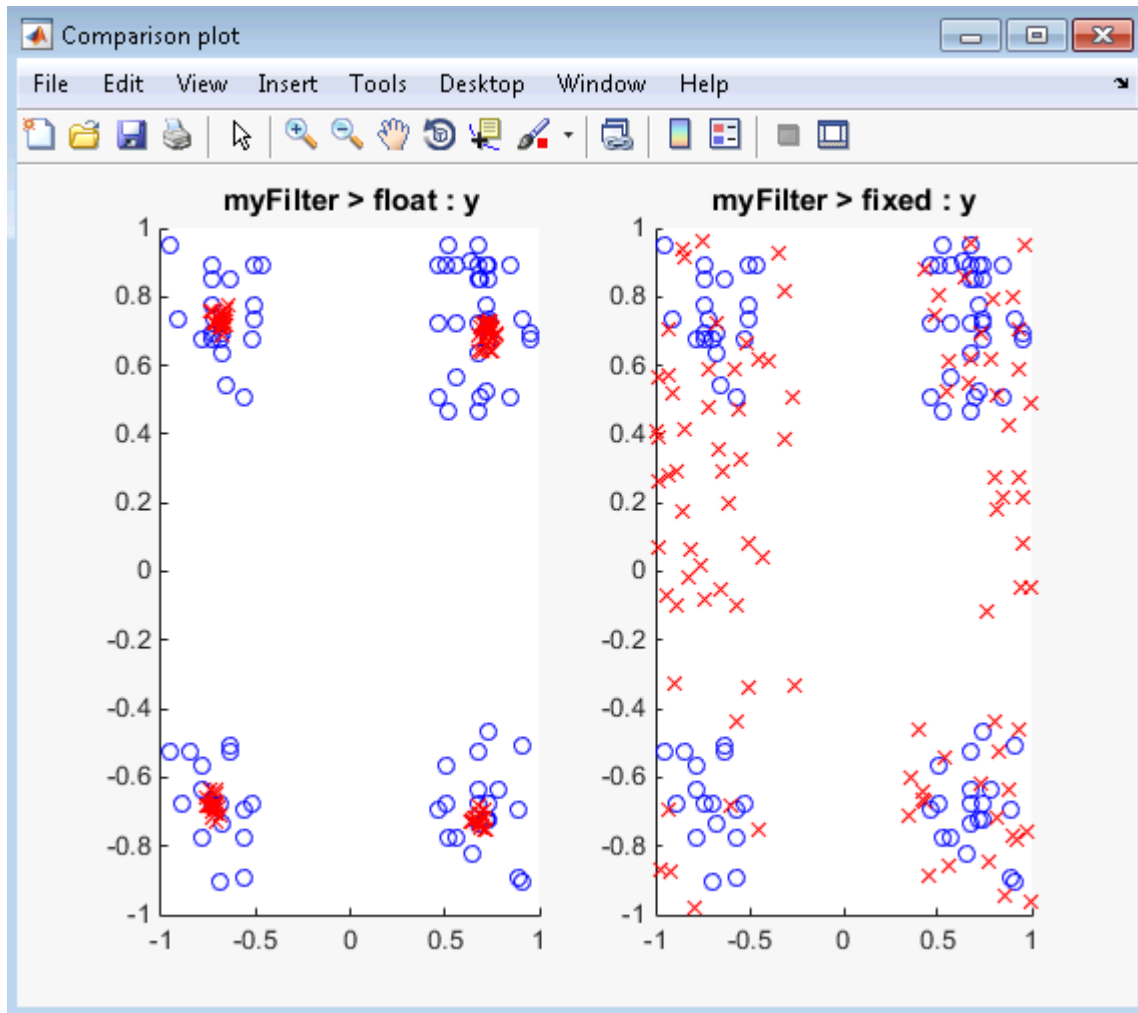
```

At the bottom right of the editor window, there is a blue bar with the text 'Next' and a right-pointing arrow.

Test Numerics and View Comparison Plots

- 1 Click **Test** arrow , select **Log inputs and outputs for comparison plots**, and then click **Test**.

The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the app uses this function to generate the comparison plot. The plot shows that the fixed-point results do not closely match the floating-point results.

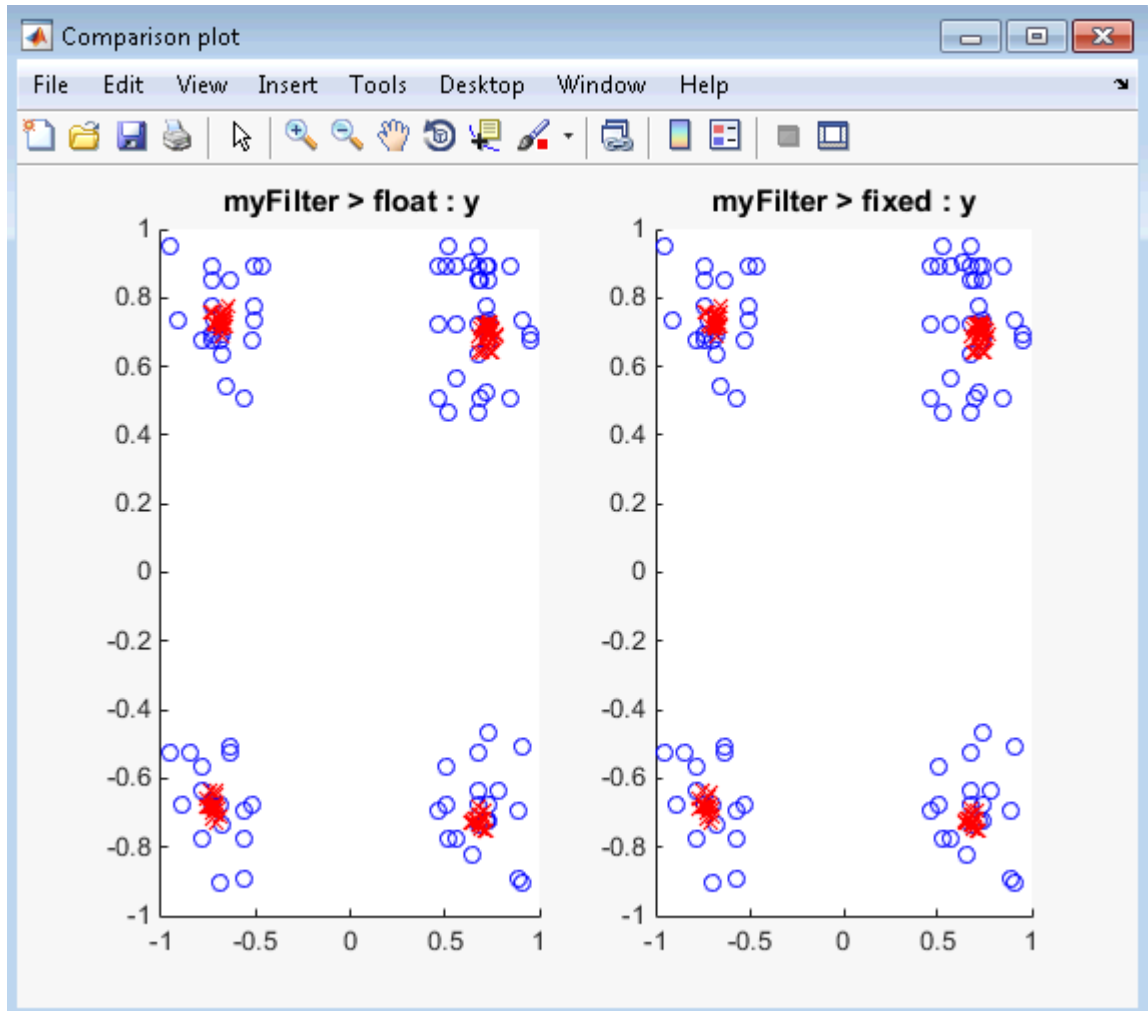


- 2 In the settings, increase the **DefaultWordLength** to 24 and then convert to fixed point again.

The app converts `myFilter.m` to fixed point and proposes fixed-point data types using the new default word length.


- 3 Run the test numerics step again.

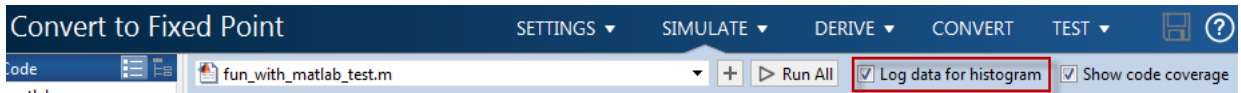
The increased word length improves the results. This time, the plot shows that the fixed-point results match the floating-point results.



Log Data for Histogram

To log data for histograms:

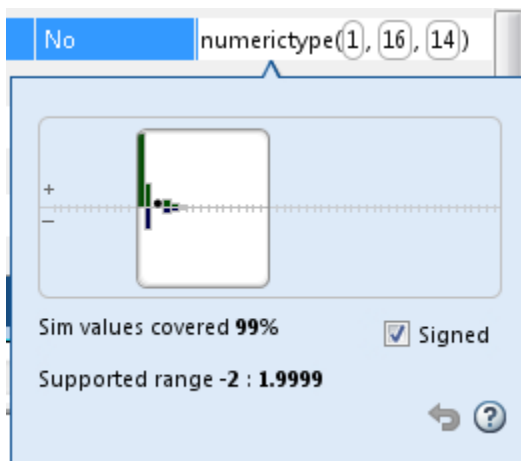
- On the **Convert to Fixed Point** page, click the **Simulate** arrow .
- Select **Log data for histogram**.



- Click **Simulate**.

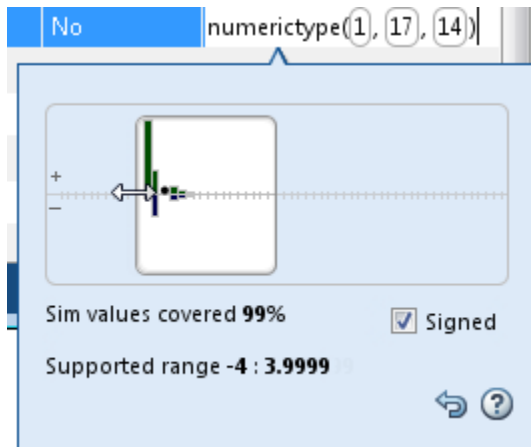
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

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 .

View and Modify Variable Information

View Variable Information

On the **Convert to Fixed Point** page of the MATLAB Coder app, you can view information about the variables in the MATLAB functions. To view information about the variables that you select in the **Source Code** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 15-103.

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 a variable in the MATLAB code window and on the **Variables** tab, use Ctrl+F. The app highlights occurrences of the variable in the code.

Modify Variable Information

If you modify variable information, the app highlights the modified values using bold text. 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 15-88.

Editing this field does not trigger static range analysis, but the app 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 15-88.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Whole Number**

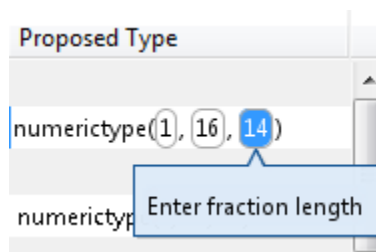
The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.

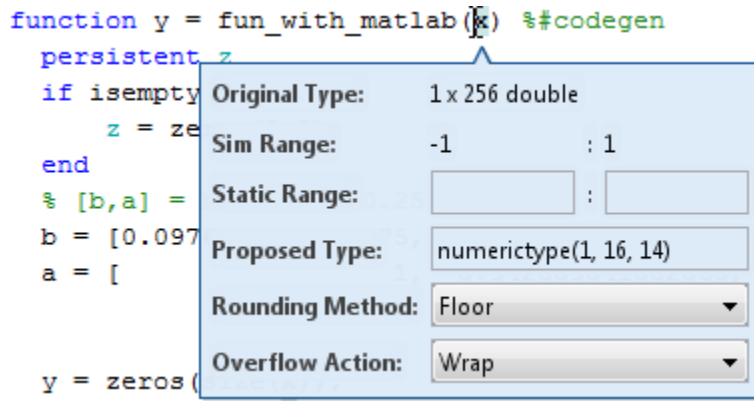
- **Proposed Type**

You can modify the signedness, word length, and fraction length settings individually:

- On the **Variables** tab, modify the value in the **ProposedType** field.



- In the code window, select a variable, and then modify 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 “Log Data for Histogram” on page 15-105.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select **Reset entire table**.

Simulation Output	Variables	Function Replacements						
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type	
<ul style="list-style-type: none"> Input x: 1 x 256 double, Sim Min: -1, Sim Max: 1, Proposed Type: numeric(1, 16, 14) Output y: 1 x 256 double, Sim Min: -0.97, Sim Max: 1.06, Proposed Type: numeric(1, 16, 14) Persistent z: 2 x 1 double, Sim Min: -0.89, Sim Max: 0.96, Proposed Type: numeric(1, 16, 15) Local b: 1 x 3 double, Sim Min: 0.1, Sim Max: 0.2, Proposed Type: numeric(0, 16, 18) a: 1 x 3 double, Sim Min: -0.94, Sim Max: 1, Proposed Type: numeric(1, 16, 14) i: double, Sim Min: 1, Sim Max: 256, Proposed Type: numeric(0, 9, 0) 								

- To revert the type of a selected variable to the type computed by the app, right-click the field and select **Undo changes**.
- To revert changes to variables, right-click the field and select **Undo changes for all variables**.

- To clear a static range value, right-click an edited field and select **Clear this static range**.
- To clear manually entered static range values, right-click anywhere on the **Variables** tab and select **Clear all manually entered static ranges**.

Promote Sim Min and Sim Max Values

With the MATLAB Coder app, you can promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.

Simulation Output	Variables	Function Replacements	Static Min	Static Max	Whole Number	Decimal	Type
<div style="border: 1px solid gray; padding: 2px;"> Copy sim ranges for all top-level inputs Copy sim ranges for all persistent variables Copy sim ranges for all global variables Clear all manually entered static ranges Reset entire table </div>							
▲ Input	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Decimal
x	1×256 double	-1	1				type(1, 16, 14)
▲ Output	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Decimal
y	1×256 double	-0.97	1.06				type(1, 16, 14)
▲ Persistent	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Decimal

To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select **Copy sim range**.
- Simulation ranges for top-level inputs, right-click the Static Min or Static Max column, and then select **Copy sim ranges for all top-level inputs**.
- Simulation ranges for persistent variables, right-click the Static Min or Static Max column, and then select **Copy sim ranges for all persistent variables**.

Automated Fixed-Point Conversion

In this section...

“Automated Fixed-Point Conversion Capabilities” on page 15-89

“Code Coverage” on page 15-90

“Proposing Data Types” on page 15-93

“Locking Proposed Data Types” on page 15-95

“Viewing Functions” on page 15-96

“Viewing Variables” on page 15-103

“Log Data for Histogram” on page 15-105

“Function Replacements” on page 15-107

“Validating Types” on page 15-108

“Testing Numerics” on page 15-108

“Detecting Overflows” on page 15-108

Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the MATLAB Coder app or at the command line using the `codegen` function `-float2fixed` option. 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 app uses them when proposing data types. In addition, you can modify and lock the proposed type so that the app cannot change it. For more information, see “Locking Proposed Data Types” on page 15-95.

For a list of supported MATLAB features and functions, see “MATLAB Language Features Supported for Automated Fixed-Point Conversion”.

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 app shows code coverage results. Your test files must 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.

The app covers basic MATLAB control constructs and shows statement coverage for basic blocks of code. The app 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


```
15 persistent current_state;
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
end
```

When you position your cursor over the coverage bar, the color highlighting extends over the code and the app 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.

```

15 persistent current_state;
16 if isempty(current_state)           Executed once
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,                         Reached 58% of the time
24
25         % value of output 'Z' depends both on state and inputs
26         if (A)                       Reached 33% of the time
27             Z = true;
28             current_state = S1;
29         else                           Reached 25% of the time
30             Z = false;
31             current_state = S2;
32         end
33     case S2,                         Reached 58% of the time
34                                     Reached 42% of the time
35
36         if (A)                       Reached 17% of the time
37             Z = false;
38             current_state = S1;
39         else                           Reached 25% of the time
40             Z = true;
41             current_state = S2;
42         end
43     case S3,                         Reached 42% of the time
44                                     Not reached
45
46         if (A)
47             Z = false;
48             current_state = S2;
49         else
50             Z = true;
51             current_state = S3;
52         end

```

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.
Orange	This behavior is expected 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 behavior is acceptable. 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” on page 15-94).

Proposing Data Types

The app 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 (also known as static ranges), 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 15-95.

Running a Simulation

During fixed-point conversion, the app generates an instrumented MEX function for your entry-point MATLAB file. If the build completes without errors, the app 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 app 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 app displays them on the **Function Replacements** tab. See “Function Replacements” on page 15-107.

Before running a simulation, specify the test file or files that you want to run. When you run a simulation, the app runs the test file, calling the instrumented MEX function. If you modify the MATLAB design code, the app 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 app cannot modify them.

If the test file fails, the errors are displayed on the **Simulation Output** tab.

Test files must 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 accuracy that you want. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the app 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 “Log Data for Histogram” on page 15-105.

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 app cannot modify them. The app 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 15-95.

When you select **Compute Derived Ranges**, the app 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 app 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. When the timeout is reached, the app aborts the analysis.

Locking Proposed Data Types

You can lock proposed data types against changes by the app using one of the following methods:

- Manually setting a proposed data type in the app.
- Right-clicking a type proposed by the tool and selecting **Lock computed value**.

The app 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

During the **Convert to Fixed Point** step of the fixed-point conversion process, you can view a list of functions in your project in the left 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 code window and the variables that they use are displayed on the **Variables** tab.

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 app retains the original function name in the fixed-point file name and appends the fixed-point suffix. For example, here the fixed-point version of `fun_with_matlab.m` is `fun_with_matlab_fixpt.m`.

The screenshot shows the MATLAB Coder interface with the 'Convert to Fixed Point' app. The main window displays the source code for 'fun_with_matlab_test.m'. The code is as follows:

```

6 function y = fun_with_matlab_fixpt(x) %#codegen
7     fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision');
8
9     persistent z
10    if isempty(z)
11        z = fi(zeros(2,1), 1, 16, 15, fm);
12    end
13    % [b,a] = butter(2, 0.25)
14    b = fi([0.0976310729378175, 0.195262145875635, 0.0976310729378175], 0, 16, 18, fm);
15    a = fi([1, -0.942809041582063, 0.333333333333333], 1, 16, 14, fm);
16
17
18    y = fi(zeros(size(x)), 1, 16, 14, fm);
19    for i=1:length(x)
20        y(i) = b(1)*x(i) + z(1);
21        z(1) = fi_signed(b(2)*x(i) + z(2) - a(2) * y(i);
22        z(2) = fi_signed(b(3)*x(i) - a(3) * y(i);
23    end
24 end
25
26

```

Below the code editor, a table shows the 'Type Validation Output' for variables x, y, z, and persistent variables.

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

Classes

The app displays information for the class and each of its methods. For example, consider a class, `Counter`, that has a static method, `MAX_VALUE`, and a method, `next`.

If you select the class, the app displays the class and its properties on the **Variables** tab.

The screenshot displays the MATLAB IDE interface. On the left, the 'Source Code' pane shows a project structure with files like 'Counter', 'next', 'MAX_VALUE', and 'use_counter'. Below it, the 'Output Files' pane lists generated files such as 'Counter_fixpt.m', 'use_counter_fixpt.m', and 'use_counter_report.html'. The main editor window shows the source code for a class named 'Counter'.

```

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
17        function v = next(this)
18            v = this.Value;
19            if this.Value == this.MAX_VALUE
20                this.Value = 0;
21            else
22                this.Value = this.Value + 1;
23            end
24        end
25    end
26 end
27

```

At the bottom of the editor, the 'Type Validation Output' tab is active, displaying a table with the following data:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
▲ Output							
▲ this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)

Navigation buttons 'Back' and 'Next' are visible at the bottom of the window.

If you select a method, the app displays only the variables that the method uses.

The screenshot displays the MATLAB IDE interface. On the left, the 'Source Code' pane shows a project structure with files like 'Counter', 'next', 'MAX_VALUE', and 'use_counter'. The 'Output Files' pane lists generated files such as 'Counter_fixpt.m' and 'use_counter_report.html'. The main editor shows the following MATLAB code:

```

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
17        function v = next(this)
18            v = this.Value;
19            if this.Value == this.MAX_VALUE
20                this.Value = 0;
21            else
22                this.Value = this.Value + 1;
23            end
24        end
25    end
26 end
27

```

Below the code editor, the 'Function Replacements' tab is active, showing a table of type validation results for the 'Output' variable 't':

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
▲ Output							
t	double	128	128			Yes	numeric(0, 8, 0)

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

Specializations

If a function is specialized, the app 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 the top-level function, the app displays all the variables on the **Variables** tab.

The screenshot shows a MATLAB IDE interface. On the left, a tree view lists source code files: 'dut', 'dut > bar > 1', 'dut > bar > 2', 'dut > bar > 3', 'dut > foo > 1', 'dut > foo > 2', and 'dut > foo > 3'. The main editor displays the source code for the 'dut' function, which calls 'foo' and 'bar' functions. Below the code, the 'Variables' tab is active, displaying a table of variables.

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
u	double	10	10			Yes	numericType(0, 4, 0)
v	double	20	20			Yes	numericType(0, 5, 0)
Output							
y	double	300	300			Yes	numericType(0, 9, 0)
Local							
tt1	double	20	20			Yes	numericType(0, 5, 0)

If you select the tree view, the app also displays the line numbers for the call to each specialization.

The screenshot displays a software interface for automated fixed-point conversion. The top section shows the source code for three functions: `dut`, `foo`, and `bar`. The bottom section shows a table of simulation output variables.

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

```

Simulation Output Variables Table:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
u	double	10	10			Yes	numerictype(0, 4, 0)
v	double	20	20			Yes	numerictype(0, 5, 0)
Output							
y	double	300	300			Yes	numerictype(0, 9, 0)
Local							
tt1	double	20	20			Yes	numerictype(0, 5, 0)

If you select a specialization, the app displays only the variables that the specialization uses.

The screenshot displays a software interface with a source code editor and a function replacement table. The source code editor shows the following 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

```

The function replacement table below the code is as follows:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
u	complex double					No	
Output							
y	complex double					No	

The interface also includes a "Source Code" list on the left with items like "dut", "foo > 1", "foo > 2", "foo > 3", "bar > 1", "bar > 2", and "bar > 3". At the bottom, there are "Back" and "Next" navigation buttons.

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 `foo > 1` is named `foo_s1`.

The screenshot displays the MATLAB Fixed-Point Designer interface. On the left, the 'Source Code' pane shows a list of files including 'dut', 'dut_wrapper_fixpt.m', 'dut_report.html', 'dut_fixpt_args.mat', 'dut_float_mex.mexw64', and 'dut_wrapper_fixpt_mex.mexw64'. The main editor shows the source code for the function 'dut_fixpt'. Below the code, the 'Variables' tab is active, displaying a table with the following data:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
u	double	10	10			Yes	numericType(0, 4, 0)
v	double	20	20			Yes	numericType(0, 5, 0)

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 app.

- **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 app 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 app.

- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

The app 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 app uses the edited values in subsequent analyses. You can revert to the types proposed by the app.

- 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 app 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 app highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Viewing Information for MATLAB Classes

The app displays:

- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Source Code** list on the **Convert to Fixed Point** page to select which class or class method to view. If you select a class method, the app highlights the method in the code window.

The screenshot shows the MATLAB IDE with a class definition for 'Counter' and its 'Variables' tab in the Type Validation Output window.

```

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
17        function v = next(this)
18            v = this.Value;
19            if this.Value == this.MAX_VALUE
20                this.Value = 0;
21            else
22                this.Value = this.Value + 1;
23            end
24        end
25    end
26 end
27

```


Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)
Output							
v	double	0	128			Yes	numerictype(0, 8, 0)

- Information about MATLAB classes on the **Variables** tab.

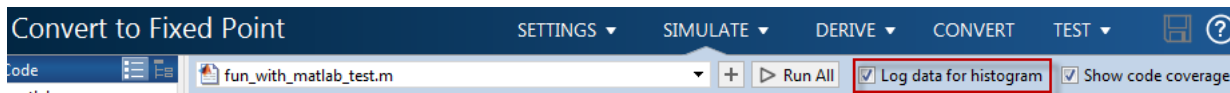
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)
Output							
v	double	0	128			Yes	numerictype(0, 8, 0)

Log Data for Histogram

To log data for histograms:

- On the **Convert to Fixed Point** page, click the **Simulate** arrow .

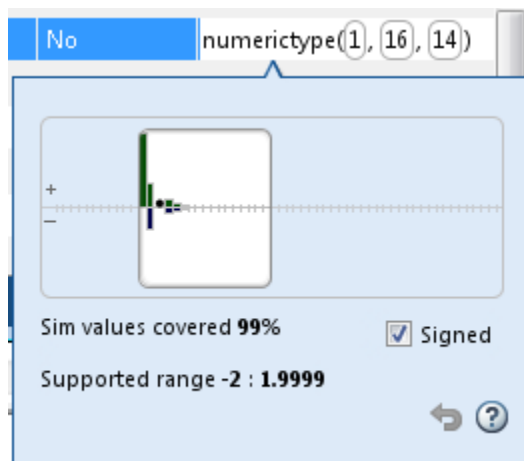
- Select **Log data for histogram**.



- Click **Simulate**.

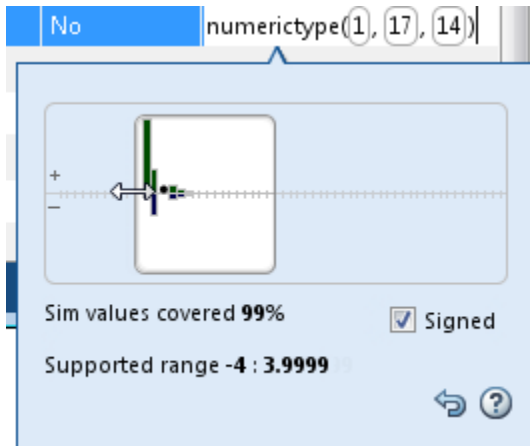
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

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 app 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.

Simulation Output				
Variables		Function Replacements		
Enter a function to replace				Custom Function ▾ +
Function or Operator	Replacement			
▲ Custom Function	<i>Function Name</i>			
foo	foo_fixedpoint			
▲ Lookup Table	<i>Interpolation Method</i>	<i>Design Min</i>	<i>Design Max</i>	<i>Number of Points</i>
exp	None	Auto	Auto	1000

You can add and remove function replacements from this list. If you enter a function replacement for a function, the replacement function is used when you build the project. If you do not enter a replacement, the app 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

Converting the code to fixed point 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 converting code to fixed point and validating the proposed fixed-point data types, click **Test** 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 app uses this test file to test numerics. Optionally, you can add test files and select to run more than one test file. The app 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 app 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.

After fixed-point simulation, if the numerical results do not meet the accuracy that you want, 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 results that you want.










Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion app 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 app 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 app also provides overflow information for these expressions.

See Also

“Detect Overflows” on page 15-42

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** 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 15-9, including these steps:

- 1 Complete the **Test** 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`.
- 2 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 -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.

The `fixedPointConverter` command generates a script in the current folder. `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” on page 15-7
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 15-9
- “Convert MATLAB Coder Project to MATLAB Script” on page 21-44

Generated Fixed-Point Code

In this section...

“Location of Generated Fixed-Point Files” on page 15-113

“Minimizing `fi`-casts to Improve Code Readability” on page 15-114

“Avoiding Overflows in the Generated Fixed-Point Code” on page 15-114

“Controlling Bit Growth” on page 15-115

“Avoiding Loss of Range or Precision” on page 15-115

“Handling Non-Constant `mpower` Exponents” on page 15-117

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.

File name	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"> • A structure for the input arguments. • The name of the fixed-point file.
<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.

File name	Description
<code>fcn_name_wrapper_fixpt.m</code>	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, <code>fcn_name_fixpt</code> .

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</code> is the local <code>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 numeric type 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 15-119

“Unsupported Constructs” on page 15-119

“Coding Style Best Practices” on page 15-120

Automated Conversion Support for MATLAB Classes

The automated fixed-point 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 MATLAB Coder app:

- 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” on page 15-104.

- 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
```

```
    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 15-122

“Prepare Your Algorithm for Code Acceleration or Code Generation” on page 15-123

“Check for Fixed-Point Support for Functions Used in Your Algorithm” on page 15-124

“Manage Data Types and Control Bit Growth” on page 15-124

“Convert to Fixed Point” on page 15-125

“Use the Histogram to Fine-Tune Data Type Settings” on page 15-126

“Optimize Your Algorithm” on page 15-127

“Avoid Explicit Double and Single Casts” on page 15-129

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>Issue</p> <p>Generation of test input and verification of results are intermingled with the algorithm code.</p> <p>Fix</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.	Algorithm in its own function <pre>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 MATLAB Coder app 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. See “Code Coverage” on page 15-90.

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” on page 4-2.

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” on page 19-6. 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” on page 19-8.

Check for Fixed-Point Support for Functions Used in Your Algorithm

The app flags unsupported function calls found in your algorithm on the **Function Replacements** tab. For example, if you use the `fft` function, 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.

Function Replacements	
Enter a function to replace	
Function or Operator	Replacement
<ul style="list-style-type: none"> Custom Function 	<i>Function Name</i>
<code>fft</code>	Replacement required to use fixed-point

You can specify additional replacement functions. For example, functions like `sin`, `cos`, and `sqrt` might support fixed point, but for better efficiency, you might want to consider an alternative implementation like a lookup table or CORDIC-based algorithm. The app provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. See “Replacing Functions Using Lookup Table Approximations” on page 15-130.

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

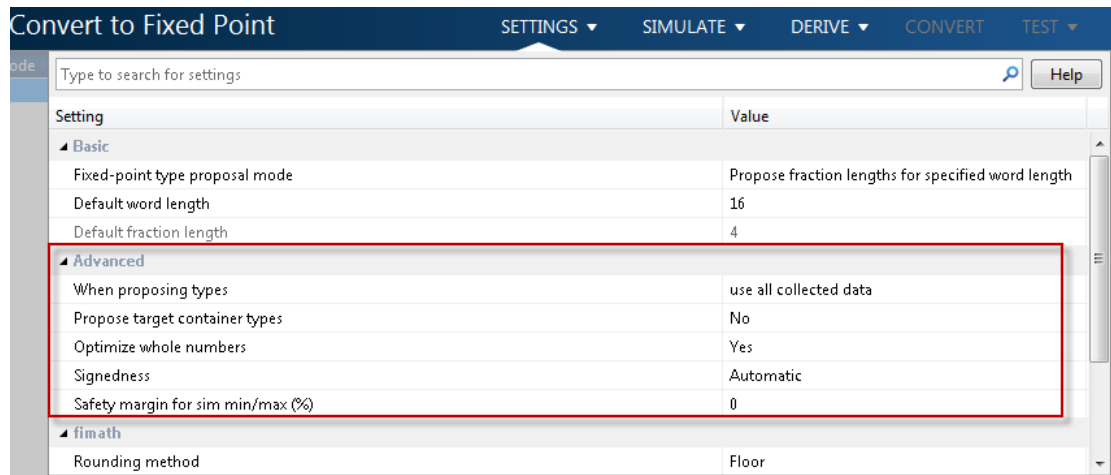
is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. In addition to preventing bit growth, subscripted assignment reduces the number of 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.



The screenshot shows the 'Convert to Fixed Point' settings window. The 'Advanced' section is highlighted with a red box. The settings are as follows:

Setting	Value
Basic	
Fixed-point type proposal mode	Propose fraction lengths for specified word length
Default word length	16
Default fraction length	4
Advanced	
When proposing types	use all collected data
Propose target container types	No
Optimize whole numbers	Yes
Signedness	Automatic
Safety margin for sim min/max (%)	0
fimath	
Rounding method	Floor


For more information, see “Specify Type Proposal Options” on page 15-38.

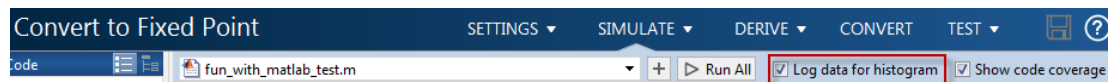
Run With Fixed-Point Types and Compare Results

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 15-94 and “Log Data for Histogram” on page 15-105 .

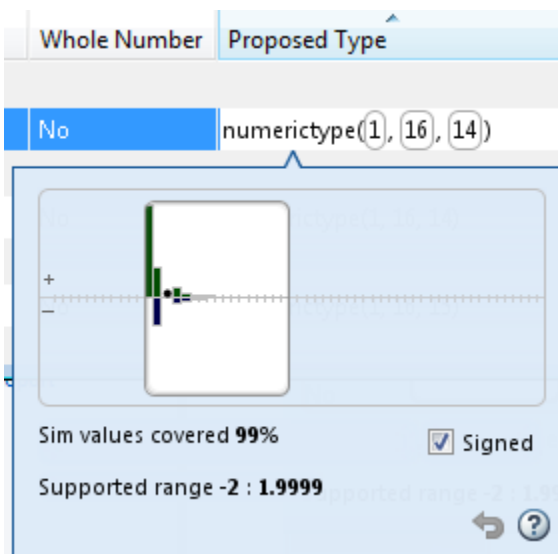
Use the Histogram to Fine-Tune Data Type Settings

To fine-tune fixed-point type settings, use the histogram. To log data for histograms, in the app, click the **Simulate** arrow  and select **Log data for histogram**.



After simulation and static analysis:

- To view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.



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 or 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 “Specify Type Proposal Options” on page 15-38.

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 function y = adder(a,b) y = a + b; end	Issue 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 & 65536) != 0) { i2 = i0 -65536; } else { i2 = i0 & 65535; } if ((i1 & 65536) != 0) { i3 = i1 -65536;</pre>
Note: In the app, set Default word length to 16.		

MATLAB Code	Best Practice	Generated C Code												
		<pre> } else { i3 = i1 & 65535; } i0 = i2 + i3; if ((i0 & 65536) != 0) { y = i0 -65536; } else { y = i0 & 65535; } return y; } </pre>												
	<p>Fix</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 app Settings. Select fimath and then set:</p> <table border="1" data-bbox="457 998 967 1288"> <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> </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 15-107.

Reimplement 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 15-107.

Replacing Functions Using Lookup Table Approximations

The MATLAB Coder 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” on page 15-53 and “Replace a Custom Function with a Lookup Table” on page 15-62.

To use lookup table approximations in the programmatic workflow, see `coder.approximation`, “Replace the exp Function with a Lookup Table” on page 16-22, and “Replace a Custom Function with a Lookup Table” on page 16-24.

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” on page 21-106). 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
- Global variables
- 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 14-2)
- 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.ArrayVectorAdder`
- `dsp.BiquadFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
- `dsp.FIRRateConverter`
- `dsp.LowerTriangularSolver`
- `dsp.LUFactor`
- `dsp.UpperTriangularSolver`
- `dsp.VariableFractionalDelay`
- `dsp.Window`

Inspecting Data Using the Simulation Data Inspector

In this section...

“What Is the Simulation Data Inspector?” on page 15-133

“Import Logged Data” on page 15-133

“Export Logged Data” on page 15-133

“Group Signals” on page 15-134

“Run Options” on page 15-134

“Create Report” on page 15-134

“Comparison Options” on page 15-134

“Enabling Plotting Using the Simulation Data Inspector” on page 15-134

“Save and Load Simulation Data Inspector Sessions” on page 15-135

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 at top**.

- Specify a Run Naming Rule

To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Options**.

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” on page 15-71.

To enable the Simulation Data Inspector in the programmatic workflow, see “Enable Plotting Using the Simulation Data Inspector” on page 16-27.

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” on page 16-28.

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 the MATLAB Coder App

- 1 On the **Convert to Fixed Point** page, to open the **Settings** dialog box, click the **Settings** arrow .
- 2 Under **Plotting and Reporting**, set **Highlight potential data type issues** to **Yes**.

When conversion is complete, open the fixed-point conversion HTML report to view the highlighting. Click **View report** in the **Type Validation Output** tab.

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.

For a strict-single precision design, specify a standard math library that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Standard math library** to **C99 (ISO)**.

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 for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. 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 has 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.

Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated

code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

Automated Fixed-Point Conversion Using Programmatic Workflow

- “Convert MATLAB Code to Fixed-Point C Code” on page 16-2
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 16-5
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 16-10
- “Detect Overflows” on page 16-18
- “Replace the exp Function with a Lookup Table” on page 16-22
- “Replace a Custom Function with a Lookup Table” on page 16-24
- “Enable Plotting Using the Simulation Data Inspector” on page 16-27
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 16-28

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” on page 16-5
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 16-10
- “Enable Plotting Using the Simulation Data Inspector” on page 16-27

More About

- “Automated Fixed-Point Conversion” on page 15-89

Propose Fixed-Point Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data using the `codegen` function.

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/.

You can use `mex -setup` to change the default compiler. See “Change 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})
end
```



```

    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		1		No	numericity(1, 16, 14)
b	double 1 x 3	0.0976310729378175	0.195262145875635			No	numericity(0, 16, 18)
i	double		1	256		Yes	numericity(0, 9, 0)
x	double 1 x 256	-0.9999756307053946		1		No	numericity(1, 16, 14)
y	double 1 x 256	-0.9696817930434206	1.0553496057969345			No	numericity(1, 16, 14)
z	double 2 x 1	-0.8907046852192462	0.957718532859117			No	numericity(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, 0.333333333333333 ], 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 = numerictype( a );
    new_nt = numerictype( 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” on page 15-7
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 16-10

Propose Fixed-Point Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges using the `codegen` function. 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/

You can use `mex -setup` to change the default compiler. See “Change 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
```

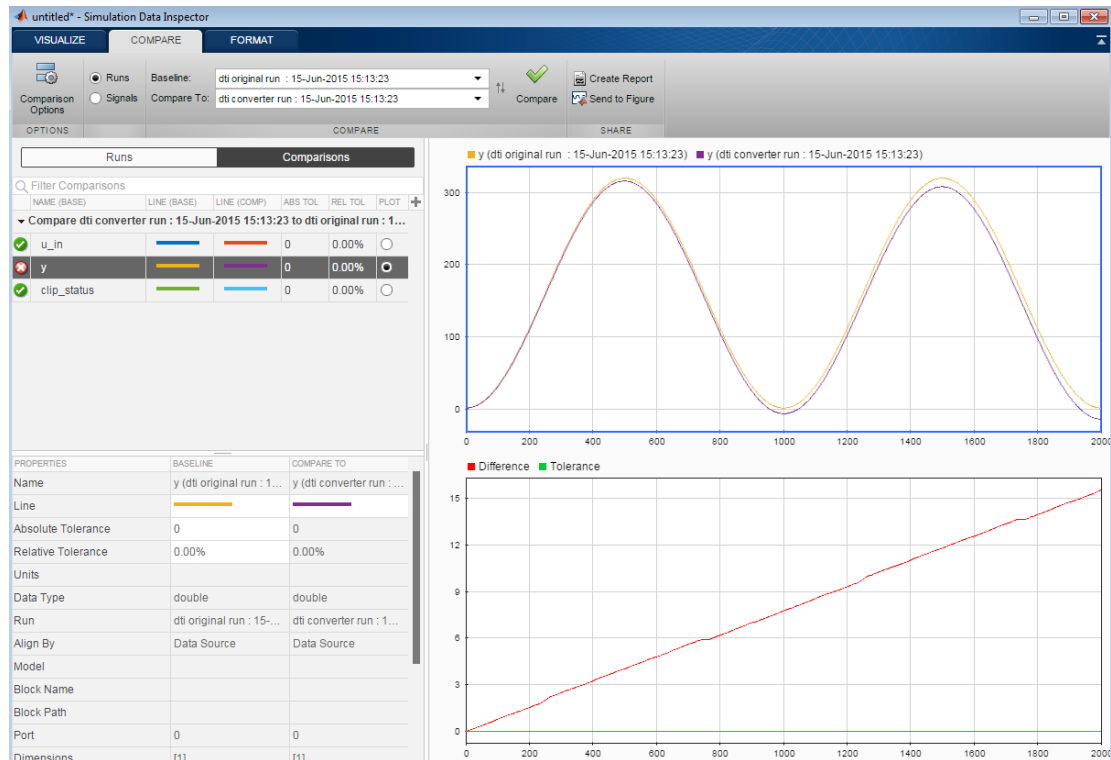
Compare Floating-Point and Fixed-Point Runs

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” on page 15-7
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 16-5

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 `codegen` function.

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/.

You can use `mex -setup` to change the default compiler. See “Change 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 `codegen` function.

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/

You can use `mex -setup` to change the default compiler. See “Change 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 input and output data logged using the `codegen` function. 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” on page 16-10.

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/

You can use `mex -setup` to change the default compiler. See “Change 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
```

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-Point conversion.
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;

```

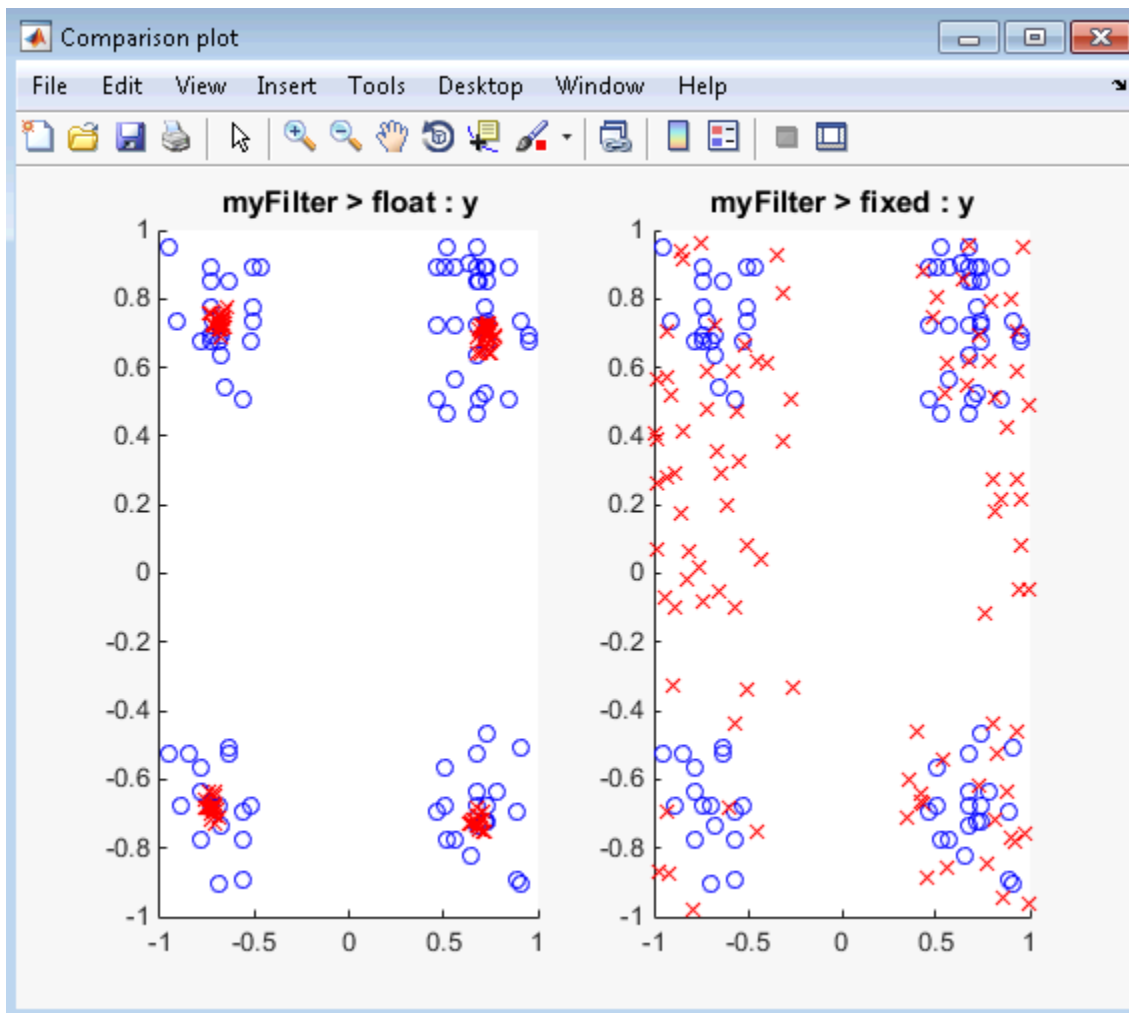
Convert to Fixed Point

Convert the floating-point MATLAB function, `myFilter`, to fixed-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.

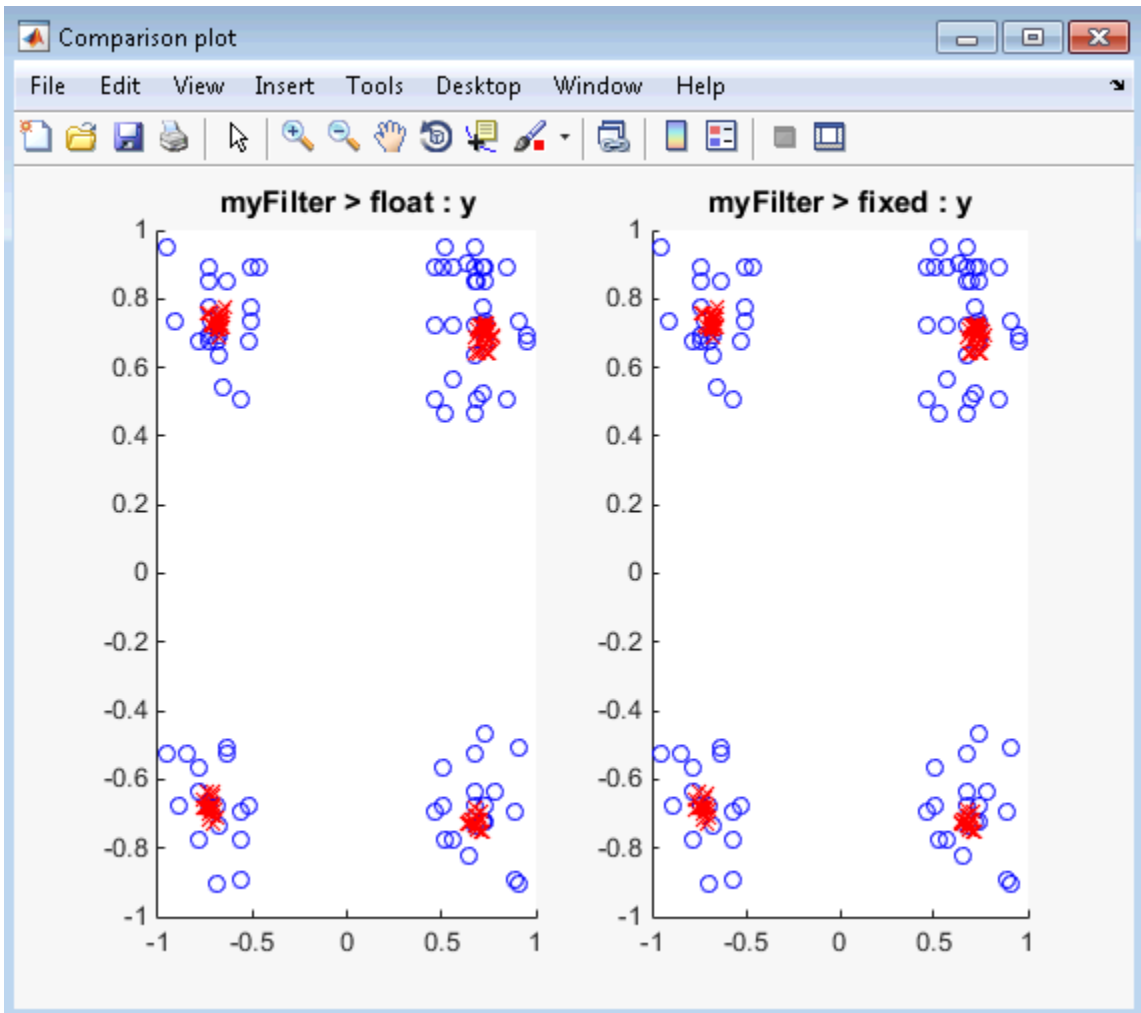


The plot shows that the fixed-point results do not closely match the floating-point results.

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.



Single-Precision Conversion

- “Generate Single-Precision C Code at the Command Line” on page 17-2
- “Generate Single-Precision C Code Using the MATLAB Coder App” on page 17-8
- “Generate Single-Precision MATLAB Code” on page 17-14
- “Choose a Single-Precision Conversion Workflow” on page 17-23
- “Single-Precision Conversion Best Practices” on page 17-24
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 17-28
- “Combining Integers and Double-Precision Numbers” on page 17-32
- “MATLAB Language Features Supported for Single-Precision Conversion” on page 17-33

Generate Single-Precision C Code at the Command Line

In this section...

“Prerequisites” on page 17-2

“Create a Folder and Copy Relevant Files” on page 17-2

“Determine the Type of the Input Argument” on page 17-4

“Generate and Run Single-Precision MEX to Verify Numerical Behavior” on page 17-5

“Generate Single-Precision C Code” on page 17-5

“View the Generated Single-Precision C Code” on page 17-6

“View Potential Data Type Issues” on page 17-6

This example shows how to generate single-precision C code from double-precision MATLAB code at the command line.

Prerequisites

To complete this example, 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/.

To change the default compiler, you can use `mex -setup`. See “Change Default Compiler”.

Create a 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 function 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, 0.333333333333333];

    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

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `fun_with_matlab` function with three input signals: chirp, step, and impulse. 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.')
```

Determine the Type of the Input Argument

To determine the type of the input argument `x`, use `coder.getArgTypes` to run the test file `fun_with_matlab_test.m`

```
types = coder.getArgTypes('fun_with_matlab_test', 'fun_with_matlab');
```


The test file runs and displays the outputs of the filter for each of the input signals. `coder.getArgTypes` determines that the input type of `x` is 1x256 double.

Generate and Run Single-Precision MEX to Verify Numerical Behavior

- 1 Before you generate single-precision C code, generate a single-precision MEX function that you can use to verify the behavior of the generated single-precision code. To indicate that you want the single-precision MEX code, use the `-singleC` option.

```
codegen -singleC fun_with_matlab -args types -report
```

During MEX generation, the code generation software detects single-precision conversion issues. Before you generate C/C++ code, fix these issues. This example does not have single-precision conversion issues.

The generated MEX accepts single-precision and double-precision input. You can use the same test file to run the double-precision MATLAB function and the single-precision MEX function. You do not have to modify the test file to call the single-precision MEX function.

- 2 Run the test file `fun_with_matlab_test.m`. This file calls the double-precision MATLAB function `fun_with_matlab.m`.

```
fun_with_matlab_test
```

- 3 The test file runs and displays the outputs of the filter for each of the input signals.
- 4 Run the test file `fun_with_matlab_test`, replacing calls to the double-precision `fun_with_matlab` function with calls to the single-precision `fun_with_matlab_mex` function.

```
coder.runTest('fun_with_matlab_test', 'fun_with_matlab')
```

- 5 The test file runs and displays the outputs of the filter for each of the input signals. The single-precision MEX function produces the same results as the double-precision MATLAB function.

Generate Single-Precision C Code

- 1 Create a code configuration object for generation of a C static library, dynamic library, or executable. To avoid use of double-precision implementations of math functions, specify `'C99'` for the standard math library.

```
cfg = coder.config('lib');  
cfg.TargetLangStandard = 'C99 (ISO)'
```

- 2 To generate single-precision C code, call `codegen` with the `-singleC` option. Enable generation of the code generation report.

```
codegen -config cfg -singleC fun_with_matlab -args {types{1}} -report
```

View the Generated Single-Precision C Code

To view the code generation report for the C code generation, click the **View Report** link.

The code generation report displays the generated code for `fun_with_matlab.c`.

- Double-precision variables have type `float` in the C code.
- The index `i` is an integer.

View Potential Data Type Issues

When you generate single-precision code, `codegen` enables highlighting of potential data type issues in the code generation report. If `codegen` cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation.

Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

See Also

`codegen` | `coder.config` | `coder.getArgTypes` | `coder.runTest`

Related Examples

- “Generate Single-Precision C Code Using the MATLAB Coder App” on page 17-8
- “Generate Single-Precision MATLAB Code” on page 17-14

More About

- “Single-Precision Conversion Best Practices” on page 17-24

- “Warnings from Conversion to Single-Precision C/C++ Code” on page 17-28

Generate Single-Precision C Code Using the MATLAB Coder App

In this section...

- “Prerequisites” on page 17-8
- “Create a Folder and Copy Relevant Files” on page 17-8
- “Open the MATLAB Coder App” on page 17-10
- “Enable Single-Precision Conversion” on page 17-11
- “Select the Source Files” on page 17-11
- “Define Input Types” on page 17-12
- “Check for Run-Time Issues” on page 17-12
- “Generate Single-Precision C Code” on page 17-12
- “View the Generated C Code” on page 17-13
- “View Potential Data Type Issues” on page 17-13

This example shows how to generate single-precision C code from double-precision MATLAB code by using the MATLAB Coder app.

Prerequisites

To complete this example, 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/.

To change the default compiler, you can use `mex -setup`. See “Change Default Compiler”.

Create a 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 function 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

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `fun_with_matlab` function with three input signals: chirp, step, and impulse. 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)

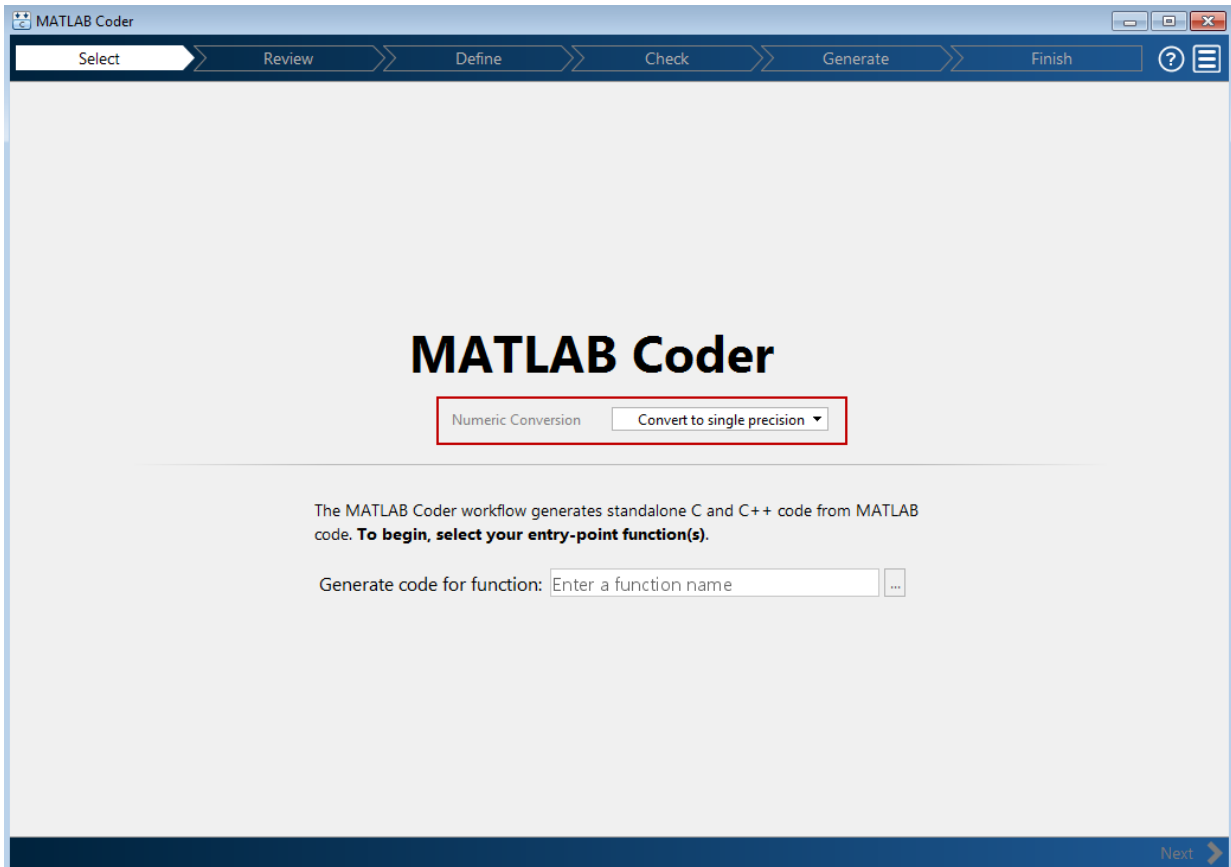
disp('Test complete.')
```

Open the MATLAB Coder App

- 1 Navigate to the work folder that contains the file for this example.
- 2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Enable Single-Precision Conversion

Set **Numeric Conversion** to **Convert** to single precision.



Select the Source Files

- 1 To add the entry-point function `fun_with_matlab` to the project, browse to the file `fun_with_matlab.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `fun_with_matlab.prj`.
- 2 Click **Next** to go to the **Define Input Types** step.

The app screens `fun_with_matlab.m` for code violations and code generation readiness issues. The app does not find issues in `fun_with_matlab.m`.

Define Input Types

- 1 On the **Define Input Types** page, to add `fun_with_matlab_test` as a test file, browse to `fun_with_matlab_test`. Click **Open**.
- 2 Click **Autodefine Input Types**.

The test file runs and displays the outputs of the filter for each of the input signals. The app determines that the input type of `x` is `double(1x256)`.

- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

To detect and fix single-precision conversion issues, perform the **Check for Run-Time Issues** step.

- 1 On the **Check for Run-Time Issues** page, the app populates the test file field with `fun_with_matlab_test`, the test file that you used to define the input types.
- 2 Click **Check for Issues**.

The app generates a single-precision MEX function from `fun_with_matlab`. It runs the test file `fun_with_matlab_test` replacing calls to `fun_with_matlab` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. Click a message to highlight the problematic code in a window where you can edit the code. In this example, the app does not detect issues.

- 3 Click **Next** to go to the **Generate Code** page.

Generate Single-Precision C Code

- 1 In the **Generate** dialog box, set **Build type** to **Static Library**.
- 2 Set **Language** to **C**.
- 3 To avoid use of double-precision standard math functions, set the standard math library to **C99 (ISO)**. Click **More Settings**. Then, click **Custom Code**. Set **Standard math library** to **C99 (ISO)**.
- 4 For other settings, use the default values.

- 5 To generate the code, click **Generate**.

MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/fun_with_matlab`.

View the Generated C Code

The app displays the generated code for `fun_with_matlab.c`.

- Double-precision variables have type `float` in the C code.
- The index `i` is an integer.

View Potential Data Type Issues

When you generate single-precision code, the app enables highlighting of potential data type issues in the code generation report. If the app cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation.

To open the code generation report, click the **View Report** link. Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

Related Examples

- “Generate Single-Precision C Code at the Command Line” on page 17-2

More About

- “Single-Precision Conversion Best Practices” on page 17-24
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 17-28

Generate Single-Precision MATLAB Code

This example shows how to generate single-precision MATLAB code from double-precision MATLAB code. This example shows the single-precision conversion workflow that you use when you want to see single-precision MATLAB code or use verification options. Optionally, you can also generate single-precision C/C++ code.

Prerequisites

To complete this example, 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/.

To change the default compiler, you can use `mex -setup`. See “Change Default Compiler”.

Create a 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 function 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

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `fun_with_matlab` function with three input signals: chirp, step, and impulse. 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 Single-Precision Configuration Object

Create a single-precision configuration object. Specify the test file name. Verify the single-precision code using the test file. Plot the error between the double-precision code and single-precision code. Use the default values for the other properties.

```
scfg = coder.config('single');
scfg.TestBenchName = 'fun_with_matlab_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

Generate Single-Precision MATLAB Code

To convert the double-precision MATLAB function, `fun_with_matlab`, to single-precision MATLAB code, use the `codegen` function with the `-double2single` option.

```
codegen -double2single scfg fun_with_matlab
```

`codegen` analyzes the double-precision code. The conversion process infers types by running the test file because you did not specify the input types for the

`fun_with_matlab` function. The conversion process selects single-precision types for the double-precision variables. It selects `int32` for index variables. When the conversion is complete, `codegen` generates a type proposal report.

View the Type Proposal Report

To see the types that the conversion process selected for the variables, open the type proposal report for the `fun_with_matlab` function. Click the link `fun_with_matlab_report.html`.

The report opens in a web browser. The conversion process converted:

- Double-precision variables to `single`.
- The index `i` to `int32`. The conversion process casts index and dimension variables to `int32`.

Single-Precision Report *fun_with_matlab*

Simulation Coverage	Code
100%	function y = fun_with_matlab(x) %#codegen
Once	<pre> persistent z if isempty(z) z = zeros(2,1); end </pre>
100%	<pre> % [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 </pre>

Variable Name	Type	Sim Min	Sim Max	Whole Number	ProposedType
a	double 1 x 3	-0.942809041582063		1 No	single
b	double 1 x 3	0.0976310729378175	0.195262145875635	No	single
i	double	1	256	Yes	int32
x	double 1 x 256	-0.9999756307053946		1 No	single
y	double 1 x 256	-0.9696817930434206	1.0553496057969345	No	single
z	double 2 x 1	-0.8907046852192462	0.957718532859117	No	single

View Generated Single-Precision MATLAB Code

To view the report for the generation of the single-precision MATLAB code, in the Command Window:

- 1 Scroll to the **Generate Single-Precision Code** step. Click the **View report** link.
- 2 On the **MATLAB code** tab, under **Functions**, click `fun_with_matlab_single`.

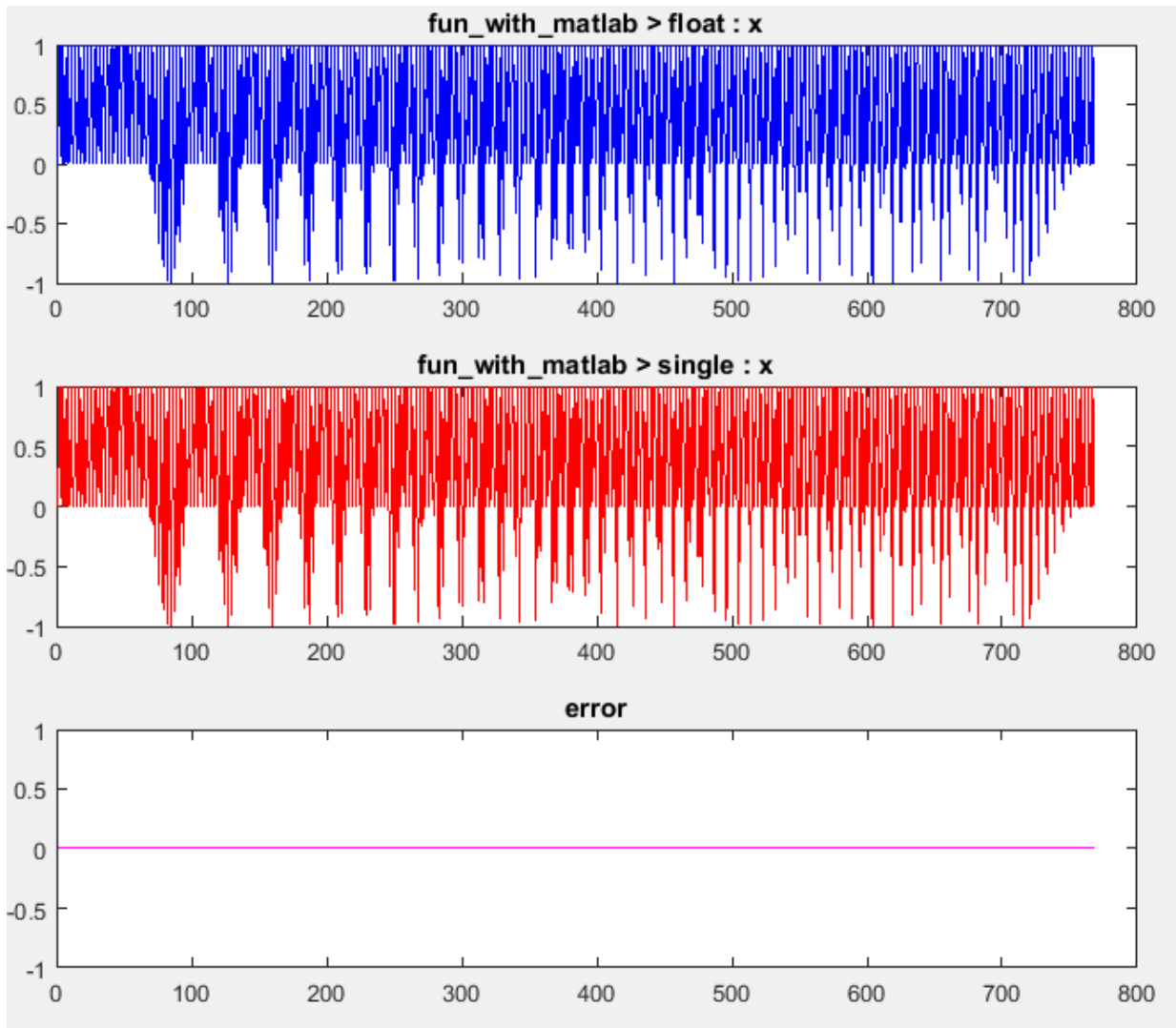
The code generation report displays the single-precision MATLAB code for `fun_with_matlab`.

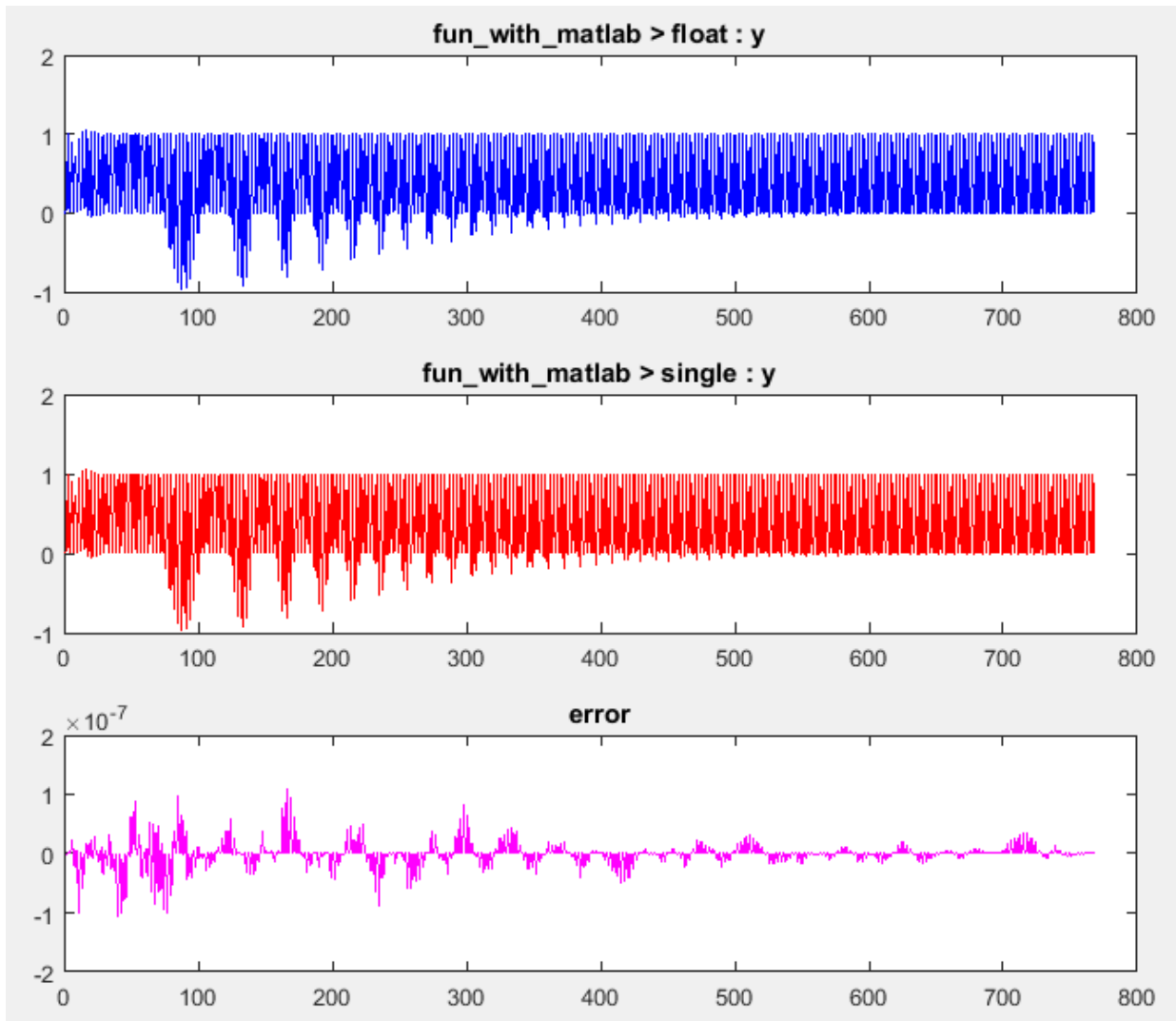
View Potential Data Type Issues

When you generate single-precision code, `codegen` enables highlighting of potential data type issues in code generation reports. If `codegen` cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation. Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

Compare the Double-Precision and Single-Precision Variables

You can see the comparison plots for the input `x` and output `y` because you selected to log inputs and outputs for comparison plots .





Optionally Generate Single-Precision C Code

If you also want to generate single-precision C code, create a code configuration object for C code generation. Use this configuration object with the `-config` option of the `codegen` function. For example:

- 1 Create a code configuration object for generation of a C static library. Specify 'C99' for the standard math library.

```
cfg = coder.config('lib');  
cfg.TargetLangStandard = 'C99 (ISO)';
```

- 2 Generate the C code. Enable generation of the code generation report.

```
codegen -double2single scfg -config cfg fun_with_matlab -report
```

- 3 To view the code generation report for the C code generation, click the **View Report** link.

The code generation report displays the generated code for `fun_with_matlab.c`.

- Double-precision variables have type `float` in the C code.
- The index `i` is an integer.

When you generate single-precision code, `codegen` enables highlighting of potential data type issues in the code generation report. If `codegen` cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation.

Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

See Also

`coder.SingleConfig` | `codegen` | `coder.config`

Related Examples

- “Generate Single-Precision C Code Using the MATLAB Coder App” on page 17-8
- “Generate Single-Precision C Code at the Command Line” on page 17-2

More About

- “Single-Precision Conversion Best Practices” on page 17-24
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 17-28

Choose a Single-Precision Conversion Workflow

The information in the following table helps you to decide which single-precision workflow to use.

Goal	Use
You want to generate single-precision C/C++ code in the most direct way using the <code>codegen</code> function.	<code>codegen</code> with <code>-singleC</code> option. See “Generate Single-Precision C Code at the Command Line” on page 17-2.
You want to generate single-precision C/C++ code in the most direct way using the MATLAB Coder app.	The MATLAB Coder app with Numeric Conversion set to Convert to single precision . See “Generate Single-Precision C Code Using the MATLAB Coder App” on page 17-8.
You want to generate only single-precision MATLAB code. You want to see the single-precision MATLAB code or use verification options.	<code>codegen</code> with the <code>-double2single</code> option and a <code>coder.SingleConfig</code> object. See “Generate Single-Precision MATLAB Code” on page 17-14.
You want to generate single-precision MATLAB code, and then generate single-precision C/C++ code from the single-precision MATLAB code.	<code>codegen</code> with the <code>-double2single</code> option and a <code>coder.SingleConfig</code> object. Also, use the <code>-config</code> object with a code configuration object for the output type that you want. See “Generate Single-Precision MATLAB Code” on page 17-14.

Single-Precision Conversion Best Practices

In this section...

“Use Integers for Index Variables” on page 17-24

“Limit Use of `assert` Statements” on page 17-24

“Initialize MATLAB Class Properties in Constructor” on page 17-24

“Provide a Test File That Calls Your MATLAB Function” on page 17-24

“Prepare Your Code for Code Generation” on page 17-25

“Verify Double-Precision Code Before Single-Precision Conversion” on page 17-25

“Best Practices for Generation of Single-Precision C/C++ Code” on page 17-25

“Best Practices for Generation of Single-Precision MATLAB Code” on page 17-26

Use Integers for Index Variables

In MATLAB code that you want to convert to single precision, it is a best practice to use integers for index variables. However, if the code does not use integers for index variables, when possible single-precision conversion using `codegen` with `-double2single` tries to detect the index variables and select `int32` types for them.

Limit Use of `assert` Statements

- Do not use `assert` statements to define the properties of input arguments.
- Do not use `assert` statements to test the type of a variable. For example, do not use

```
assert(isa(a, 'double'))
```

Initialize MATLAB Class Properties in Constructor

Do not initialize MATLAB class properties in the `properties` block. Instead, use the constructor to initialize the class properties.

Provide a Test File That Calls Your MATLAB Function

Separate your core algorithm from other code that you use to test and verify the results. Create a test file that calls your double-precision MATLAB algorithm. You can use the test file to:

- Automatically define properties of the top-level function inputs.
- Verify that the double-precision algorithm behaves as you expect. The double-precision behavior is the baseline against which you compare the behavior of the single-precision versions of your algorithm.
- Compare the behavior of the single-precision version of your algorithm to the double-precision baseline.

For best results, the test file must exercise the algorithm over its full operating range.

Prepare Your Code for Code Generation

MATLAB code that you want to convert to single precision must comply with code generation requirements. See “MATLAB Programming for Code Generation”.

To help you identify unsupported functions or constructs in your MATLAB code, add the `%codegen` pragma to the top of your MATLAB file. When you edit your code in the MATLAB editor, the MATLAB Code Analyzer flags functions and constructs that are not supported for code generation. See “Check Code With the Code Analyzer” on page 19-6. When you use the MATLAB Coder app, the app screens your code for code generation readiness. At the function line, you can use the Code Generation Readiness Tool. See “Check Code Using the Code Generation Readiness Tool” on page 19-8.

Verify Double-Precision Code Before Single-Precision Conversion

Before you begin the single-precision conversion process, verify that you can successfully generate code from your double-precision MATLAB code. Generate and run a MEX version of your double-precision MATLAB code so that you can:

- Detect and fix compilation issues.
- Verify that the generated single-precision code behaves the same as the double-precision MATLAB code.

See “Why Test MEX Functions in MATLAB?” on page 20-4

Best Practices for Generation of Single-Precision C/C++ Code

When you generate single-precision C/C++ code by using the MATLAB Coder app or `codegen` with the `-singleC` option, follow these best practices:

Set the Standard Math Library to C99

When you generate C/C++ libraries or executables, by default, the code generation software uses the C89 /C90 (ANSI) standard math library. When you generate single-precision C/C++ code using this library, the code generation software warns you if a function in this library uses double precision. To avoid this warning, set the standard math library to C99 (ISO). See “Warnings from Conversion to Single-Precision C/C++ Code” on page 17-28.

Cast Large Double Constant to Integer

For a constant greater than 2^{24} , in your original double-precision MATLAB function, cast the constant to an integer type that is large enough for the constant value. For example:

```
a = int32(2^24 + 1);
```

Generate and Run Single-Precision MEX Before Generating Single-Precision C/C++ Code

Before you generate single-precision C code, generate and run a single-precision MEX version of your MATLAB code. When you follow this practice, you can detect and fix compiler issues. You can verify that the single-precision MEX function has the same functionality as the MATLAB code.

If you use `codegen` with `-singleC`:

- 1 Generate the single-precision MEX.
- 2 Call `coder.runTest` to run your test file, replacing calls to the double-precision MATLAB code with calls to the single-precision MEX code.

If you use the MATLAB Coder app, perform the **Check for Run-Time Issues** step with single-precision conversion enabled.

Best Practices for Generation of Single-Precision MATLAB Code

When you use `codegen` with the `-double2single` option to generate single-precision MATLAB code, follow these best practices:

Use the `-args` Option to Specify Input Properties

When you generate single-precision MATLAB code, if you specify a test file, you do not have to specify argument properties with the `-args` option. In this case, the code

generation software runs the test file to determine the properties of the input types. However, running the test file can slow the code generation. It is a best practice to determine the input properties one time with `coder.getArgTypes`. Then, pass the properties to the `-args` option. For example:

```
types = coder.getArgTypes('myfun_test', 'myfun');
scfg = coder.config('single');
codegen -double2single scfg -args types myfun -report
```

When you repeat the code generation in the same MATLAB session, this practice saves you time.

Test Numerics and Log I/O Data

When you use the `codegen` function with the `-double2single` option to generate single-precision MATLAB code, enable numerics testing and I/O data logging for comparison plots. To use numerics testing, you must provide a test file that calls your MATLAB function. To enable numerics testing and I/O data logging, create a `coder.SingleConfig` object. Set the `TestBenchName`, `TestNumerics`, and `LogIOForComparisonPlotting` properties. For example:

```
scfg = coder.config('single');
scfg.TestBenchName = 'mytest';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

More About

- “Warnings from Conversion to Single-Precision C/C++ Code” on page 17-28

Warnings from Conversion to Single-Precision C/C++ Code

When you generate single-precision C/C++ code by using the MATLAB Coder app or `codegen` with the `-singleC` option, you can receive the following warnings.

Function Uses Double-Precision in the C89/C90 Standard

If the standard math library is C89/C90, the conversion process warns you when a function uses double-precision code in the C89/C90 standard.

Consider the function `mysine`.

```
function c = mysine(a)
c = sin(a);
end
```

Generate single-precision code for `mysine`.

```
x = -pi:0.01:pi;
codegen -singleC mysine -args {x} -config:lib -report
```

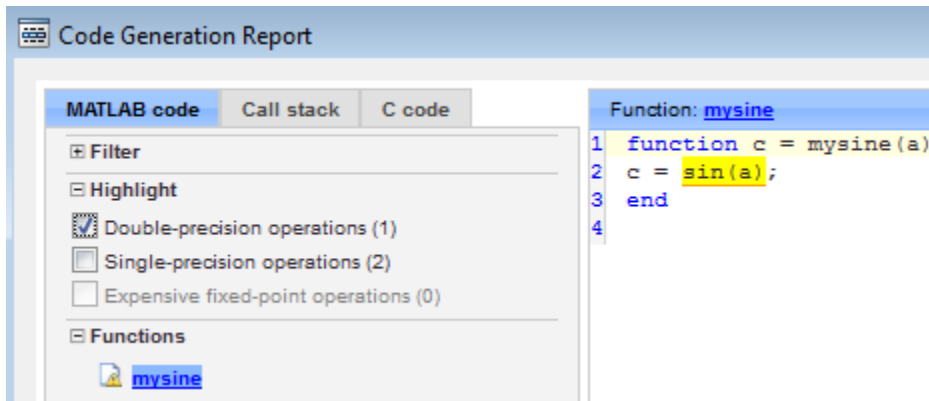
`codegen` warns that `sin` uses double-precision in the C89/C90 (ANSI) standard.

Warning: The function `sin` uses double-precision in the C89/C90 (ANSI) standard. For single-precision code, consider using the C99 (ISO) standard or use your own function.

To open the code generation report, click the **View Report** link.

To see that double-precision operations remain in the converted code, click the **MATLAB code** tab. Under **Highlight**, select the **Double-precision operations** check box. Under **Functions**, click `mysine`.

In the code pane, the report highlights the `sin` call.



To address this warning, specify use of the C99 (ISO) standard math library.

- At the command line:

```
cfg = coder.config('lib');
cfg.TargetLangStandard = 'C99 (ISO)'
```

- In the app, in the project build settings, on the **Custom Code** tab, set **Standard math library** to C99 (ISO).

Built-In Function Is Implemented in Double-Precision

Some built-in MATLAB functions are implemented using double-precision operations. The conversion process warns that the code generated for these functions contains double-precision operations.

Consider the function `geterf` that calls the built-in function `erf`.

```
function y = geterf(x)
y = erf(x);
end
```

Generate single-precision code for `geterf`.

```
codegen -singleC -config:lib -args {1} geterf -report
```

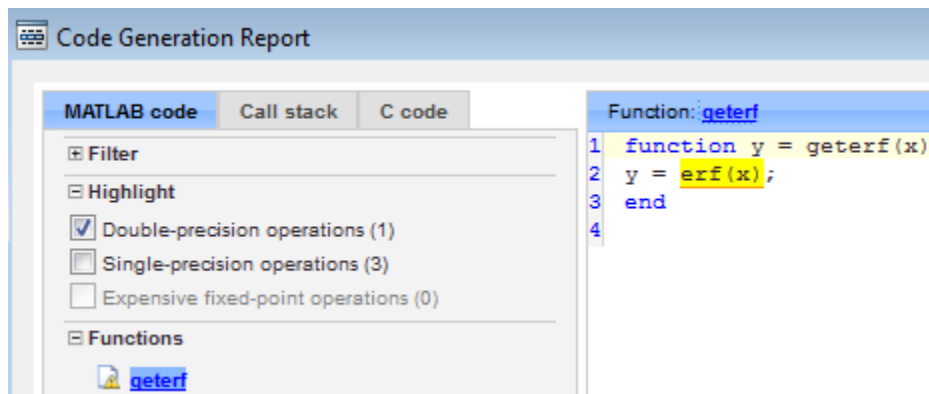
`codegen` warns that `erf` is implemented in double precision.

```
Warning: The builtin function erf is implemented in double-precision. Code generated for this function will contain doubles.
```

To open the code generation report, click the **View Report** link.

To see that double-precision operations remain in the converted code, click the **MATLAB code** tab. Under **Highlight**, select the **Double-precision operations** check box. Under **Functions**, click `geterf`.

In the code pane, the report highlights the `erf` call.



To address this warning, rewrite your code so that it does not use the function that is implemented in double precision.

Built-In Function Returns Double-Precision

If a built-in MATLAB function returns a double-precision output, the conversion process generates a warning.

Consider the function `mysum` that calls the built-in function `sum`.

```
function y = mysum(x)
y = sum(int32(x));
end
```

Generate single-precision code for `mysum`.

```
A = 1:10;
codegen -singleC -config:lib -args {A} mysum -report
```

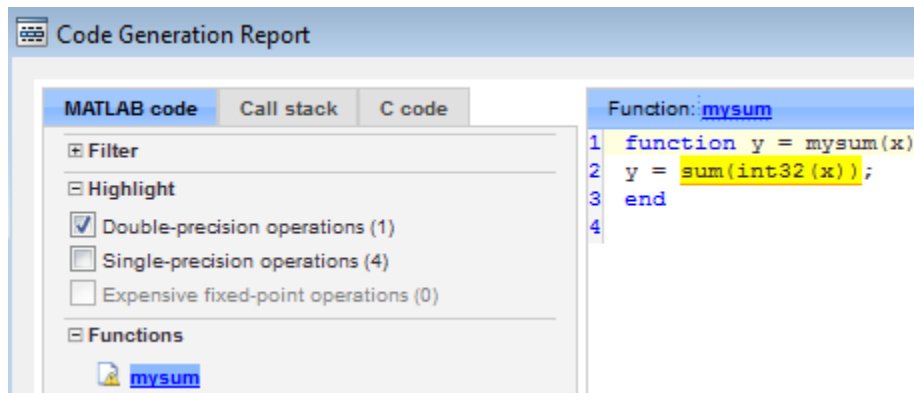
`codegen` warns that `mysum` is implemented in double precision.

Warning: The output of builtin function sum is double-precision and has been cast to single-precision. The code generated for the builtin function may still contain doubles.

To open the code generation report, click the **View Report** link.

To see that double-precision operations remain in the converted code, click the **MATLAB code** tab. Under **Highlight**, select the **Double-precision operations** check box. Under **Functions**, click **mysum**.

In the code pane, the report highlights the `sum` call.



To address this warning, specify that you want the function to return the 'native' class.

```
(sum(int32(1), 'native'))
```

Using this option causes the function to return the same type as the input.

More About

- “Single-Precision Conversion Best Practices” on page 17-24

Combining Integers and Double-Precision Numbers

MATLAB supports the combination of integers of the same class and scalar double-precision numbers. MATLAB does not support the combination of integers and single-precision numbers. If you use the MATLAB Coder app or `codegen` with the `-singleC` option to generate single-precision C/C++ code, your MATLAB code cannot combine integers and double-precision numbers. Converting an expression that combines integers and doubles results in an illegal MATLAB expression. To work around this limitation, cast the numbers so that the types of the numbers match. Either cast the integer numbers to double-precision or cast the double-precision numbers to the integer class.

For example, consider the function `dut` that returns the sum of `a` and `b`.

```
function c = dut(a,b)
c = a + b;
end
```

Generate single-precision code using `codegen` with the `-singleC` option. Specify that the first argument is double and the second argument is `int32`.

```
codegen -singleC -config:lib dut -args {0, int32(2)} -report
```

Code generation fails. The message suggests that you cast the operands so that they have the same types.

Rewrite the code so that it cast `a` to the type of `b`.

```
function c = dut(a,b)
c = int32(a) + b;
end
```

MATLAB Language Features Supported for Single-Precision Conversion

In this section...

“MATLAB Language Features Supported for Single-Precision Conversion” on page 17-33

“MATLAB Language Features Not Supported for Single-Precision Conversion” on page 17-34

MATLAB Language Features Supported for Single-Precision Conversion

Single-precision conversion supports the following MATLAB language features:

- N-dimensional arrays.
- Matrix operations, including deletion of rows and columns.
- Variable-size data (see “Generate Code for Variable-Size Data” on page 21-106). Comparison plotting does not support variable-size data.
- Subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 7-31).
- Complex numbers (see “Code Generation for Complex Data” on page 6-4).
- Numeric classes (see “Supported Variable Types” on page 5-17).
- Program control statements `if`, `switch`, `for`, `while`, and `break`.
- Arithmetic, relational, and logical operators.
- Local functions.
- Global variables.
- Persistent variables (see “Define and Initialize Persistent Variables” on page 5-10).
- Structures.
- Characters.

Single-precision conversion does not support the complete set of Unicode characters. Characters are restricted to 8 bits of precision in generated code. Many mathematical operations require more than 8 bits of precision. If you intend to convert your MATLAB algorithm to single precision, it is a best practice not to perform arithmetic with characters.

- MATLAB classes. Single-precision conversion supports:
 - Class properties
 - Constructors
 - Methods
 - Specializations

It does not support class inheritance or packages.

- Function calls (see “Resolution of Function Calls for Code Generation” on page 14-2)
- `varargin` and `varargout` are supported when you use `codegen` with `-singleC`. They are not supported when you use `codegen` with `-double2single`.

MATLAB Language Features Not Supported for Single-Precision Conversion

Single-precision conversion does not support the following features:

- Anonymous functions
- Cell arrays
- Function handles
- Java
- Nested functions
- Recursion
- Sparse matrices
- `try/catch` statements
- `varargin` and `varargout` are not supported when you use `codegen` with `-double2single`. They are supported when you use `codegen` with `-singleC`

Setting Up a MATLAB Coder Project

- “Set Up a MATLAB Coder Project” on page 18-2
- “Specify Properties of Entry-Point Function Inputs Using the App” on page 18-3
- “Automatically Define Input Types Using the App” on page 18-4
- “Define Input Parameters by Example Using the App” on page 18-5
- “Define or Edit Input Parameter Type Using the App” on page 18-12
- “Define Constant Input Parameters Using the App” on page 18-23
- “Define Inputs Programmatically in the MATLAB File” on page 18-24
- “Add Global Variables Using the App” on page 18-25
- “Specify Global Variable Type and Initial Value Using the App” on page 18-26
- “Changing Output Type” on page 18-30
- “Slow Operations in MATLAB Coder App” on page 18-33
- “Unable to Open a MATLAB Coder Project” on page 18-34

Set Up a MATLAB Coder Project

- 1 To open the app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.
- 2 Create a project or open an existing project. See “Create a Project” on page 18-2 and “Open an Existing Project” on page 18-2.
- 3 If the app detects code generation readiness issues in your entry-point functions, address these issues.
- 4 Define the properties of the entry-point function input types. See “Specify Properties of Entry-Point Function Inputs Using the App” on page 18-3.
- 5 Check for run-time issues. Provide code or a test file that the app can use to test your code. The app generates a MEX function. It runs your test code or test file, replacing calls to your MATLAB function with calls to the MEX function. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.
- 6 Configure the build settings. Select the build type, language, and production hardware. Optionally, modify other build settings. See “Configure Build Settings” on page 21-27.


You can now generate code.

Create a Project

On the **Select Source Files** page, specify the MATLAB files from which you want to generate code. An entry-point function is a function that you call from MATLAB. Do not add files that have spaces in their names.

The app creates a project that has the name of the first entry-point function.

Open an Existing Project

- 1 On the app toolbar, click  and select **Open existing project**.
- 2 Type or select the project.

The app closes other open projects.

If the project is a Fixed-Point Converter project, and you have a Fixed-Point Designer license, the project opens in the Fixed-Point Converter app.

Specify Properties of Entry-Point Function Inputs Using the App

Why Specify Input Properties?

Because C and C++ are statically typed languages, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. To infer variable properties in MATLAB files, MATLAB Coder must 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, you do not need to specify properties of inputs to local functions or external functions called by the primary function.

Unless you use the tilde (~) character to specify unused function inputs, you must specify the same number and order of inputs as the MATLAB function . If you use the tilde character, the inputs default to real, scalar doubles.

See Also

- “Properties to Specify” on page 21-46

Specify an Input Definition Using the App

Specify an input definition using one of the following methods:

- Autodefine Input Types
- Define Type
- Define by Example
- Define Constant
- “Define Inputs Programmatically in the MATLAB File” on page 18-24

Automatically Define Input Types Using the App

If you specify a test file that calls the project entry-point functions, the MATLAB Coder app can infer the input parameter types by running the test file. If a test file calls an entry-point function multiple times with different size inputs, the app takes the union of the inputs. The app infers that the inputs are variable size, with an upper bound equal to the size of the largest input.

Before using the app to automatically define function input parameter types, you must add at least one entry-point file to your project. You must also specify code that calls your entry-point functions with the expected input types. It is a best practice to provide a test file that calls your entry-point functions. The test file can be either a MATLAB function or a script. The test file must call the entry-point function at least once.

To automatically define input types:

- 1 On the **Define Input Types** page, specify a test file. Alternatively, you can enter code directly.
- 2 Click **Autodefine Input Types**.

The app runs the test file and infers the types for entry-point input parameters. The app displays the inferred types.

Note: If you automatically define the input types, the entry-point functions must be in a writable folder.

Define Input Parameters by Example Using the App

In this section...

“Define an Input Parameter by Example” on page 18-5

“Specify Input Parameters by Example” on page 18-6

“Specify a Structure Type Input Parameter by Example” on page 18-7

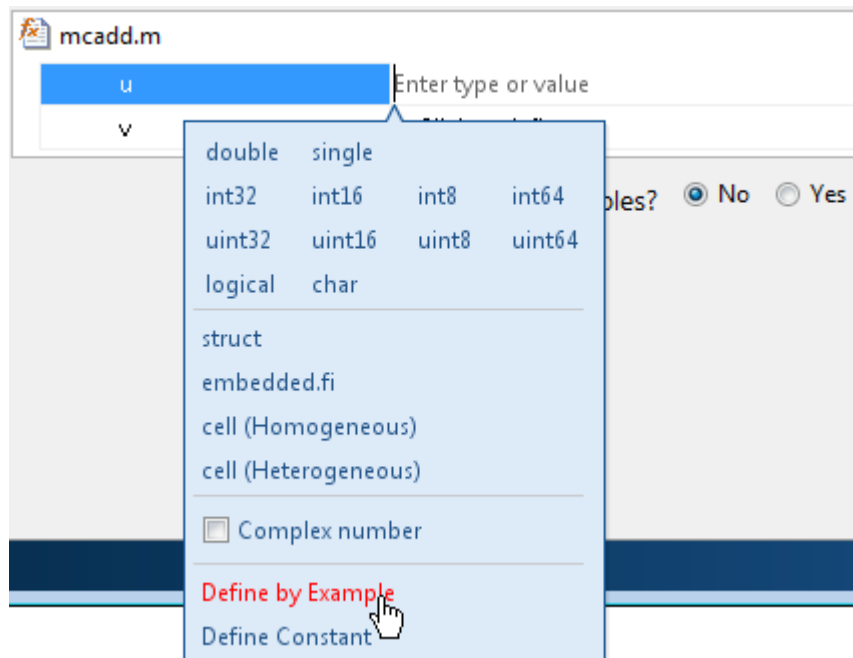
“Specify a Cell Array Type Input Parameter by Example” on page 18-7

“Specify an Enumerated Type Input Parameter by Example” on page 18-9

“Specify a Fixed-Point Input Parameter by Example” on page 18-10

Define an Input Parameter by Example

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.



- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter a MATLAB expression. The variable has the class, size, and complexity of the value of the expression.

Specify Input Parameters by Example

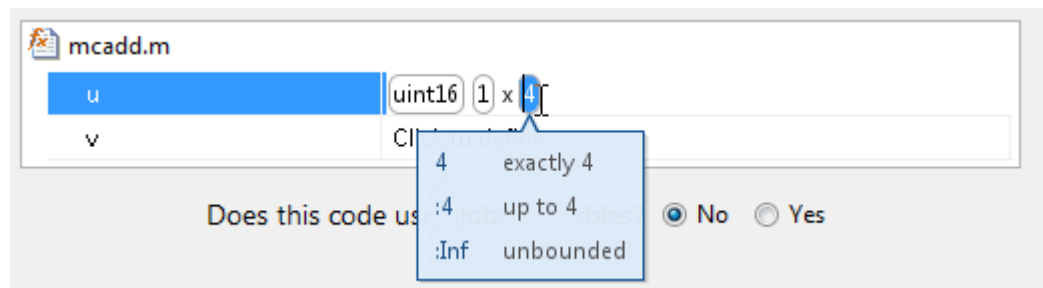
This example shows how to specify a 1-by-4 vector of unsigned 16-bit integers.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter:

```
zeros(1,4,'uint16')
```

The input type is `uint16(1x4)`.

- 5 Optionally, after you specify the input type, you can specify that the input is variable size. For example, select the second dimension.



- 6 To specify that the second dimension is variable size with an upper bound of 4, select `:4`. Alternatively, to specify that the second dimension is unbounded, select `:Inf`.

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, enter 0.

Specify a Structure Type Input Parameter by Example

This example shows how to specify a structure with two fields **a** and **b**. The input type of **a** is scalar double. The input type of **b** is scalar char.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter:

```
struct('a', 1, 'b', 'x')
```

The type of the input parameter is `struct(1x1)`. The type of field **a** is `double(1x1)`. The type of field **b** is `char(1x1)`.

- 5 For an array of structures, to specify the size of each dimension, click the dimension and specify the size. For example, enter 4 for the first dimension.
- 6 To specify that the second dimension is variable size with an upper bound of 4, select `:4`. Alternatively, to specify that the second dimension is unbounded select `:Inf`.

Alternatively, specify the size of the array of structures in the `struct` function call. For example, `struct('a', { 1 2}, 'b', {'x', 'y'})` specifies a 1x2 array of structures with fields **a** and **b**. The type of field **a** is `double(1x1)`. The type of field **b** is `char(1x1)`.

To modify the type definition, see “Specify a Structure Input Parameter” on page 18-14.

Specify a Cell Array Type Input Parameter by Example

This example shows how to specify a cell array input by example. When you define a cell array by example, the app determines whether the cell array is homogeneous or heterogeneous. See “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2. If you want to control whether the cell array is homogeneous or heterogeneous, specify the cell array by type. See “Specify a Cell Array Input Parameter” on page 18-18.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.

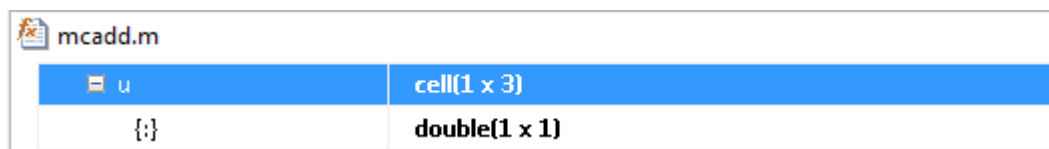
3 Select **Define by Example**.

4 In the field to the right of the parameter, enter an example cell array.

- If all cell array elements have the same properties, the cell array is homogeneous. For example, enter:

```
{1 2 3}
```

The input is a 1x3 cell array. The type of each element is `double(1x1)`.



The screenshot shows a window titled 'mcadd.m' with a table of parameters. The first row is highlighted in blue and shows a parameter 'u' with a value of 'cell(1 x 3)'. The second row shows curly braces containing a colon, '{:}', with a type of 'double(1 x 1)'.

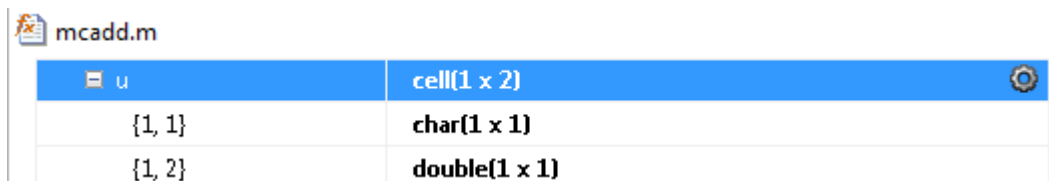
u	cell(1 x 3)
{:}	double(1 x 1)

The colon inside curly braces{:} indicates that all elements have the same properties.

- If elements of the cell array have different classes, the cell array is heterogeneous. For example, enter:

```
{'a', 1}
```

The input is a 1x2 cell array. For a heterogeneous cell array, the app lists each element. The type of the first element is `char(1x1)`. The type of the second element is `double(1x1)`.



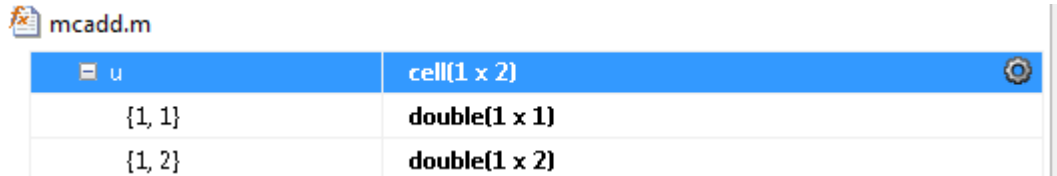
The screenshot shows a window titled 'mcadd.m' with a table of parameters. The first row is highlighted in blue and shows a parameter 'u' with a value of 'cell(1 x 2)'. The second row shows curly braces containing '1, 1', '{1, 1}', with a type of 'char(1 x 1)'. The third row shows curly braces containing '1, 2', '{1, 2}', with a type of 'double(1 x 1)'.

u	cell(1 x 2)
{1, 1}	char(1 x 1)
{1, 2}	double(1 x 1)

- For some example cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For these cell arrays, the app uses heuristics to determine whether the cell array is homogeneous or heterogeneous. For example, for the example cell array, enter:

```
{1 [2 3]}
```

The elements have the same class, but different sizes. The app determines that the input is a 1x2 heterogeneous cell array. The type of the first element is `double(1x1)`. The type of the second element is `double(1x2)`.



u	cell[1 x 2]
{1, 1}	double(1 x 1)
{1, 2}	double(1 x 2)

However, the example cell array, `{1 [2 3]}`, can also be a homogeneous cell array whose elements are `1x:2 double`. If you want this cell array to be homogeneous, do one of the following:

- Specify the cell array input by type. Specify that the input is a homogeneous cell array. Specify that the elements are `1x:2 double`. See “Specify a Cell Array Input Parameter” on page 18-18.
- Right-click the variable. Select **Homogeneous**. Specify that the elements are `1x:2 double`.

If you use `coder.typeof` to specify that the example cell array is variable size, the app makes the cell array homogeneous. For example, for the example input, enter:

```
coder.typeof({1 [2 3]}, [1 3], [0 1])
```

The app determines that the input is a `1x:3 homogeneous cell array` whose elements are `1x:2 double`.

To modify the type definition, see “Specify a Cell Array Input Parameter” on page 18-18.

Specify an Enumerated Type Input Parameter by Example

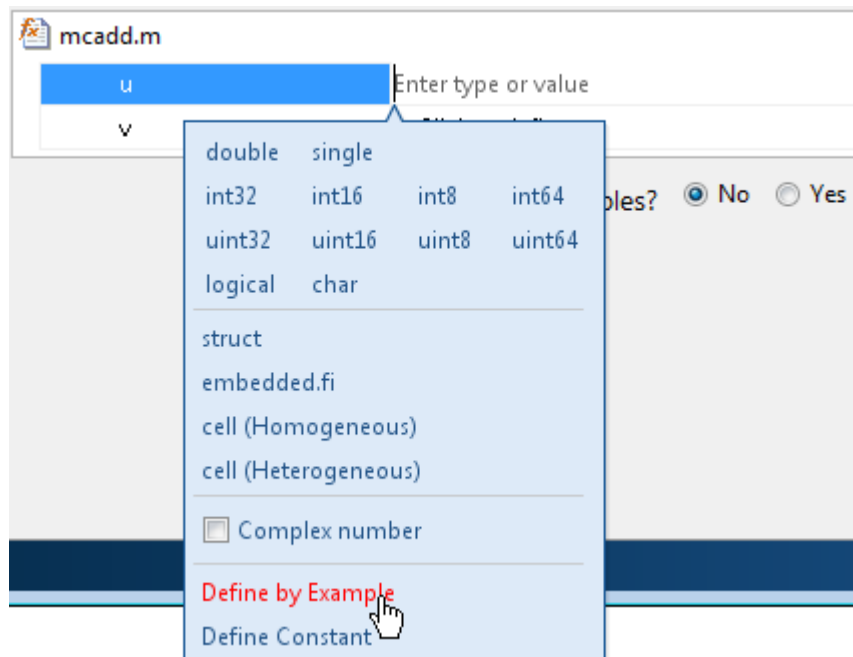
This example shows how to specify that an input uses the enumerated type `MyColors`.

Suppose that `MyColors.m` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

To specify that an input has the enumerated type `MyColors`:

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.



- 3 Select **Define by Example**.
- 4 In the field to the right of the parameter, enter the following MATLAB expression:
`MyColors.red`

Specify a Fixed-Point Input Parameter by Example

To specify fixed-point inputs, Fixed-Point Designer software must be installed.

This example shows how to specify a signed fixed-point type with a word length of eight bits, and a fraction length of three bits.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.

- 3** Select **Define by Example**.
- 4** In the field to the right of the parameter, enter:

```
fi(10, 1, 8, 3)
```

The app sets the type of input `u` to `fi(1x1)`. By default, if you do not specify a local `fimath`, the app uses the default `fimath`. See “`fimath` for Sharing Arithmetic Rules”.

Optionally, modify the fixed-point properties or the size of the input. See “Specify a Fixed-Point Input Parameter by Type” on page 18-14 and “Define or Edit Input Parameter Type Using the App” on page 18-12.

Define or Edit Input Parameter Type Using the App

In this section...

“Define or Edit an Input Parameter Type” on page 18-12

“Specify an Enumerated Type Input Parameter by Type” on page 18-13

“Specify a Fixed-Point Input Parameter by Type” on page 18-14

“Specify a Structure Input Parameter” on page 18-14

“Specify a Cell Array Input Parameter” on page 18-18

Define or Edit an Input Parameter Type

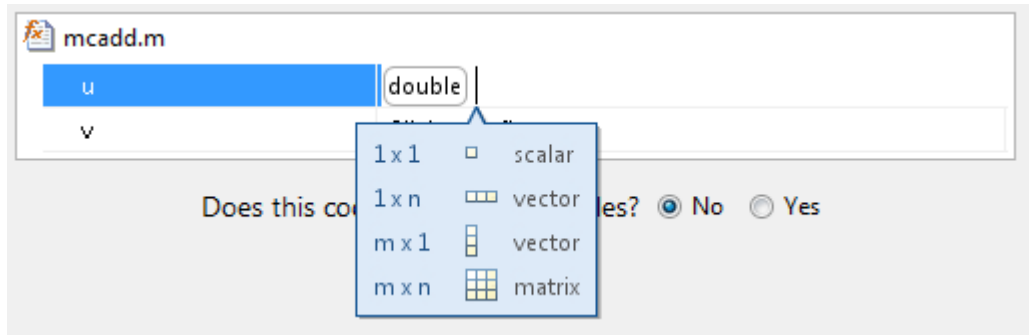
The following procedure shows you how to define or edit `double`, `single`, `int64`, `int32`, `int16`, `int8`, `uint64`, `uint32`, `uint16`, `uint8`, `logical`, and `char` types.

For more information about defining other types, see the following table.

Input Type	Link
A structure (struct)	“Specify a Structure Input Parameter” on page 18-14
A cell array (cell (Homogeneous) or cell (Heterogeneous))	“Specify a Cell Array Input Parameter” on page 18-18
A fixed-point data type (embedded.fi)	“Specify a Fixed-Point Input Parameter by Type” on page 18-14
An input by example (Define by Example)	“Define Input Parameters by Example Using the App” on page 18-5
A constant (Define Constant)	“Define Constant Input Parameters Using the App” on page 18-23

- 1 Click the field to the right of the input parameter name.
- 2 Optionally, for numeric types, to make the parameter a complex type, select the **Complex number** check box.
- 3 Select the input type.

The app displays the selected type. It displays the 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, the app 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.

Specify an Enumerated Type Input Parameter by Type

To specify that an input uses the enumerated type `MyColors`:

- 1 Suppose that the enumeration `MyColors` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

- 2 On the **Define Input Types** page, click **Let me enter input or global types directly**.

- 3 In the field to the right of the input parameter, enter `MyColors`.


Specify a Fixed-Point Input Parameter by Type

To specify fixed-point inputs, Fixed-Point Designer software must be installed.

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **embedded.fi**.
- 4 Select the size. If you do not specify the size, the size defaults to 1x1.
- 5 Specify the input parameter `numerictype` and `fimath` properties.

If you do not specify a local `fimath`, the app uses the default `fimath`. See “Default `fimath` Usage to Share Arithmetic Rules”.

To modify the `numerictype` or `fimath` properties, open the properties dialog box. To open the properties dialog box, click to the right of the fixed-point type definition.

Optionally, click .

Specify a Structure Input Parameter

When a primary input is a structure, the app 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:

- For each field of an input structure, specify class, size, and complexity.
- For each field that is a fixed-point class, also specify `numerictype`, and `fimath`.


Specify Structures by Type

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **struct**.

The app displays the selected type, `struct`. The app displays the size options.

- 4 Specify that 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, the app defines inputs as scalars.
- 5 If your input is not scalar, enter sizes for each dimension. Click the dimension. Enter the size. Select from the size options. For example, for size 10:
 - To specify fixed size, select 10.
 - To specify variable size with an upper bound of 10, select :10.
 - To specify unbounded variable size, select :Inf.
- 6 Optionally, specify properties for the structure in the generated code. See “Set Structure Properties” on page 18-15.
- 7 Add fields to the structure. Specify the class, size, and complexity of the fields. See “Add a Field to a Structure” on page 18-17.

Set Structure Properties

- 1 Click to the right of the structure definition. Optionally, click .
- 2 In the dialog box, specify properties for the structure in the generated code.

Property	Description
C type definition name	Name for the structure type 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, the app does not generate the definition of the structure type. You must provide it in a custom include file.</p> <p>Dependency: C type definition name enables this option.</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 Additional include directories parameter on the

Property	Description
	<p>project settings dialog box Custom Code tab.</p> <p>By default, the generated code contains #include 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. If you specify the C type definition header file, the app includes that header file exactly at the point where it is required.</p> <p>Must be a nonempty string.</p> <p>Dependency: When Type definition is externally defined is set to Yes, this option is enabled.</p>

Property	Description
Data alignment boundary	<p>The run-time memory alignment of structures of this type in bytes.</p> <p>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. You can take advantage of target-specific function implementations that require aligned data. By default, the structure is not aligned on any specific boundary so it is not 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: When Type definition is externally defined is set to Yes, this option is enabled.</p>

Rename a Field in a Structure

Select the name field of the structure that you want to rename. Enter the new name.

Add a Field to a Structure

- 1 To the right of the structure, click **+**
- 2 Enter the field name. Specify the class, size, and complexity of the field.

Insert a Field into a Structure

- 1 Select the structure field below which you want to add another field.
- 2 Right-click the structure field.
- 3 Select **Insert Field Below**.

The app adds the field after the field that you selected.

- 4 Enter the field name. Specify the class, size, and complexity of the field.

Remove a Field from a Structure

- 1 Right-click the field that you want to remove.
- 2 Select **Remove Field**.

Specify a Cell Array Input Parameter

For code generation, cell arrays are homogeneous or heterogeneous. See “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2. A homogeneous cell array is represented as an array in the generated code. All elements have the same properties. A heterogeneous cell array is represented as a structure in the generated code. Elements can have different properties.

Specify a Homogeneous Cell Array

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **cell (Homogeneous)**.

The app displays the selected type, `cell`. The app displays the 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, the app defines inputs as scalars.
- 5 If your input is not scalar, enter sizes for each dimension. Click the dimension. Enter the size. Select from the size options. For example, for size 10:
 - To specify fixed size, select 10.
 - To specify variable size with an upper bound of 10, select `:10`.
 - To specify unbounded variable size, select `:Inf`.

Below the cell array variable, a colon inside curly braces `{:}` indicates that the cell array elements have the same properties (class, size, and complexity).

- 6 To specify the class, size, and complexity of the elements in the cell array, click the field to the right of `{:}`.

Specify a Heterogeneous Cell Array

- 1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2 Click the field to the right of the input parameter that you want to define.
- 3 Select **cell (Heterogeneous)**.

The app displays the selected type, **cell**. The app displays the size options.


- 4 Specify that 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, the app defines inputs as scalars.
- 5 Optionally, if your input is not scalar, enter sizes m and n . A heterogeneous cell array is fixed size.

The app lists the cell array elements. It uses indexing notation to specify each element. For example, $\{1, 2\}$ indicates the element in row 1, column 2.

- 6 Specify the class, size, and complexity for each cell array element.
- 7 Optionally, add elements. See “Add an Element to a Heterogeneous Cell Array” on page 18-22
- 8 Optionally, specify properties for the structure that represents the cell array in the generated code. See “Set Structure Properties for a Heterogeneous Cell Array” on page 18-19.

Set Structure Properties for a Heterogeneous Cell Array

A heterogeneous cell array is represented as a structure in the generated code. You can specify the properties for the structure that represents the cell array.

- 1 Click to the right of the cell array definition. Optionally click .
- 2 In the dialog box, specify properties for the structure in the generated code.

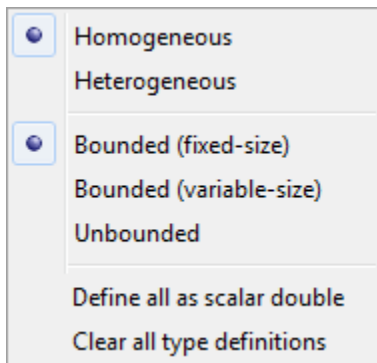
Property	Description
C type definition name	Name for the structure type in the generated code.
Type definition is externally defined	Default: No — type definition is not externally defined. If you select Yes to declare an externally defined structure, the app does not

Property	Description
	<p>generate the definition of the structure type. You must provide it in a custom include file.</p> <p>Dependency: C type definition name enables this option.</p>
C type definition header file	<p>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 Additional include directories parameter on the project settings dialog box Custom Code tab.</p> <p>By default, the generated code contains #include 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. If you specify the C type definition header file, the app includes that header file exactly at the point where it is required.</p> <p>Must be a nonempty string.</p> <p>Dependency: When Type definition is externally defined is set to Yes, this option is enabled.</p>

Property	Description
Data alignment boundary	<p>The run-time memory alignment of structures of this type in bytes.</p> <p>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. You can take advantage of target-specific function implementations that require aligned data. By default, the structure is not aligned on any specific boundary so it is not 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: When Type definition is externally defined is set to Yes, this option is enabled.</p>

Change Classification as Homogeneous or Heterogeneous

To change the classification as homogeneous or heterogeneous, right-click the variable. Select **Homogeneous** or **Heterogeneous**.



The app clears the definitions of the elements.

Change the Size of the Cell Array

- 1 In the definition of the cell array, click a dimension. Specify the size.
- 2 For a homogeneous cell array, specify whether the dimension is variable size and whether the dimension is bounded or unbounded. Alternatively, right-click the variable. Select **Bounded (fixed-size)**, **Bounded (variable-size)**, or **Unbounded**.
- 3 For a heterogeneous cell array, the app adds elements so that the cell array has the specified size and shape.

Add an Element to a Heterogeneous Cell Array

- 1 In the definition of the cell array, click a dimension. Specify the size. For example, enter 1 for the first dimension and 4 for the second dimension.

The app adds elements so that the cell array has the specified size and shape. For example for a 1x4 heterogeneous cell array, the app lists four elements: $\{1, 1\}$, $\{1, 2\}$, $\{1, 3\}$, and $\{1, 4\}$.

- 2 Specify the properties of the new elements.


Define Constant Input Parameters Using the App

- 1** On the **Define Input Types** page, click **Let me enter input or global types directly**.
- 2** Click the field to the right of the input parameter name.
- 3** Select **Define Constant**.
- 4** In the field to the right of the parameter name, enter the value of the constant or a MATLAB expression that represents the constant.

The app 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 in your MATLAB entry-point files.

To instruct the MATLAB Coder app to determine input types from the assert statements in your code, on the app toolbar, click . Select **Determine input types from code preconditions**. If you enable this option:

- The app labels all entry-point function inputs as **Deferred**. It determines the input types at compile time.
- In this project, you cannot use other input specification methods to specify input types.

See “Define Input Properties Programmatically in the MATLAB File” on page 21-67.

Note: If you enable fixed-point conversion (requires a Fixed-Point Designer license), the app disables the **Determine input types from code preconditions** option.

Add Global Variables Using the App

To add global variables to the project:

- 1 On the **Define Input Types** page, set **Does this code use global variables?** to **Yes**.

By default, the app 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, but before building the project, specify its type and initial value. Otherwise, you must create a variable with the same name in the global workspace. See “Specify Global Variable Type and Initial Value Using the App” on page 18-26.

Specify Global Variable Type and Initial Value Using the App

In this section...

“Why Specify a Type Definition for Global Variables?” on page 18-26

“Specify a Global Variable Type” on page 18-26

“Define a Global Variable by Example” on page 18-26

“Define or Edit Global Variable Type” on page 18-27

“Define Global Variable Initial Value” on page 18-28

“Define Global Variable Constant Value” on page 18-29

“Remove Global Variables” on page 18-29

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 global variable. If you do not initialize the global data, the app looks for the variable in the MATLAB global workspace. If the variable does not exist, the app generates an error.

For MEX functions, if you use global data, you must also specify whether to synchronize this data between MATLAB and the MEX function.

Specify a Global Variable Type

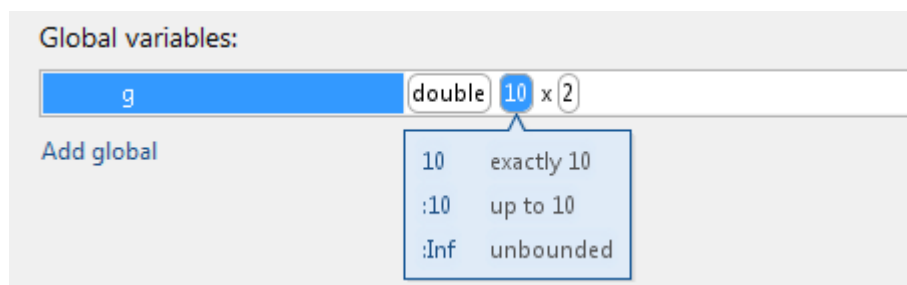
- 1 Specify the type of each global variable using one of the following methods:
 - Define by example
 - Define type
- 2 Define an initial value for each global variable.

If you do not provide a type definition and initial value for a global variable, create a variable with the same name and suitable class, size, complexity, and value in the MATLAB workspace.

Define a Global Variable by Example

- 1 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:

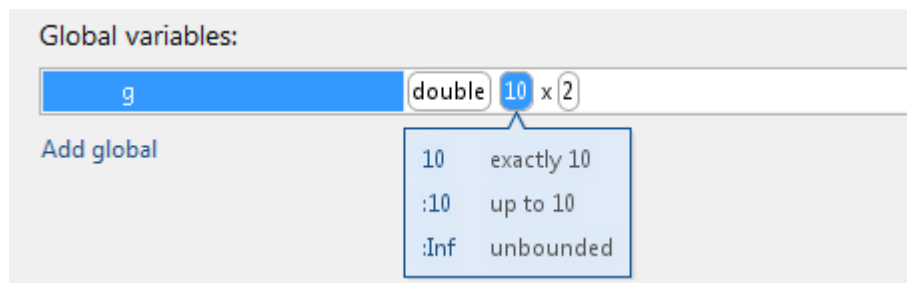
- Fixed size. In this example, select **10**.
- Variable size, up to a specified limit, by using the **:** prefix. In this example, to specify that your input can vary in size up to **10**, select **:10**.
- Unbounded variable size by selecting **:Inf**.

Define or Edit Global Variable Type

- 1 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`.

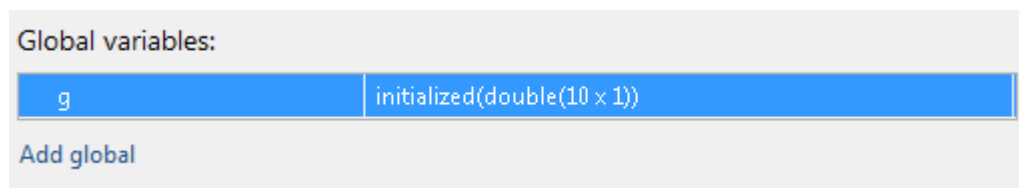
Define Global Variable Initial Value

- “Define Initial Value Before Defining Type” on page 18-28
- “Define Initial Value After Defining Type” on page 18-29

Define Initial Value Before Defining Type

- 1 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 initial value type as the global variable type.

The project shows that the global variable is initialized.

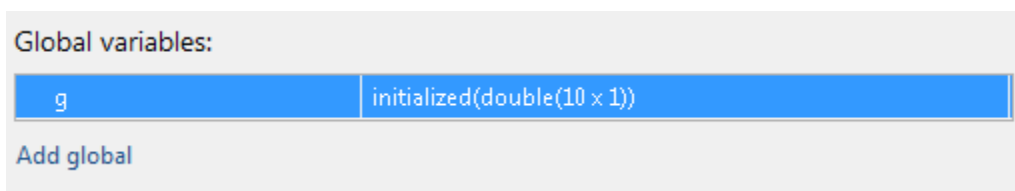


If you change the type of a global variable after defining its initial value, you must redefine the initial value.

Define Initial Value After Defining Type

- Click the type field of a predefined global variable.
- Select **Define Initial Value**.
- Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

The project shows that the global variable is initialized.



Define Global Variable Constant Value

- 1 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.

Remove Global Variables

- 1 Right-click the global variable.
- 2 From the menu, select **Remove Global**.

Changing Output Type

In this section...

“Project Settings” on page 18-30

“Configuration Object Parameters” on page 18-31

MEX functions use a different set of configuration parameters than libraries and executables use. When you switch the output type between MEX Function and Source Code, Static Library, Dynamic Library, or C/C++ Executable, verify these settings.

If you enable any of the following parameters when the output type is 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.

Project Settings

Project Settings Dialog Box Tab	Parameter Name
Paths	Working folder
	Build folder
	Search paths
Speed	Saturate on integer overflow
Memory	Enable variable-sizing
	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

Project Settings Dialog Box Tab	Parameter Name
	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

Configuration Object Parameters


- 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

Slow Operations in MATLAB Coder App

By default, the MATLAB Coder app checks your entry-point functions for code generation readiness. For some large entry-point functions, or functions with many calls, these checks can take a long time. If the checks take a long time, certain app or MATLAB operations can be slower than expected or appear to be unresponsive.

To determine if slow operations are due to the code generation readiness checks, disable the checks. If you disable the checks during execution of the checks, the app stops the checks. If you disable the checks and your code has issues, the app flags them during the **Check for Run-Time Issues** or the **Generate Code** steps.

To disable the checks for code generation readiness, on the app toolbar, click  and clear **Check code generation readiness**.

The checks remain disabled for the current project until you enable them. Unless the code generation readiness checks are too slow, it is a best practice to enable the checks.

Unable to Open a MATLAB Coder Project

When you open a project from a different release, if necessary, the MATLAB Coder app updates the project file so that the format is compatible with the release that you are using. Before the app updates the project file, it creates a backup file with the name *project_name*.prj.bak. For example, the backup file name for `myproject.prj` is `myproject.prj.bak`. If the backup file exists, the app inserts an integer between the `prj` and `bak` extensions to make the file name unique. For example, if `myproject.prj.bak` exists, the app creates the backup file `myproject.prj.2.bak`.

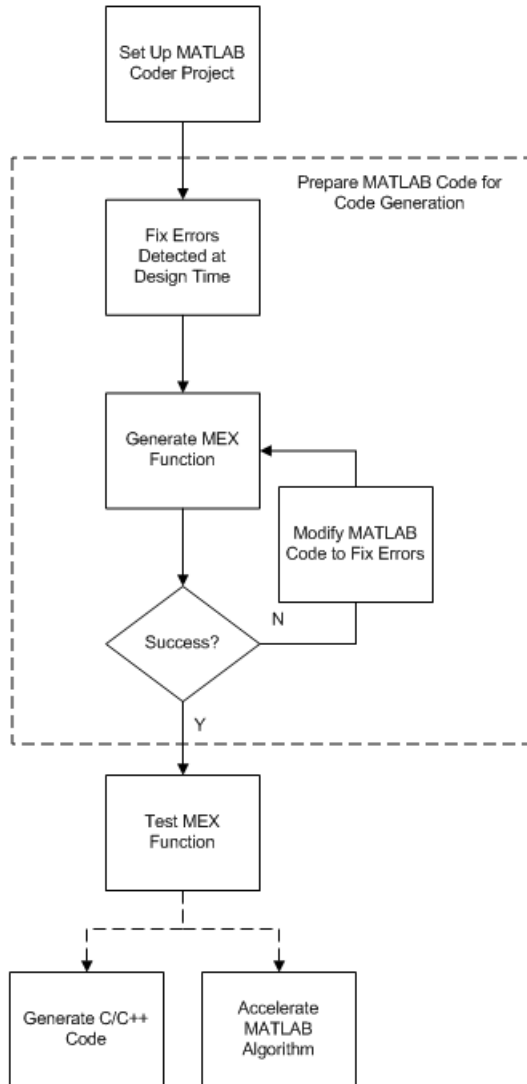
If the project file is from a release before R2015a, the app also displays a message about the project file update and backup. To use the project in a release before R2015a, use the backup project file instead of the updated project file.

To use a backup project file, remove the extensions that follow the `prj` extension. For example, rename `myproject.prj.2.bak` to `myproject.prj`. If you use the backup project file in the release that created it, the project is the same as the original project. If you use the backup project file in a different release than the one that created it, you can possibly lose some information. For example, if you open a project file in a release that does not recognize a setting in the file, that setting is lost. For best results, open the backup project file in the release in which you created it.

Preparing MATLAB Code for C/C++ Code Generation

- “Workflow for Preparing MATLAB Code for Code Generation” on page 19-2
- “Fixing Errors Detected at Design Time” on page 19-4
- “Using the Code Analyzer” on page 19-5
- “Check Code With the Code Analyzer” on page 19-6
- “Check Code Using the Code Generation Readiness Tool” on page 19-8
- “Code Generation Readiness Tool” on page 19-9
- “Unable to Determine Code Generation Readiness” on page 19-15
- “Generate MEX Functions Using the MATLAB Coder App” on page 19-16
- “Generate MEX Functions at the Command Line” on page 19-21
- “Fix Errors Detected at Code Generation Time” on page 19-23
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 19-24
- “Running MEX Functions” on page 19-26
- “Debugging Strategies” on page 19-27

Workflow for Preparing MATLAB Code for Code Generation



See Also

- “Set Up a MATLAB Coder Project” on page 18-2
- “Fixing Errors Detected at Design Time” on page 19-4
- “Generate MEX Functions Using the MATLAB Coder App” on page 19-16
- “Fix Errors Detected at Code Generation Time” on page 19-23
- “Workflow for Testing MEX Functions in MATLAB” on page 20-2
- “C/C++ Code Generation” on page 21-5
- “Accelerate MATLAB Algorithms” on page 26-14

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 19-6
- “Check Code Using the Code Generation Readiness Tool” on page 19-8
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 19-24
- “Debugging Strategies” on page 19-27

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.

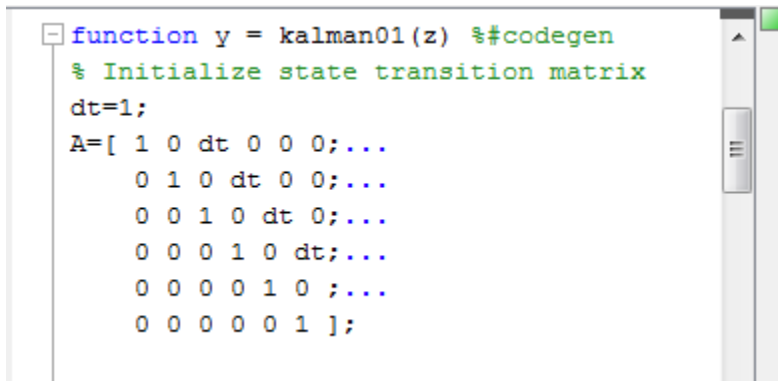
Check Code With the Code Analyzer

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.



```
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 19-8

“Run Code Generation Readiness Tool from the Current Folder Browser” on page 19-8

“Run the Code Generation Readiness Tool Using the MATLAB Coder App” on page 19-8

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 command prompt, enter:

```
coder.screener('filename')
```

The **Code Generation Readiness** tool opens for the file named `filename`. The tool provides a code generation readiness score and lists issues that you must fix 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 you must fix prior to code generation.

Run the Code Generation Readiness Tool Using the MATLAB Coder App

After you add entry-point files to your project, the MATLAB Coder app analyzes the functions for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page. You can review and fix issues.

See “Code Generation Readiness Tool” on page 19-9.

Code Generation Readiness Tool

In this section...

“Information That the Code Generation Readiness Tool Provides” on page 19-9

“Summary Tab” on page 19-10

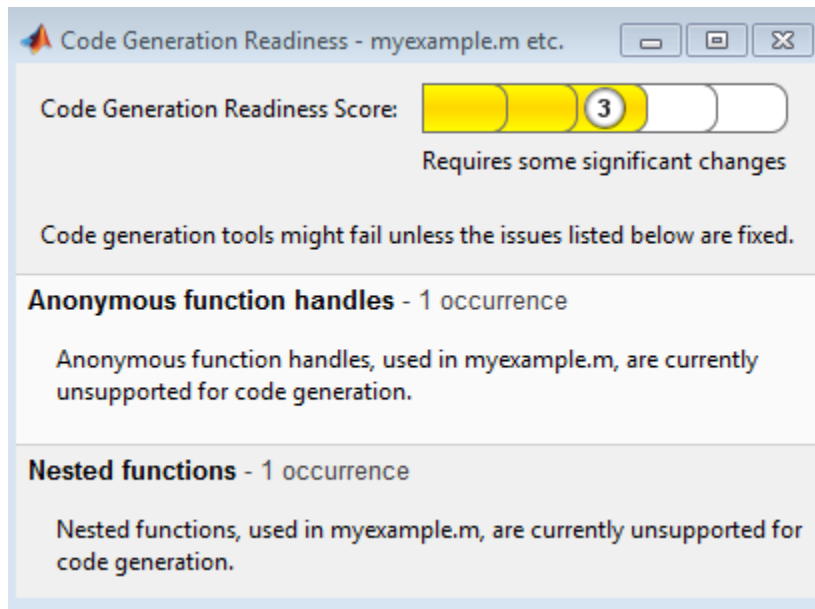
“Code Structure Tab” on page 19-11

“See Also” on page 19-14

Information That the Code Generation Readiness Tool Provides

The code generation readiness tool screens MATLAB code for features and functions that code generation does not support. The tool provides a report that lists the source files that contain unsupported features and functions. The report also indicates the amount of work required to make the MATLAB code suitable for code generation. It is possible that the tool does not detect all code generation issues. Under certain circumstances, it is possible that the tool can report false errors. Therefore, before you generate C code, verify that your code is suitable for code generation by generating a MEX function.

Summary Tab

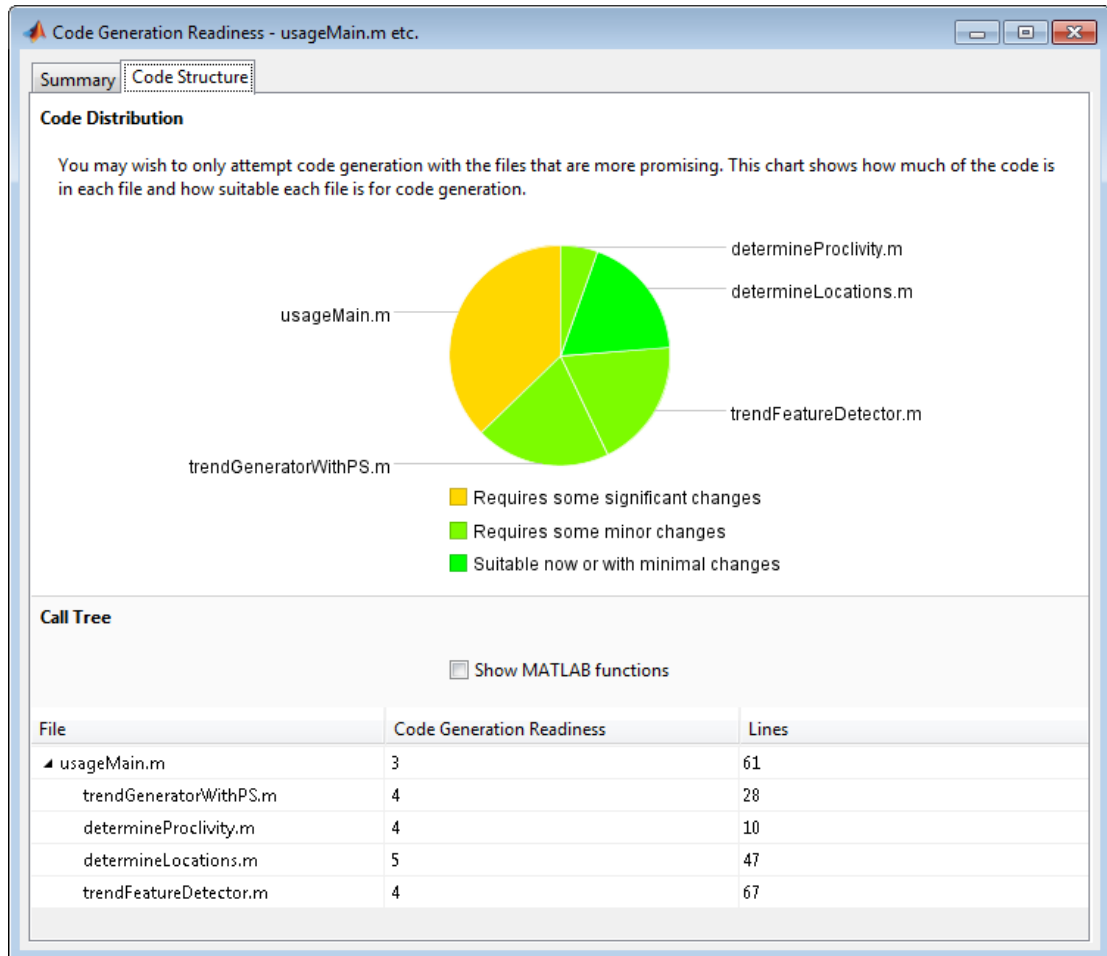


The **Summary** tab provides a **Code Generation Readiness Score**, which ranges from 1 to 5. A score of 1 indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool does not detect code generation issues; the code is ready to use with minimal or no changes.

On this tab, the tool also displays information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. To learn more about the issues and how to fix them, use the Code Analyzer.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features, such as anonymous function handles 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 displays information about the relative size of each file and how suitable each file is for code generation.

Code Distribution

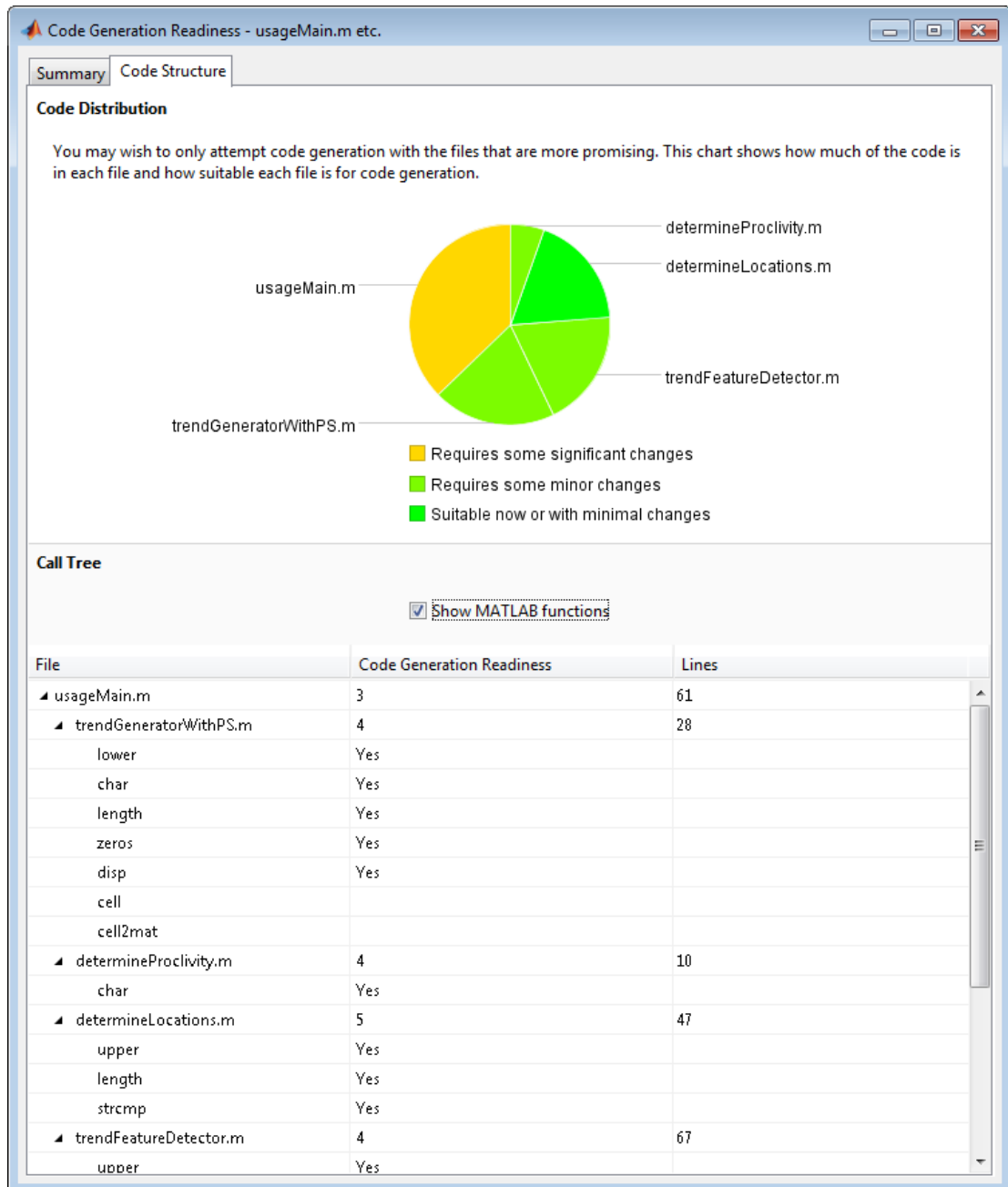
The **Code Distribution** pane displays a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. During the planning phase of a project, you can use this information for estimation and scheduling. If the report indicates that multiple files are not suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane displays information about 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 detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool does not detect code generation issues. The code is ready to use with minimal or no 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 that your function calls. For each of these MATLAB functions, if code generation supports the function, the report sets **Code Generation Readiness** to **Yes**.



See Also

- “Check Code Using the Code Generation Readiness Tool” on page 19-8

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 App

In this section...

“Workflow for Generating MEX Functions Using the MATLAB Coder App” on page 19-16

“Generate a MEX Function Using the MATLAB Coder App” on page 19-16

“Configure Project Settings” on page 19-19

“Build a MATLAB Coder Project” on page 19-19

“See Also” on page 19-20

Workflow for Generating MEX Functions Using the MATLAB Coder App

Step	Action	Details
1	Set up the MATLAB Coder project.	“Set Up a MATLAB Coder Project” on page 18-2
2	Specify the build configuration parameters. Set Build type to MEX.	“Configure Project Settings” on page 19-19
3	Build the project.	“Build a MATLAB Coder Project” on page 19-19

Generate a MEX Function Using the MATLAB Coder App

This example shows how to generate a MEX function from MATLAB code using the MATLAB Coder app.

Create the Entry-Point Function

In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

Create the Test File

In the same local writable folder, create a MATLAB file, `mcadd_test.m`, that calls `mcadd` with example inputs. The example inputs are scalars with type `int16`.


```
function y = mcadd_test
y = mcadd(int16(2), int16(3));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

- 1 On the **Select Source Files** page, type or select the name of the entry-point function `mcadd`.

The app creates a project with the default name `mcadd.prj`.

- 2 Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify the test file `mcadd_test.m` that MATLAB Coder uses to automatically define types for `u` and `v`:

- 1 Enter or select the test file `mcadd_test.m`.
- 2 Click **Autodefine Input Types**.

The test file, `mcadd_test.m`, calls the entry-point function, `mcadd`, with the example input types. MATLAB Coder infers that inputs `u` and `v` are `int16(1x1)`.

- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it

is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

- 1 To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .


The app populates the test file field with `mcadd_test`, the test file that you used to define the input types.

- 2 Click **Check for Issues**.

The app generates a MEX function. It runs the test file replacing calls to `mcadd` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

- 3 Click **Next** to go to the **Generate Code** step.

Generate the MEX Function

- 1 To open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set **Build type** to MEX and **Language** to C. Use the default values for the other project build configuration settings.
- 3 Click **Generate**.

The app indicates that code generation succeeded. It displays the source MATLAB files and the generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Build Log** tab, it displays the build log, including compiler warnings and errors.

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. It creates a platform-specific extension for the MEX file. See “Naming Conventions” on page 21-82.

- 4 To view the code generation report, click **View Report**.
- 5 Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to the generated output.

Configure Project Settings

To open the project settings dialog box:

- 1 To open the **Generate** dialog box, click the **Generate** arrow .
- 2 Click **More Settings**.

To change a project setting, click the tab that contains the setting that you want to change. For example, to change the **Saturate on integer overflow** setting, click the **Speed** tab.

MEX functions use a different set of configuration parameters than libraries and executables. When you change the output type from **MEX Function** to **Source Code Static Library**, **Dynamic Library**, or **Executable**, verify these settings. See “Changing Output Type” on page 18-30.

See Also

- “Enable Code Generation Reports” on page 22-27
- “Using the MATLAB Coder App” on page 21-125
- “How to Disable Inlining Globally Using the MATLAB Coder App” on page 21-135
- “Generate Traceable Code” on page 22-2
- “Disabling Run-Time Checks Using the MATLAB Coder App” on page 26-17

Build a MATLAB Coder Project

To build a project using the specified settings, on the **Generate Code** page, click **Generate**. As the MATLAB Coder app builds a project, it displays the build progress. When the build is complete, the app provides details about the build on the **Build Log** tab.

If the code generation report is enabled or build errors occur, the app generates a report. The report provides detailed information about the most recent build, and provides a link to the report.

To view the report, click the **View report** link. The report provides links to your MATLAB code and generated C/C++ files and compile-time type information for the variables in your MATLAB code. If build errors occur, the report lists errors and warnings.

See Also

- “Code Generation Reports” on page 22-10
- “Generate Code for Multiple Entry-Point Functions” on page 21-87
- “Generate Code for Global Data” on page 21-93

See Also

- “Generate Code for Multiple Entry-Point Functions” on page 21-87
- “Generate Code for Global Data” on page 21-93
- “Configure Build Settings” on page 21-27

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 21-81
3	Fix errors detected by the code analyzer.	“Fixing Errors Detected at Design Time” on page 19-4
4	Specify build configuration parameters.	“Specify Build Configuration Parameters” on page 21-33
5	Specify properties of primary function inputs.	“Primary Function Input Specification” on page 21-46
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 19-22

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 21-82.

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” on page 19-21.

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” on page 21-46
- “MEX Function Generation at the Command Line”
- “Generate Code for Multiple Entry-Point Functions” on page 21-87
- “Generate Code for Global Data” on page 21-93

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 “Programming Considerations for Code Generation”. Choose a debugging strategy for detecting and correcting code generation errors in your MATLAB code. For more information, see “Debugging Strategies” on page 19-27.

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” on page 14-2. To get detailed diagnostics, add the `%#codegen` directive to each external function that you want `codegen` to compile.

See Also

- “Code Generation Reports” on page 22-10
- “Why Test MEX Functions in MATLAB?” on page 20-4
- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Debugging Strategies” on page 19-27
- “Declaring MATLAB Functions as Extrinsic Functions” on page 14-12

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

- “Programming Considerations for Code Generation”
- “Data Definition”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data” on page 7-4
- “Control Dynamic Memory Allocation” on page 21-107
- “Control Run-Time Checks” on page 26-16

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

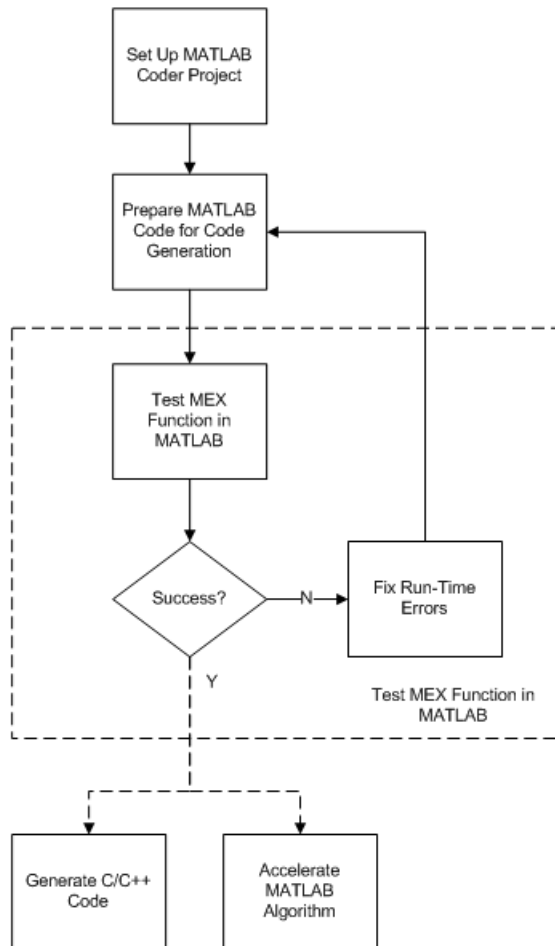
Before you perform code verification, choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other's functions. The following table describes two general strategies, each of which has advantages and disadvantages.

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	<ol style="list-style-type: none"> 1 Verify that your lowest-level (leaf) functions are compliant. 2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function. 	<ul style="list-style-type: none"> • Efficient • Unlikely to cause errors • Easy to isolate code generation syntax violations 	Requires application tests that work from the bottom up
Top-down verification	<ol style="list-style-type: none"> 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” on page 14-12. 2 Verify that your top-level function is compliant. 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. 	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"> • Extrinsic declarations • Additional assignment statements as required to convert opaque values returned by extrinsic functions to nonopaque values (see “Working with mxArray” on page 14-17).

Testing MEX Functions in MATLAB

- “Workflow for Testing MEX Functions in MATLAB” on page 20-2
- “Why Test MEX Functions in MATLAB?” on page 20-4
- “Running MEX Functions” on page 20-5
- “Verify MEX Functions Using the MATLAB Coder App” on page 20-6
- “Verify MEX Functions at the Command Line” on page 20-7
- “Debug Run-Time Errors” on page 20-8
- “Using MEX Functions That MATLAB Coder Generates” on page 20-11
- “Using Generated C/C++ Code” on page 20-12

Workflow for Testing MEX Functions in MATLAB



See Also

- “Set Up a MATLAB Coder Project” on page 18-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 19-2

- “Why Test MEX Functions in MATLAB?” on page 20-4
- “Debug Run-Time Errors” on page 20-8
- “C/C++ Code Generation” on page 21-5
- “Accelerate MATLAB Algorithms” on page 26-14

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 Using the MATLAB Coder App” on page 20-6. For more information on how to test a MEX function at the command line, see “Verify MEX Functions at the Command Line” on page 20-7.

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. See “Fix Errors Detected at Code Generation Time” on page 19-23 and “Debug Run-Time Errors” on page 20-8.

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” on page 26-16.

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 Using the MATLAB Coder App

In the MATLAB Coder app, after you generate a MEX function, you can verify that the generated MEX function has the same functionality as the original MATLAB entry-point function. Provide a test file that calls the original MATLAB entry-point function. The test file can be a function or a script. The test file must be in the same folder as the original entry-point function.

- 1 On the **Generate Code** page, click **Verify Code**.
- 2 Type or select the test file name.
- 3 To run the test file without replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **MATLAB code**. Click **Run Generated Code**.
- 4 To run the test file, replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **Generated code**. Click **Run Generated Code**.
- 5 Compare the results of running the original MATLAB function with the results of running the MEX function.

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 20-8
“Handling Run-Time Errors” on page 20-10

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 20-8 “Handling Run-Time Errors” on page 20-10.

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” on page 26-17.

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 20-9, 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”.

Using MEX Functions That MATLAB Coder Generates

When you specify MEX for the output (build) type, MATLAB Coder generates a binary MATLAB executable (MEX) version of your MATLAB function. You can call the MEX function from MATLAB. See “Call MEX-File Functions”.

How you use the MEX function depends on your goal.

Goal	See
Accelerate your MATLAB function.	“MATLAB Algorithm Acceleration”
Test generated function for functionality and run-time issues.	“Why Test MEX Functions in MATLAB?” on page 20-4
Debug your MEX function.	“Debug Run-Time Errors” on page 20-8

Using Generated C/C++ Code

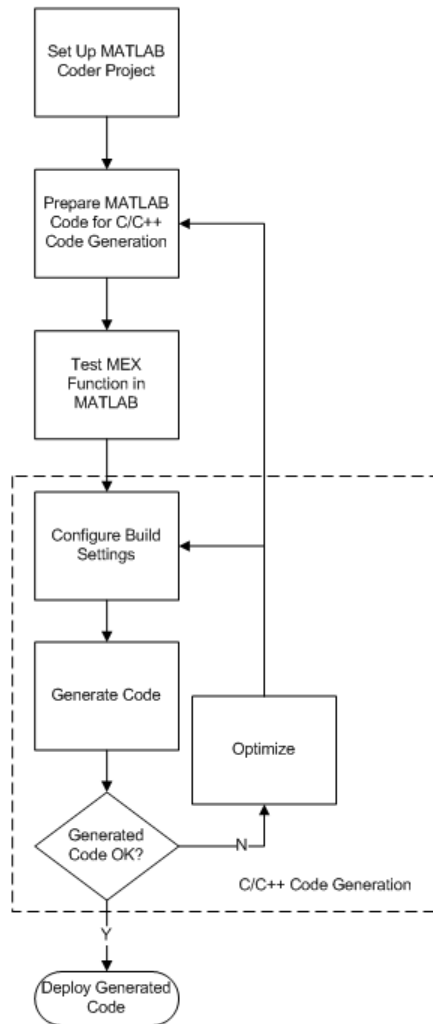
Goal	See
Package generated files into a zip file for relocation to another development environment.	“Package Code for Other Development Environments” on page 25-44
Call generated code from MATLAB code.	“External Code Integration for Code Generation” on page 28-2
Generate an example C/C++ main function. Use that function to integrate generated code into a C application.	“Use an Example C Main in an Application” on page 25-21
Integrate generated code into a C/C++ application.	“Use a C Dynamic Library in a Microsoft Visual Studio Project” on page 25-9
Integrate generated code that uses <code>emxArrays</code> .	<ul style="list-style-type: none">• “Use an Example C Main in an Application” on page 25-21• “C Code Interface for Arrays” on page 7-17
Generate a C/C++ Executable.	“Generating Standalone C/C++ Executables from MATLAB Code” on page 21-15

Generating C/C++ Code from MATLAB Code

- “Code Generation Workflow” on page 21-3
- “C/C++ Code Generation” on page 21-5
- “Generating C/C++ Static Libraries from MATLAB Code” on page 21-6
- “Generating C/C++ Dynamically Linked Libraries from MATLAB Code” on page 21-10
- “Generating Standalone C/C++ Executables from MATLAB Code” on page 21-15
- “Configure Build Settings” on page 21-27
- “Specify Data Types Used in Generated Code” on page 21-39
- “Standard Math Libraries” on page 21-40
- “Change the Standard Math Library” on page 21-41
- “Share Build Configuration Settings” on page 21-42
- “Convert MATLAB Coder Project to MATLAB Script” on page 21-44
- “Primary Function Input Specification” on page 21-46
- “Specify Cell Array Inputs at the Command Line” on page 21-56
- “Control Constant Inputs in MEX Function Signatures” on page 21-63
- “Define Input Properties Programmatically in the MATLAB File” on page 21-67
- “Speed Up Compilation” on page 21-79
- “Disable Creation of the Code Generation Report” on page 21-80
- “Paths and File Infrastructure Setup” on page 21-81
- “Generate Code for Multiple Entry-Point Functions” on page 21-87
- “Generate Code for Global Data” on page 21-93
- “Specify Global Cell Arrays at the Command Line” on page 21-103
- “Generate Code for Enumerated Types” on page 21-105

- “Generate Code for Variable-Size Data” on page 21-106
- “Code Generation for MATLAB Classes” on page 21-124
- “How MATLAB Coder Partitions Generated Code” on page 21-125
- “Requirements for Signed Integer Representation” on page 21-138
- “Customize the Post-Code-Generation Build Process” on page 21-139
- “Troubleshooting” on page 21-177

Code Generation Workflow



See Also

- “Set Up a MATLAB Coder Project” on page 18-2

- “Workflow for Preparing MATLAB Code for Code Generation” on page 19-2
- “Workflow for Testing MEX Functions in MATLAB” on page 20-2
- “Configure Build Settings” on page 21-27
- “C/C++ Code Generation” on page 21-5

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 21-6
C/C++ dynamic libraries from your MATLAB code	“Generating C/C++ Dynamically Linked Libraries from MATLAB Code” on page 21-10
C/C++ executables from your MATLAB code	“Generating Standalone C/C++ Executables from MATLAB Code” on page 21-15
MEX functions from your MATLAB code	“Generate MEX Functions Using the MATLAB Coder App” on page 19-16

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 22-10.

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 MATLAB Coder App” on page 21-6

“Generate a C Static Library at the Command Line” on page 21-8

Generate a C Static Library Using the MATLAB Coder App

This example shows how to generate a C static library from MATLAB code using the MATLAB Coder app.

In this example, you create a MATLAB function that adds two numbers. You use the app to create a MATLAB Coder project and generate a C static library for the MATLAB code.

Create the Entry-Point Function

In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

Create the Test File

In the same local writable folder, create a MATLAB file, `mcadd_test.m`, that calls `mcadd` with example inputs. The example inputs are scalars with type `int16`.

```
function y = mcadd_test
y = mcadd(int16(2), int16(3));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

- 1 On the **Select Source Files** page, type or select the name of the entry-point function `mcadd`.

The app creates a project with the default name `mcadd.prj` in the current folder.

- 2 Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify the test file `mcadd_test.m` that MATLAB Coder can use to automatically define types for `u` and `v`:

- 1 Enter or select the test file `mcadd_test.m`.
- 2 Click **Autodefine Input Types**.

The test file, `mcadd_test.m`, calls the entry-point function, `mcadd` with the example input types. MATLAB Coder infers that inputs `u` and `v` are `int16(1x1)`.

- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

- 1 To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .

The app populates the test file field with `mcadd_test`, the test file that you used to define the input types.


- 2 Click **Check for Issues**.

The app generates a MEX function. It runs the test file replacing calls to `mcadd` with calls to the MEX function. If the app detects issues during the MEX function

generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

- 3 Click **Next** to go to the **Generate Code** step.

Generate C Code

- 1 To open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set **Build type** to **Static Library (.lib)** and **Language** to **C**. Use the default values for the other project build configuration settings.
- 3 Click **Generate**.

The app indicates that code generation succeeded. It displays the source MATLAB files and generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Build Log** tab, it displays the build log, including compiler warnings and errors. By default, in the code window, the app displays the C source code file, `mcadd.c`. To view a different file, in the **Source Code** or **Output Files** pane, click the file name.

MATLAB Coder generates a standalone C static library `mcadd` in the `codegen\lib\mcadd` folder. It generates the minimal set of `#include` statements for header files required by the selected code replacement library.

- 4 To view the code generation report, click **View Report**.
- 5 Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to generated output.

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 21-10

“Generate a C Dynamically Linked Library Using the MATLAB Coder App” on page 21-10

“Generate a C Dynamic Library at the Command Line” on page 21-13

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.
- When an executable that uses the DLL runs, the DLL must be on the system path so that the executable can access it.

If you generate a DLL that uses dynamically allocated variable-size data, MATLAB Coder provides exported utility functions to interact with this data in the generated code. For more information, see “Utility Functions for Creating `emxArray` Data Structures” on page 7-20.

Generate a C Dynamically Linked Library Using the MATLAB Coder App

This example shows how to generate a C DLL from MATLAB code using the MATLAB Coder app.

Create the Entry-Point Functions

Write two MATLAB functions, `ep1` and `ep2`. `ep1` takes one input, a single scalar. `ep2` takes two inputs that are double scalars. In a local writable folder:

- 1 Create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

- 2 Create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

Create the Test File

In the folder that contains `ep1.m` and `ep2.m`, create a MATLAB file, `ep_test.m`, that calls `ep1` and `ep2` with example inputs.

```
function [y, y1] = ep_test
y = ep1(single(2));
y1 = ep2(double(3), double(4));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

- 1 On the **Select Source Files** page, type or select the name of the entry-point function `ep1`.

The app creates a project with the default name `ep1.prj` in the current folder.

- 2 To add `ep2` to the list of entry-point functions, click **Add Entry-Point Function**. Type or select the name of the entry-point function `ep2`.
- 3 Click **Next** to go to the **Define Input Types** step. The app analyzes the functions for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all

entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify a test file that MATLAB Coder can use to automatically define types:

- 1 Enter or select the test file `ep_test.m`.
- 2 Click **Autodefine Input Types**.

The test file, `eps_test.m`, calls the entry-point functions `ep1` and `ep2` with the example input types. MATLAB Coder infers that for `ep1`, input `u` is `single(1x1)`. For `ep2`, `u` and `v` are `double(1x1)`.

- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

- 1 To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .


The app populates the test file field with `ep_test`, the test file that you used to define the input types.

- 2 Click **Check for Issues**.

The app generates a MEX function named `ep1_mex` for `ep1` and `ep2`. It runs the test file `ep_test` replacing calls to `ep1` and `ep2` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

- 3 Click **Next** to go to the **Generate Code** step.

Generate C Code

- 1 To open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set **Build type** to **Dynamic Library** and **Language** to **C**. Use the default values for the other project build configuration settings.

3 Click **Generate**.

The app indicates that code generation succeeded. It displays the source MATLAB files and generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Build Log** tab, it displays the build log, including compiler warnings and errors. By default, in the code window, the app displays the C source code file, `ep1.c`. To view a different file, in the **Source Code** or **Output Files** pane, click the file name.

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++. MATLAB Coder generates a standalone C static library `mcadd` in the `codegen\lib\mcadd` folder.

4 To view the code generation report, click **View Report**.

5 Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to generated output.

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 local writable 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 21-29.

Generating Standalone C/C++ Executables from MATLAB Code

In this section...

“Generate a C Executable Using the MATLAB Coder App” on page 21-15

“Generate a C Executable at the Command Line” on page 21-24

“Specifying main Functions for C/C++ Executables” on page 21-25

“Specify main Functions” on page 21-26

Generate a C Executable Using the MATLAB Coder App

This example shows how to generate a C executable from MATLAB code using the MATLAB Coder app. In this example, you generate an executable for a MATLAB function that generates a random scalar value. Using the app, you:

- 1 Generate a C `main` function that calls the generated library function.
- 2 Copy and modify the generated `main.c` and `main.h`.
- 3 Modify the project settings so that the app can find the modified `main.c` and `main.h`.
- 4 Generate the executable.

Create the Entry-Point Function

In a local writable folder, create 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();
```

Create the Test File

In the same local writable folder, create a MATLAB file, `coderand_test.m`, that calls `coderand`.

```
function y = coderand_test()
y = coderand();
```

Open the MATLAB Coder app

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

- 1 On the **Select Source Files** page, type or select the name of the entry-point function `coderand`.

The app creates a project with the default name `coderand.prj` in the current folder.

- 2 Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types


Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

In this example, the function `coderand` does not have inputs.

Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

- 1 To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .

Select or enter the test file `coderand_test`.

- 2 Click **Check for Issues**.

The app generates a MEX function for `coderand`. It runs the test file replacing calls to `coderand` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages.


Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

- 3 Click **Next** to go to the **Generate Code** step.

Generate a C main Function

When you generate an executable, you must provide a C/C++ function. By default, when you generate C/C++ source code, static libraries, dynamically linked libraries, or executables, MATLAB Coder generates a `main` function. This generated main function is a template that you modify for your application. See “Incorporate Generated Code Using an Example Main Function” on page 25-18. After you copy and modify the generated main function, you can use it for generation of the C/C++ executable. Alternatively, you can write your own main function.

Before you generate the executable for `coderand`, generate a `main` function that calls `coderand`.

- 1 To open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set **Build type** to **Source Code** and **Language** to **C**. Use the default values for the other project build configuration settings.
- 3 Click **More Settings**.
- 4 On the **All Settings** tab, under **Advanced**, verify that **Generate example main** is set to **Generate**, but do not compile, an example main function. Click **Close**.
- 5 Click **Generate**.

MATLAB Coder generates a `main.c` file and a `main.h` file. The app indicates that code generation succeeded.

- 6 Click **Next** to open the **Finish Workflow** page.

On the **Finish Workflow** page, under **Generated Output**, you see that `main.c` is in the subfolder `coderand\codegen\lib\coderand\examples`.

Copy the Generated Example Main Files

Because subsequent code generation can overwrite the generated example files, before you modify these files, copy them to a writable folder outside of the `codegen` folder. For this example, copy `main.c` and `main.h` from the subfolder `coderand\codegen\lib\coderand\examples` to a writable folder, for example, `c:\myfiles`.

Modify the Generated Example Main Files

- 1 In the folder that contains a copy of the example main files, open `main.c`.

Generated `main.c`

```
/* ***** */
/* This automatically generated example C main file shows how to call */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly. */
/* Instead, make a copy of this file, modify it, and integrate it into */
/* your development environment. */
/* */
/* This file initializes entry-point function arguments to a default */
/* size and value before calling the entry-point functions. It does */
/* not store or use any values returned from the entry-point functions. */
/* If necessary, it does pre-allocate memory for returned values. */
/* You can use this file as a starting point for a main function that */
/* you can deploy in your application. */
/* */
/* After you copy the file, and before you deploy it, you must make the */
/* following changes: */
/* * For variable-size function arguments, change the example sizes to */
/* the sizes that your application requires. */
/* * Change the example values of function arguments to the values that */
/* your application requires. */
/* * If the entry-point functions return values, store these values or */
/* otherwise use them as required by your application. */
/* */
/* ***** */
/* Include Files */
#include "rt_nonfinite.h"
#include "coderand.h"
#include "main.h"
#include "coderand_terminate.h"
#include "coderand_initialize.h"

/* Function Declarations */
static void main_coderand(void);

/* Function Definitions */

/*
 * Arguments      : void
 */
```

```
* Return Type : void
*/
static void main_coderand(void)
{
    double r;

    /* Call the entry-point 'coderand'. */
    r = coderand();
}

/*
* Arguments      : int argc
*                  const char * const argv[]
* Return Type    : int
*/
int main(int argc, const char * const argv[])
{
    (void)argc;
    (void)argv;

    /* Initialize the application.
       You do not need to do this more than one time. */
    coderand_initialize();

    /* Invoke the entry-point functions.
       You can call entry-point functions multiple times. */
    main_coderand();

    /* Terminate the application.
       You do not need to do this more than one time. */
    coderand_terminate();
    return 0;
}

/*
* File trailer for main.c
*
* [EOF]
*/
```

2 Modify `main.c` so that it prints the results of a `coderand` call:

- In `main_coderand`, delete the line

```
double r;
```

- In `main_coderand`, replace

```
r = coderand()  
with
```

```
printf("coderand=%g\n", coderand());
```

- For this example, `main` does not have arguments. In `main`, delete the lines:

```
(void)argc;  
(void)argv;
```

Change the definition of `main` to

```
int main()
```

Modified `main.c`

```
/* Include Files */  
#include "rt_nonfinite.h"  
#include "coderand.h"  
#include "main.h"  
#include "coderand_terminate.h"  
#include "coderand_initialize.h"  
  
/* Function Declarations */  
static void main_coderand(void);  
  
/* Function Definitions */  
  
/*  
 * Arguments      : void  
 * Return Type   : void  
 */  
static void main_coderand(void)  
{  
    /* Call the entry-point 'coderand'. */  
    printf("coderand=%g\n", coderand());  
}  
  
/*  
 * Arguments      : int argc  
 *                 const char * const argv[]  
 * Return Type   : int  
 */
```

```

int main()
{
    /* Initialize the application.
       You do not need to do this more than one time. */
    coderand_initialize();

    /* Invoke the entry-point functions.
       You can call entry-point functions multiple times. */
    main_coderand();

    /* Terminate the application.
       You do not need to do this more than one time. */
    coderand_terminate();
    return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */

```

3 Open main.h

Generated main.h

```

*****/
/* This automatically generated example C main file shows how to call */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly. */
/* Instead, make a copy of this file, modify it, and integrate it into */
/* your development environment. */
/* */
/* This file initializes entry-point function arguments to a default */
/* size and value before calling the entry-point functions. It does */
/* not store or use any values returned from the entry-point functions. */
/* If necessary, it does pre-allocate memory for returned values. */
/* You can use this file as a starting point for a main function that */
/* you can deploy in your application. */
/* */
/* After you copy the file, and before you deploy it, you must make the */
/* following changes: */
/* * For variable-size function arguments, change the example sizes to */
/* the sizes that your application requires. */

```

```
/* * Change the example values of function arguments to the values that */
/* your application requires. */
/* * If the entry-point functions return values, store these values or */
/* otherwise use them as required by your application. */
/* */
/*****
#endif __MAIN_H__
#define __MAIN_H__

/* Include Files */

#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include "rtwtypes.h"
#include "coderand_types.h"

/* Function Declarations */
extern int main(int argc, const char * const argv[]);

#endif

/*
 * File trailer for main.h
 *
 * [EOF]
 */
```

4 Modify main.h:

- Add `stdio` to the include files:

```
#include <stdio.h>
```
- Change the declaration of `main` to

```
extern int main()
```

Modified main.h

```
#ifndef __MAIN_H__
#define __MAIN_H__

/* Include Files */
```

```

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include "rtwtypes.h"
#include "coderand_types.h"



/* Function Declarations */
extern int main();

#endif

/*
 * File trailer for main.h
 *
 * [EOF]
 */

```

Generate the Executable

- 1 To open the **Generate Code** page, expand the workflow steps  and click **Generate**
- 2 To open the **Generate** dialog box, click the **Generate** arrow .
- 3 Set **Build type** to Executable (.exe).
- 4 Click **More Settings**.
- 5 On the **Custom Code** tab, in **Additional source files**, enter `main.c`
- 6 On the **Custom Code** tab, in **Additional include directories**, enter the location of the modified `main.c` and `main.h` files. For example, `c:\myfiles`. Click **Close**.
- 7 To generate the executable, click **Generate**.

The app indicates that code generation succeeded.

- 8 Click **Next** to go to the **Finish Workflow** step.
- 9 Under **Generated Output**, you can see the location of the generated executable `coderand.exe`.

Run the Executable

To run the executable in MATLAB on a Windows platform:

```
system('coderand')
```

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. For a C executable, provide a C file, `main.c`. For 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.


By default, when you generate C/C++ source code, static libraries, dynamically linked libraries, or executables, MATLAB Coder generates a `main` function. This generated main function is a template that you modify for your application. See “Incorporate Generated Code Using an Example Main Function” on page 25-18. After you copy and modify the generated main function, you can use it for generation of the C/C++ executable. Alternatively, you can write your own main function.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder 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 25-7.

Specify main Functions

Specifying main Functions Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 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 21-25.
 - 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 21-34). 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 21-25.) 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 myFunction -args {0}
```

MATLAB Coder compiles and links the main function with the C/C++ code that it generates from `myFunction.m`.

Configure Build Settings

In this section...

“Specify Build Type” on page 21-27

“Specify a Language for Code Generation” on page 21-29

“Specify Output File Name” on page 21-31

“Specify Output File Locations” on page 21-31

“Parameter Specification Methods” on page 21-33

“Specify Build Configuration Parameters” on page 21-33

Specify Build Type

Build Types

MATLAB Coder can generate code for the following output types:

- MEX function
- Standalone C/C++ code
- 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 21-25.

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 21-131.

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.

Specify the Build Type Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to one of the following.
 - Source Code
 - MEX
 - Static Library
 - Dynamic Library
 - Executable

If you select **Source Code**, 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. This option is equivalent to **Static Library** with the **Generate code only** box selected. See “Generate Code Only” on page 21-79.

Code generation uses a different set of configuration parameters for MEX functions than it uses for the other build types. . When you switch the output type between **MEX Function** and **Source**, **Static Library**, **Dynamic Library**, or **Executable**, verify these settings. For more information, see “Changing Output Type” on page 18-30.

Specifying the Build Type at the Command Line

Call `codegen` with the `-config` option. For example, suppose that 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 21-46.

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 21-29.

To Generate:	Use This Command:
MEX function using the default code generation options	<code>codegen foo</code>

To Generate:	Use This Command:
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 codegen -config cfg foo</pre>
Standalone C/C++ code and compile it to a library using the default code generation options	<pre>codegen -config:lib foo</pre>
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>Note: You must specify a <code>main</code> function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 21-25</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>Note: You must specify a <code>main</code> function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 21-25</p>


Specify a Language for Code Generation

- “Specify the Language Using the MATLAB Coder App” on page 21-30

- “Specifying the Language Using the Command-Line Interface” on page 21-30

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.

Specify the Language Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Language** to C or C++.

Note: If you specify C++, MATLAB Coder wraps the C code into `.cpp` files. You can use a C++ compiler and interface with external C++ applications. MATLAB Coder does not generate C++ classes.

Specifying the Language Using the Command-Line Interface

- 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 21-34
- “Setting Up the C or C++ Compiler”

Specify Output File Name

Specify Output File Name Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 In the **Output file name** field, enter the file name.

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

Specify Output File Location Using the MATLAB Coder App

The output file location must not contain:

- Spaces (Spaces can lead to code generation failures in certain operating system configurations).
- Tabs
- \, \$, #, *, ?
- Non-7-bit ASCII characters, such as Japanese characters.

- 1** To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2** Set **Build type** to **Source Code**, **Static Library**, **Dynamic Library**, or **Executable** (depending on your requirements).
- 3** Click **More Settings**.
- 4** 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 alphabetically 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
- 5** 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
The MATLAB Coder app	The project settings dialog box.	“Specify Build Configuration Parameters MATLAB Coder App” on page 21-33
codegen at the command line and want to specify a few parameters	Configuration objects	“Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 21-34
codegen in build scripts		
codegen at the command line and want to specify many parameters	Configuration object dialog boxes	“Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 21-38

Specify Build Configuration Parameters

- “Specify Build Configuration Parameters MATLAB Coder App” on page 21-33
- “Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 21-34
- “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 21-38

You can specify build configuration parameters from the MATLAB Coder project settings dialog box, the command line, or configuration object dialog boxes.

Specify Build Configuration Parameters MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to **Source Code**, **Static Library**, **Dynamic Library**, or **Executable** (depending on your requirements).
- 3 Click **More Settings**.

The project settings dialog box provides the set of configuration parameters applicable to the output type that you select. Code generation uses a different set of configuration parameters for MEX functions than it uses for the other build types.

When you switch the output type between MEX Function and Source Code, Static Library, Dynamic Library, or Executable, verify these settings. See “Changing Output Type” on page 18-30.

- 4 Modify the parameters as required. For more information about parameters on a tab, click **Help**.

Changes to the parameter settings take place immediately.

Specify 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>
<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 21-36.

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 “Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 21-34
 - Dialog boxes, as described in “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 21-38
- 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 coder.MexCodeConfig	<code>cfg = coder.config('mex');</code>
Code generation configuration object for generating a standalone C/C++ library or executable coder.CodeConfig	<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>Note: If an Embedded Coder license is available, creates a <code>coder.EmbeddedCodeConfig</code> object.</p> <p>If you use concurrent licenses, to disable the 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 coder.EmbeddedCodeConfig	<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>Note: Requires an Embedded Coder license; otherwise creates a <code>coder.CodeConfig</code> object.</p>
Hardware implementation configuration object coder.HardwareImplementation	<code>hwcfg = coder.HardwareImplementation</code>

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 21-37.

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 that 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 21-36.

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`.

Specify Data Types Used in Generated Code

In this section...


“Specify Data Type Using the MATLAB Coder App” on page 21-39

“Specify Data Type at the Command Line” on page 21-39

MATLAB Coder can use built-in C data types or predefined types from `rtwtypes.h` in generated code. By default, when the generated code declares variables, it uses built-in C types.

You can explicitly specify the data types used in generated code in the project settings dialog box or at the command line.

Specify Data Type Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to **Source Code**, **Static Library**, **Dynamic Library**, or **Executable** (depending on your requirements).
- 3 Click **More Settings**.
- 4 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'`.

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 the project build settings, on the **Custom Code** tab, set the **Standard math library** parameter.
- In a code configuration object, set the `TargetLangStandard` parameter.

See Also


- “Standard Math Libraries” on page 21-40
- “Specify Build Configuration Parameters MATLAB Coder App” on page 21-33
- “Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 21-34

Share Build Configuration Settings

To share build configuration settings between multiple projects or between the project and command-line workflow, you can export settings to and import settings from a code generation configuration object.

Export Settings

To export the current project settings to a code generation configuration object stored in the base workspace:

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to **Source Code**, **Static Library**, **Dynamic Library**, or **Executable** (depending on your requirements).
- 3 Click **More Settings**.
- 4 Click **Import/Export Settings**.
- 5 In the **Variable name** field, specify a name for the configuration object.
- 6 Click **Export to Variable**.


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	coder.MexCodeConfig
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	coder.EmbeddedCodeConfig

You can then either import these settings into another project or use the configuration object 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 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to **Source Code**, **Static Library**, **Dynamic Library**, or **Executable** (depending on your requirements).
- 3 Click **More Settings**.
- 4 Click **Import/Export Settings**.
- 5 In the **Variable name** field, specify the name of the configuration object.
- 6 Click **Import from Variable**.

See Also

- “Configure Build Settings” on page 21-27
- `coder.config`
- “Convert MATLAB Coder Project to MATLAB Script” on page 21-44

Convert MATLAB Coder Project to MATLAB Script

After you define input types, 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.

You can convert a project using the MATLAB Coder app or the command-line interface.

Convert a Project Using the MATLAB Coder App

- 1 On the app toolbar, click , and then select **Convert to script**.
- 2 Specify the script name and click **Save**.

Convert a Project Using the Command-Line Interface

To convert a project to a script using the command-line interface, use the `-tocode` option of the `coder` command. The project file must be on the search path.

For example, to convert the project, `myproject.prj` to the script named `myscript.m` use this command:

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

The `coder` command overwrites a file that has the same name as the script. If you omit the `-script` option, the `coder` command writes the script to the Command Window.

For more information about the `-tocode` option, see `coder`.

Run the Script

- 1 Make sure that the entry-point functions that are arguments to `codegen` in the script are on the search path.
- 2 Run the script. For example:

```
myscript
```

The following variables appear in the base workspace.

Variable	For
<code>cfg</code>	Configuration object
<code>ARGS</code>	Types of input arguments, if the project has entry-point function inputs
<code>ARG</code>	Types of cell array elements, if the project has cell array inputs. A script can reuse <code>ARG</code> for different cell array elements
<code>GLOBALS</code>	Types and initial values of global variables, if the project has global variables

`cfg`, `ARGS`, `ARG`, 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:
C/C++ Dynamic Library	<code>coder.CodeConfig</code>
C/C++ Executable	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 21-42.

For a project that includes fixed-point conversion, project to script conversion 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” on page 15-110.

Primary Function Input Specification

In this section...

“Why You Must Specify Input Properties” on page 21-46

“Properties to Specify” on page 21-46

“Rules for Specifying Properties of Primary Inputs” on page 21-49

“Methods for Defining Properties of Primary Inputs” on page 21-50

“Define Input Properties by Example at the Command Line” on page 21-51

“Specify Constant Inputs at the Command Line” on page 21-53

“Specify Variable-Size Inputs at the Command Line” on page 21-54

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

For	Specify properties				
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	<p>Specify properties for each field according to its class</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"> • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify <code>numericity</code>, and <code>fimath</code>. 				
Other inputs	✓	✓	✓		

Default Property Values

MATLAB Coder assigns the following default values for properties of primary function inputs.

Property	Default
<code>class</code>	<code>double</code>
<code>size</code>	<code>scalar</code>
<code>complexity</code>	<code>real</code>
<code>numericity</code>	No default
<code>fimath</code>	MATLAB default <code>fimath</code> object

Specifying Default Values for Structure Fields

In most cases, when you do not 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” on page 21-77 and “Specifying Class and Size of Structure Array” on page 21-77.

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 that 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
“Specify Properties of Entry-Point Function Inputs Using the App” on page 18-3	<ul style="list-style-type: none"> • If you are working in a MATLAB Coder project, easy to use • Does not alter original MATLAB code • MATLAB Coder saves the definitions in the project file 	<ul style="list-style-type: none"> • Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Define Input Properties by Example at the Command Line” on page 21-51 Note: If you define input properties programmatically in the MATLAB file, you cannot use this method	<ul style="list-style-type: none"> • Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a few primary inputs 	<ul style="list-style-type: none"> • Must be specified at the command line every time you invoke <code>codegen</code> (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Define Input Properties Programmatically in the MATLAB File” on page 21-67	<ul style="list-style-type: none"> • Integrated with MATLAB code; no need to redefine properties each time you invoke MATLAB Coder • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures 	<ul style="list-style-type: none"> • Uses complex syntax • 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.

Define Input Properties by Example at the Command Line

- “Command-Line Option `-args`” on page 21-51
- “Rules for Using the `-args` Option” on page 21-51
- “Specifying Properties of Primary Inputs by Example at the Command Line” on page 21-52
- “Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line” on page 21-52

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” on page 21-73 for `codegen`.

For information about specifying cell array inputs, see “Specify Cell Array Inputs at the Command Line” on page 21-56.

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 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 do 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” on page 21-63.

Specifying a Structure as a Constant Input

Suppose that 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” on page 21-106.

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}
```

Specify Cell Array Inputs at the Command Line

To specify cell array inputs at the command line, use the same methods that you use for other types of inputs. You can:

- Provide an example cell array input to the `-args` option of the `codegen` command.
- Provide a `coder.CellType` object to the `-args` option of the `codegen` command. To create a `coder.CellType` object, use `coder.typeof`.
- Use `coder.Constant` to specify a constant cell array input.

For code generation, cell arrays are classified as homogeneous or heterogeneous. See “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2. When you provide an example cell array to `codegen` or `coder.typeof`, the function determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for `{1 [2 3]}` can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `coder.CellType` `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

If you have a test file, you can use `coder.getArgTypes` to determine input types. In the output cell array of types, for cell array inputs, `coder.getArgTypes` returns a `coder.CellType` object. If you want a different classification (homogeneous or heterogeneous), use the `makeHomogeneous` or `makeHeterogeneous` methods.

Specify Cell Array Inputs by Example

To specify a cell array input by example, provide an example cell array in the `-args` option of the `codegen` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

```
codegen myfunction -args {{1 2 3}} -report
```

The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.

- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

```
codegen myfunction -args {'a', 1} -report
```

The input argument is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.

Specify the Type of the Cell Array Input

To specify the type of a cell array input, use `coder.typeof` to create a `coder.CellType` object. Pass the `coder.CellType` object to the `-args` option of the `codegen` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

```
t = coder.typeof({1 2 3});
codegen myfunction -args {t} -report
```

The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.

- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

```
t = coder.typeof({'a', 1});
codegen myfunction -args {t}
```

The input argument is a 1x2 heterogeneous cell array whose first element is a 1x1 char and whose second element is a 1x1 double.

You can also use the advanced function `coder.newtype` to create a `coder.CellType` object.

Make a Homogeneous Copy of a Type

If `coder.typeof` returns a heterogeneous cell array type, but you want a homogeneous type, use the `makeHomogeneous` method to make a homogeneous copy of the type.

The following code creates a heterogeneous type.

```
t = coder.typeof({1 [2 3]})  
  
t =  
  
coder.CellType  
  1x2 heterogeneous cell  
    f0: 1x1 double  
    f1: 1x2 double
```

To make a homogeneous copy of the type, use:

```
t = makeHomogeneous(t)  
  
t =  
  
coder.CellType  
  1x2 homogeneous cell  
    base: 1x:2 double
```

Alternatively, use this notation:

```
t = makeHomogeneous(coder.typeof({1 [2 3]}))  
  
t =  
  
coder.CellType  
  1x2 homogeneous cell  
    base: 1x:2 double
```

The classification as homogeneous is permanent. You cannot later use the `makeHeterogeneous` method to make a heterogeneous copy of the type.

If the elements of a type have different classes, such as `char` and `double`, you cannot use `makeHomogeneous` to make a homogeneous copy of the type.

If you use `coder.cstructname` to specify a name for the structure type that represents a type in the generated code, you cannot create a homogeneous copy of the type.

Make a Heterogeneous Copy of a Type

If `coder.typeof` returns a homogeneous cell array type, but you want a heterogeneous type, use the `makeHeterogeneous` method to make a heterogeneous copy of the type.

The following code creates a homogeneous type.

```
t = coder.typeof({1 2 3})

t =

coder.CellType
  1x3 homogeneous cell
  base: 1x1 double
```

To make the type heterogeneous, use:

```
t = makeHeterogeneous(t)

t =

coder.CellType
  1x3 heterogeneous cell
  f0: 1x1 double
  f1: 1x1 double
  f2: 1x1 double
```

Alternatively, use this notation:

```
t = makeHeterogeneous(coder.typeof({1 2 3}))

t =

coder.CellType
  1x3 heterogeneous cell
  f0: 1x1 double
  f1: 1x1 double
  f2: 1x1 double
```

The classification as heterogeneous is permanent. You cannot later use the `makeHomogeneous` method to make a homogeneous copy of the type.

If a type is variable size, you cannot use `makeHeterogeneous` to make a heterogeneous copy of it.

Specify Variable-Size Cell Array Inputs

You can specify variable-size cell array inputs in the following ways:

- In the `coder.typeof` call.

For example, to specify a variable-size cell array whose first dimension is fixed and whose second dimension has an upper bound of 5:

```
t = coder.typeof({1}, [1 5], [0 1])  
  
t =  
  
coder.CellType  
  1x:5 homogeneous cell  
  base: 1x1 double
```

For elements with the same classes, but different sizes, you can use the `coder.typeof` `size` and `variable dimensions` arguments to create a variable-size homogeneous cell array type. For example, the following code does not use the `size` and `variable dimensions` arguments. This code creates a type for a heterogeneous cell array.

```
t = coder.typeof({1 [2 3]})  
  
t =  
  
coder.CellType  
  1x2 heterogeneous cell  
  f0: 1x1 double  
  f1: 1x2 double
```

The following code, that uses the `size` and `dimensions` arguments, creates a type for a variable-size homogeneous type cell array:

```
t = coder.typeof({1 [2 3]}, [1 5], [0 1])  
  
t =  
  
coder.CellType  
  1x:5 homogeneous cell  
  base: 1x:2 double
```

- Use `coder.resize`.

For example, to specify a variable-size cell array whose first dimension is fixed and whose second dimension has an upper bound of 5:

```
t = coder.typeof({1});
t = coder.resize(t, [1 5], [0,1])

t =

coder.CellType
  1x5 homogeneous cell
  base: 1x1 double
```

You cannot use `coder.resize` with a heterogeneous cell array type.

Specify Type Name for Heterogeneous Cell Array Inputs

A heterogeneous cell array is represented in the generated code as a structure. To specify the name of the structure type in the generated code, use `coder.cstructname`.

For example, to specify the name `myname` for the cell array type in the generated code:

```
t = coder.typeof({'a', 1})
t = coder.cstructname(t, 'myname')

t =

coder.CellType
  1x2 heterogeneous cell myname
  f0: 1x1 char
  f1: 1x1 double
```

If you use `coder.cstructname` with a homogeneous cell array type, `coder.cstructname` returns a heterogeneous copy of the type. However, it is a best practice to use the `makeHeterogeneous` method of the `coder.CellType` object to make a heterogeneous copy of a homogeneous cell array type. Then, you can use `coder.cstructname` with the heterogeneous copy of the type.

Specify Constant Cell Array Inputs

To specify that a cell array input is constant, use the `coder.Constant` function with the `-args` option of the `codegen` command. For example:

```
codegen myfunction -args {coder.Constant({'red', 1 'green', 2, 'blue', 3})} -report
```

The input is a 1x6 heterogeneous cell array. The sizes and classes of the elements are:

- 1x3 char
- 1x1 double
- 1x5 char
- 1x1 double
- 1x4 char
- 1x1 double

See Also

[coder.CellType](#) | [coder.getArgTypes](#) | [coder.newtype](#) | [coder.resize](#) | [coder.typeof](#)

Related Examples

- “Define Input Properties by Example at the Command Line” on page 21-51
- “Specify Constant Inputs at the Command Line” on page 21-53

More About

- “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2

Control Constant Inputs in MEX Function Signatures

In this section...

“Control MEX Function Signature Using the MATLAB Coder App” on page 21-63

“Control MEX Function Signature at the Command-Line Interface” on page 21-63


“Options for Controlling Constant Inputs in MEX Function Signatures” on page 21-64

“Call MEX Function with a Constant Input” on page 21-65

“See Also” on page 21-66

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 MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to MEX.
- 3 Click **More Settings**.
- 4 On the **All Settings** tab, set **Constant Inputs** to one of the menu options. See “Options for Controlling Constant Inputs in MEX Function Signatures” on page 21-64.

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 = coder.config('mex');
mexcfg.ConstantInputs = 'IgnoreValues';
```

For a description of the options, see “Options for Controlling Constant Inputs in MEX Function Signatures” on page 21-64

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"> • The MEX function signature includes the constant inputs. When you call the function, you must provide the constant inputs. • 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. • Allows you to use the same test file to run the original MATLAB algorithm and the MEX function. • Slows down execution of the MEX function. • This setting is the default.
Ignore input value	'IgnoreValues'	<ul style="list-style-type: none"> • The MEX function signature includes the constant inputs. When you call the function, you must provide the constant inputs. • The run-time values of the constant inputs can differ from the compile-time values.

Constant Inputs (Project)	ConstantInputs (Configuration Object)	Description
		<ul style="list-style-type: none"> Allows you to use the same test file to run the original MATLAB algorithm and the MEX function.
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” on page 21-53
- “Define Constant Input Parameters Using the App” on page 18-23

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.

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.

When specifying input properties using the `assert` function, use one of the following methods. Use the exact syntax that is provided; do not modify it.

- “Specify Any Class” on page 21-67
- “Specify `fi` Class” on page 21-68
- “Specify Structure Class” on page 21-68
- “Specify Cell Array Class” on page 21-69
- “Specify Fixed Size” on page 21-69
- “Specify Scalar Size” on page 21-69
- “Specify Upper Bounds for Variable-Size Inputs” on page 21-70
- “Specify Inputs with Fixed- and Variable-Size Dimensions” on page 21-70
- “Specify Size of Individual Dimensions” on page 21-70
- “Specify Real Input” on page 21-71
- “Specify Complex Input” on page 21-71
- “Specify `numericType` of Fixed-Point Input” on page 21-71
- “Specify `fimath` of Fixed-Point Input” on page 21-72
- “Specify Multiple Properties of Input” on page 21-72

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'));
```

...

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'));  
...
```

You must specify both the **fi** class and the **numericType**. See “Specify numericType of Fixed-Point Input” on page 21-71. You can also set the **fiMath** properties, see “Specify fiMath of Fixed-Point Input” on page 21-72. 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 that they appear in the structure definition.

Specify Cell Array Class

```
assert(iscell( param ))
assert(isa( param, 'cell' ))
```

Sets the input parameter *param* to the MATLAB class `cell` (cell array). For example, to set the class of input `C` to a `cell`, call:

```
...
assert(iscell(C));
...
```

or

```
...
assert(isa(C, 'cell'));
...
```

To specify the properties of cell array elements, see “Specifying Properties of Cell Arrays” on page 21-75.

Specify Fixed Size

```
assert ( all ( size ( param ) == [ dims ] ) )
```

Sets the input parameter *param* to the size that dimensions *dims* specifies. 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_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 nonnegative.

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 and all dimensions simultaneously. You can also specify individual dimensions 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));
...
```

Specifying the `numerictype` for a variable does not automatically specify that the variable is fixed point. You must specify both the `fi` class and the `numerictype`.

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.
- For a fixed-point input, you must specify both the `fi` class and the `numericType`. See “Specify `numericType` of Fixed-Point Input” on page 21-71. You can also set the `fimath` properties. See “Specify `fimath` of Fixed-Point Input” on page 21-72. 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 nonnegative.
- 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);
```

You must specify both the `fi` class and the `numericity`.

Specifying Properties of Cell Arrays

To specify the MATLAB class `cell` (cell array), use one of the following syntaxes:

```
assert(iscell(param))
assert(isa(param, 'cell'))
```

For example, to set the class of input `C` to `cell`, use:

```
...
assert(iscell(C));
...

or

...
assert(isa(C, 'cell'));
...

```

You can also specify the size of the cell array and the properties of the cell array elements. The number of elements that you specify determines whether the cell array is

homogeneous or heterogeneous. See “Homogeneous vs. Heterogeneous Cell Arrays” on page 9-2.

If you specify the properties of the first element only, the cell array is homogeneous. For example, the following code specifies that **C** is a 1x3 homogeneous cell array whose elements are 1x1 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1 3]));
assert(isa(C{1}, 'double'));
...
```

If you specify the properties of the first element only, but also assign a structure type name to the cell array, the cell array is heterogeneous. Each element has the properties of the first element. For example, the following code specifies that **C** is a 1x3 heterogeneous cell array. Each element is a 1x1 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1 3]));
assert(isa(C{1}, 'double'));
coder.cstructname(C, 'myname');
...
```

If you specify the properties of each element, the cell array is heterogeneous. For example, the following code specifies a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x3 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1 2]));
assert(isa(C{1}, 'char'));
assert(all(size(C{2}) == [1 3]));
assert(isa(C{2}, 'double'));
...
```

If you specify more than one element, you cannot specify that the cell array is variable size, even if all elements have the same properties. For example, the following code specifies a variable-size cell array. Because the code specifies the properties of the first and second elements, code generation fails.

```
...
assert(isa(C, 'cell'));
```

```
assert(all(size(C) <= [1 2]));
assert(isa(C{1}, 'double'));
assert(isa(C{2}, 'double'));
...
```

In the previous example, if you specify the first element only, you can specify that the cell array is variable-size. For example:

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1 2]));
assert(isa(C{1}, 'double'));
...
```

Specifying Class and Size of Scalar Structure

Suppose that you define `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

The following code specifies the properties of the function input `S` and its fields:

```
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 do not 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. At 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 that 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** by 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. At a minimum, you must specify the class of all fields.

Speed Up Compilation

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.

Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build Type** to **Static Library**, **Dynamic Library**, or **Executable**.
- 3 Select the **Generate code only** check box.

Alternatively, set **Build type** to **Source Code**. This build type is equivalent to **Static Library** with the **Generate code only** box selected.

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 Creation of the Code Generation Report

If you disable creation of the code generation report, you can speed up code generation, unless an error occurs. If an error occurs, the code generation software creates a report even if you disabled creation of the report.

To disable creation of the code generation report:

- In the MATLAB Coder app, in the project build settings, on the **Debugging** tab, clear the **Always create a code generation report** check box.
- At the command line, when you generate code, do not use the `-report` option. If you specify a code configuration object, make sure that the `GenerateReport` property is set to `false`.

By default, creation of the code generation report is disabled.

Related Examples

- “Enable Code Generation Reports” on page 22-27

More About

- “Configure Build Settings” on page 21-27
- “Code Generation Reports” on page 22-10

Paths and File Infrastructure Setup

In this section...

“Compile Path Search Order” on page 21-81

“Specify Folders to Search for Custom Code” on page 21-81

“Naming Conventions” on page 21-82


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 MATLAB for execution. See “Resolution of Function Calls for Code Generation” on page 14-2.

Specify 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 (Spaces can lead to code generation failures in certain operating system configurations)
- Tabs
- \, \$, #, *, ?
- Non-7-bit ASCII characters, such as Japanese characters

To specify additional folders	Do this
Using the MATLAB Coder app	<ol style="list-style-type: none"> 1 To open the Generate dialog box, on the Generate Code page, click the Generate arrow . 2 Click More Settings.

To specify additional folders	Do this
	3 On the Paths tab, in the Search paths field, either browse to add a folder to the search path or enter the full path. The search path must not contain spaces.
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 21-82
- “Reserved Keywords” on page 21-82
- “Conventions for Naming Generated files” on page 21-85

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 21-83
- “C++ Reserved Keywords” on page 21-83
- “Reserved Keywords for Code Generation” on page 21-83
- “MATLAB Coder Code Replacement Library Keywords” on page 21-84

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 21-83.

C Reserved Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
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.getTflReservedIdentifiers()
```

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 `crviewer`.

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
Linus Torvalds' Linux (32-bit)	.mexglx	.a	None

Platform	MEX File Extension	MATLAB Coder Library Extension	MATLAB Coder Executable Extension
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 Multiple Entry-Point Functions” on page 21-87

“Generate Code for More Than One Entry-Point Function Using the MATLAB Coder App” on page 21-87

“Generating Code for More Than One Entry-Point Function at the Command Line” on page 21-90

“How to Call an Entry-Point Function in a MEX Function” on page 21-91

“How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code” on page 21-92

Advantages of Generating Code for Multiple Entry-Point Functions

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.

Generate Code for More Than One Entry-Point Function Using the MATLAB Coder App

This example shows how to generate code for multiple entry-point functions using the MATLAB Coder app.

Create the Entry-Point Functions

- 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 local writable folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

Create the Test File

In the folder that contains `ep1.m` and `ep12.m`, create a MATLAB file, `ep_test.m`, that calls `ep1` and `ep2` with example inputs.

```
function [y, y1] = ep_test
y = ep1(single(2));
y1 = ep2(double(3), double(4));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

Specify Source Files

- 1 On the **Select Source Files** page, type or select the name of the entry-point function `ep1`.

The app creates a project with the default name `ep1.prj` in the current folder.

- 2 To add `ep2` to the list of entry-point functions, click **Add Entry-Point Function**. Type or select the name of the entry-point function `ep2`.
- 3 Click **Next** to go to the **Define Input Types** step. The app analyzes the functions for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify a test file that MATLAB Coder can use to automatically define types:

- 1 Enter or select the test file `ep_test.m`.

2 Click **Autodefine Input Types**.

The test file, `ep_test.m`, calls the entry-point functions `ep1` and `ep2` with the example input types. MATLAB Coder infers that for `ep1`, input `u` is `single(1x1)`. For `ep2`, `u` and `v` are `double(1x1)`.

3 Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

1 To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .

The app populates the test file field with `ep_test`, the test file that you used to define the input types.

2 Click **Check for Issues**.

The app generates a MEX function named `ep1_mex` for `ep1` and `ep2`. It runs the test file `ep_test` replacing calls to `ep1` and `ep2` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

3 Click **Next** to go to the **Generate Code** step.

Generate MEX Function

1 To open the **Generate** dialog box, click the **Generate** arrow .

2 Set **Build type** to MEX.

3 Verify that the **Output file name** is `ep1_mex`. By default, the app uses the name of the alphabetically first entry-point function.

4 Click **Generate**.

MATLAB Coder builds the project. It 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. It creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 21-82.

You can now test your MEX function in MATLAB. See “How to Call an Entry-Point Function in a MEX Function” on page 21-91.

If you generate a static library for `ep1` and `ep2`, MATLAB Coder builds the project and generates a C library, `ep1`, and supporting files in the default folder, `codegen/lib/ep1`.

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 22-10.

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 25-12.
- Generates the C/C++ executable using `codegen`.

See the example, “Call a C Static Library Function from C Code” on page 25-2.

Generate Code for Global Data

In this section...

“Workflow” on page 21-93

“Declare Global Variables” on page 21-93

“Define Global Data” on page 21-94

“Synchronizing Global Data with MATLAB” on page 21-95

“Define Constant Global Data” on page 21-99

“Limitations of Using Global Data” on page 21-102

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 21-94.

- 3 Generate code using the MATLAB Coder app 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 21-95.

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 in the MATLAB global workspace, in a MATLAB Coder project, or at the command line. If you do not initialize global data in the 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 21-93 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 Using the MATLAB Coder App

- 1 On the **Define Input Types** page, for **Does this code use global variables?**, select **Yes**.
- 2 Under **Global variables**, 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.

- 3 Click the field to the right of the global variables name. Specify the type and initial value of the global variable. See “Specify Global Variable Type and Initial Value Using the App” on page 18-26.

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 21-93, 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}}`. For cell arrays, you must use this format. See “Specify Global Cell Arrays at the Command Line” on page 21-103.

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 21-96.

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 21-101.

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 21-97.

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.
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 21-96.

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 Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to MEX.
- 3 Click **More Settings**.
- 4 On the **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 `foo` 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');
```

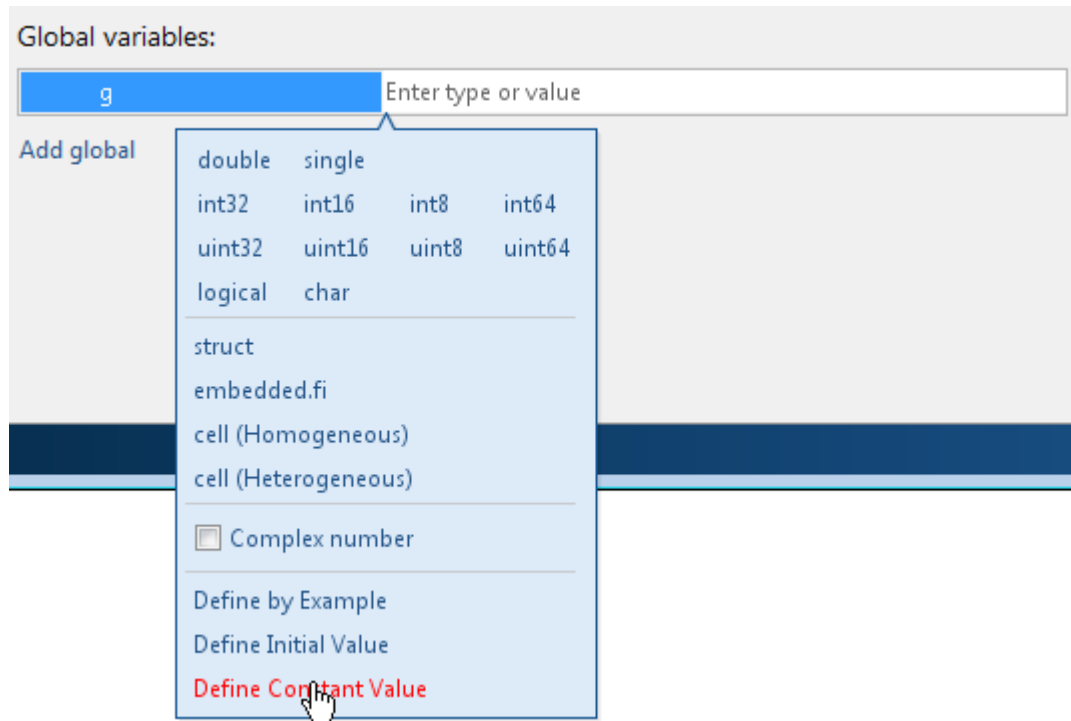
- Not synchronized, if the global data synchronization mode is **Disabled**. When synchronization is disabled, you cannot use the `-sync:on` option to control the synchronization for specific extrinsic calls.

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 MATLAB Coder App

- 1** On the **Define Input Types** page, for **Does this code use global variables?**, select **Yes**.
- 2** Under **Global Variables**, enter a name for the global variable.
- 3** Click the field to the right of the global variable name.
- 4** Select **Define Constant Value**.



- 5 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” on page 22-18.

Limitations of Using Global Data

You cannot use global data with the `coder.cstructname` function.

Specify Global Cell Arrays at the Command Line

To specify global cell array inputs, use the `-globals` option of the `codegen` command with this syntax:

```
codegen myfunction -globals {global_var, {type_object, initial_value}}
```

For example:

- To specify that the global variable `g` is a 1x3 cell array whose elements have class `double` and whose initial value is `{1 2 3}`, use:

```
codegen myfunction -globals {'g', {coder.typeof({1 1 1}), {1 2 3}}}
```

Alternatively, use:

```
t = coder.typeof({1 1 1});
codegen myfunction -globals {'g', {t, {1 2 3}}}
```

The global variable `g` is a 1x3 homogeneous cell array whose elements are 1x1 double.

To make `g` heterogeneous, use:

```
t = makeHeterogeneous(coder.typeof({1 1 1}));
codegen myfunction -globals {'g', {t, {1 2 3}}}
```

- To specify that `g` is a cell array whose first element has type `char`, whose second element has type `double`, and whose initial value is `{'a', 1}`, use:

```
codegen myfunction -globals {'g', {coder.typeof({'a', 1}), {'a', 1}}}
```

The global variable `g` is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.

- To specify that `g` is a cell array whose first element has type `double`, whose second element is a 1x2 double array, and whose initial value is `{1 [2 3]}`, use:

```
codegen myfunction -globals {'g', {coder.typeof({1 [2 3]}), {1 [2 3]}}}
```

Alternatively, use:

```
t = coder.typeof({1 [2 3]});
codegen myfunction -globals {'g', {t, {1 [2 3]}}}
```

The global variable `g` is a 1x2 heterogeneous cell array whose first element is 1x1 double and whose second element is 1x2 double.

Global variables that are cell arrays cannot have variable size.

See Also

`codegen` | `coder.typeof`

Related Examples

- “Generate Code for Global Data” on page 21-93

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” on page 10-3.
- 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” on page 10-24
- “Define Enumerated Data for Code Generation” on page 10-8
- “Specify an Enumerated Type Input Parameter by Example” on page 18-9
- “Specify an Enumerated Type Input Parameter by Type” on page 18-13

Generate Code for Variable-Size Data

In this section...

“Disable Support for Variable-Size Data” on page 21-106

“Control Dynamic Memory Allocation” on page 21-107

“Generating Code for MATLAB Functions with Variable-Size Data” on page 21-109

“Generate Code for a MATLAB Function That Expands a Vector in a Loop” on page 21-110


“Using Dynamic Memory Allocation for an "Atoms" Simulation” on page 21-116

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.

Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Memory** tab, select or clear **Enable variable-sizing**.

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 21-106), you also disable dynamic memory allocation. You can modify dynamic memory allocation settings from the project settings dialog box or the command line.

Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 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 max size at or above threshold	Dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to the Dynamic memory allocation threshold . Variable-size arrays whose size is less

Setting	Action
	than this threshold are allocated on the stack.

- 4 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.

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 Dynamic memory allocation threshold 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 21-110

- 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” on page 7-22.

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” on page 7-6 and “Diagnosing and Fixing Size Mismatch Errors” on page 7-22.
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 21-107.

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 `myunique_tol`” on page 21-111
- “Step 1: Add Compilation Directive for Code Generation” on page 21-111
- “Step 2: Address Issues Detected by the Code Analyzer” on page 21-111
- “Step 3: Generate MEX Code” on page 21-112
- “Step 4: Generate C Code” on page 21-113
- “Step 5: Specify an Upper Bound for the Output Vector” on page 21-114
- “Step 6: Change the Dynamic Memory Allocation Threshold” on page 21-115

About the MATLAB Function `myuniquetol`

This example uses the function `myuniquetol`. 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 = myuniquetol(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 = myuniquetol(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, you expect vector `B` to expand in size because it adds values from vector `A`. Therefore, you can ignore this warning.

Step 3: Generate MEX Code

It is a best practice to generate MEX code before you generate C/C++ code. Generating MEX code can identify code generation issues that are harder to detect at run time.

- 1 Generate a MEX function for `myunique1`:

```
codegen -report myunique1 -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 `myunique1`:

- 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” on page 21-54.

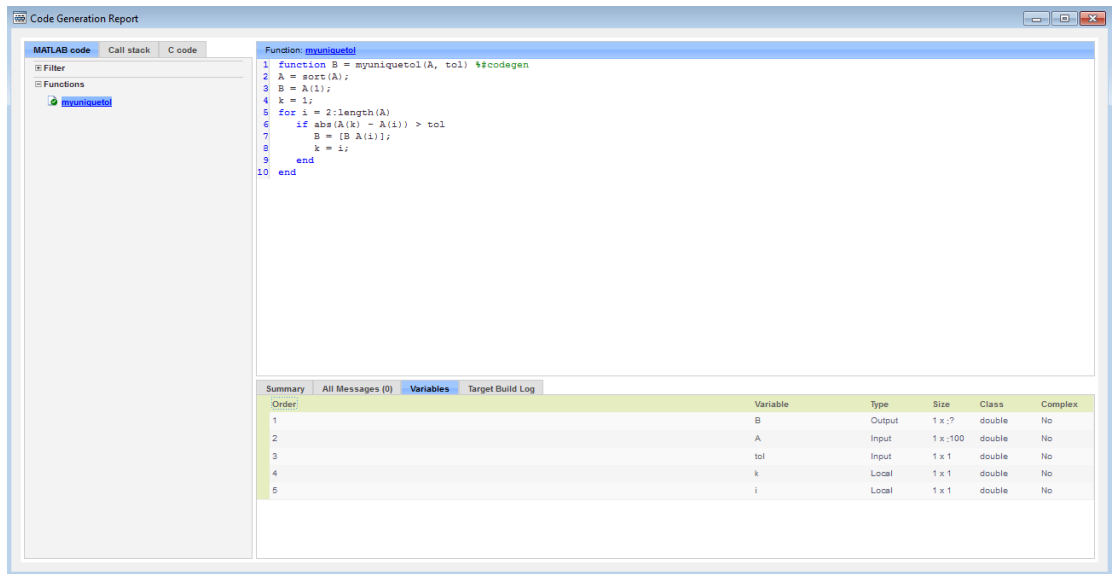
- The second argument, `coder.typeof(0)`, defines input `tol` as a real double scalar.

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.

Code generation is successful. `codegen` does not detect issues. In the current folder, `codegen` generates a MEX function for `myunique1` and provides a link to the code generation report.

- 2 Click the **View report** link.
- 3 In the code generation report, select the **Variables** tab.



The size of A is 1x:100 because you specified that A is variable size with an upper bound of 100. The size of variable B is 1x:?, indicating that it is variable size with no upper bounds.

Step 4: 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 myunique_tol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

`codegen` generates a static library in the default location, `codegen\lib\myunique_tol` 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 `myuniquetol.h`.

The function declaration is:

```
extern void myuniquetol(const double A_data[], 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:

- `double A_data[]`: the definition of `A`.
- `int A_size[2]`: the actual size of the input.

The code generation software determines that `B` is variable size with unknown upper bounds. It 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” on page 7-17.

Step 5: Specify an Upper Bound for the Output Vector

You specified that the input `A` is variable size with an upper bound of 100. Therefore, you know that the output `B` cannot be larger than 100 elements.

- Use `coder. varsize` to indicate that `B` is variable size with an upper bound of 100.

```
function B = myuniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 100], [0 1]);
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.

```
codegen -config cfg -report myuniquetol -args {coder.typeof(0,[1 100],1),coder.typeof(0)}
```

The function declaration is:

```
extern void myuniquetol(const double A_data[], const int A_size[2],...
    double tol, double B_data[], int B_size[2]);
```

The code generation software statically allocates the memory for B. It stores the size of B in `int B_size[2]`.

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. This step specifies that the second dimension of A has an upper bound of 10000.

- 1 Change the upper bound of B to match the upper bound of A.

```
function B = myuniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 10000], [0 1]);
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
```

- 2 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 myuniquetol -args {coder.typeof(0,[1 10000],1),coder.typeof(0)}
```

- 3 View the generated code in the report. Because the maximum size of A and B now exceed the dynamic memory allocation threshold, `codegen` allocates A and B dynamically on the heap. In the generated code, A and B have type `emxArray_real_T`.

```
extern void myuniquetol(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 = repmat(atom,1,0);  
atoms = run_atoms_mex(atoms,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
```

```
.
..
buildInfo.mat
codeInfo.mat
examples
interface
rtGetInf.c
rtGetInf.h
rtGetInf.obj
rtGetNaN.c
rtGetNaN.h
rtGetNaN.obj
rt_nonfinite.c
rt_nonfinite.h
rt_nonfinite.obj
rtw_proj.tmw
rtwtypes.h
run_atoms.c
run_atoms.h
run_atoms.lib
run_atoms.obj
run_atoms_emxAPI.c
run_atoms_emxAPI.h
run_atoms_emxAPI.obj
run_atoms_emxutil.c
run_atoms_emxutil.h
run_atoms_emxutil.obj
run_atoms_initialize.c
run_atoms_initialize.h
run_atoms_initialize.obj
run_atoms_ref.rsp
run_atoms_rtw.bat
run_atoms_rtw.mk
run_atoms_terminate.c
run_atoms_terminate.h
run_atoms_terminate.obj
run_atoms_types.h
```

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

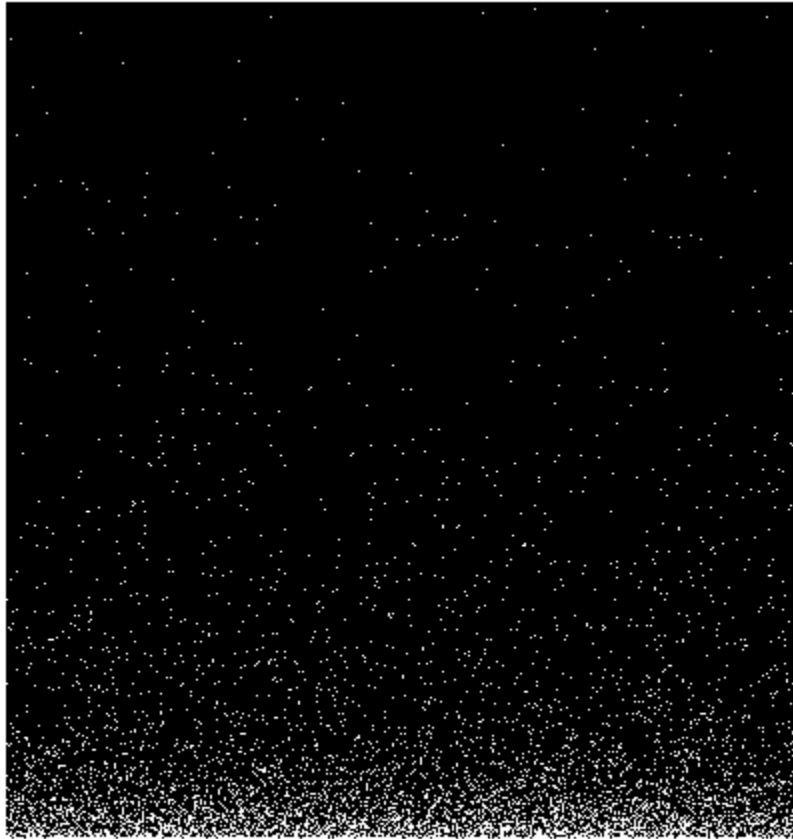
Running the executable produced `'atoms_state.mat'`. Now, recreate the structure array from the saved matrix

```
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 21-125

“How to Select the File Partitioning Method” on page 21-125

“Partitioning Generated Files with One C/C++ File Per MATLAB File” on page 21-126

“Generated Files and Locations” on page 21-131

“File Partitioning and Inlining” on page 21-133


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 21-133).

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 21-125. This option facilitates integrating your code with existing embedded software.

How to Select the File Partitioning Method

Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 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**.

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 app, to generate a C static library for `foo.m`:

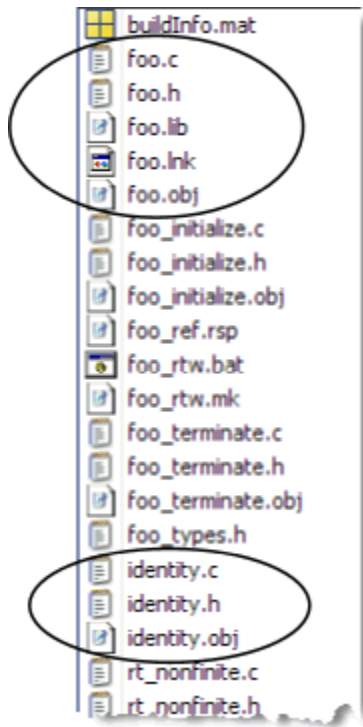
- 1 Define the inputs `u` and `v`. For more information, see “Specify Properties of Entry-Point Function Inputs Using the App” on page 18-3.
- 2 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 3 Set the **Build type** to **Static Library**

- 4 Click **More Settings**.
- 5 On the **All Settings** tab, under **Function Inlining**, set the **Inline threshold** parameter to 0
- 6 Click **Close**
- 7 To generate the library, click **Generate**.

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:

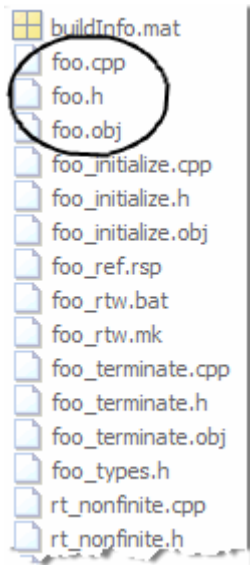
- 1 Select C++ as your target language:

```
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`:

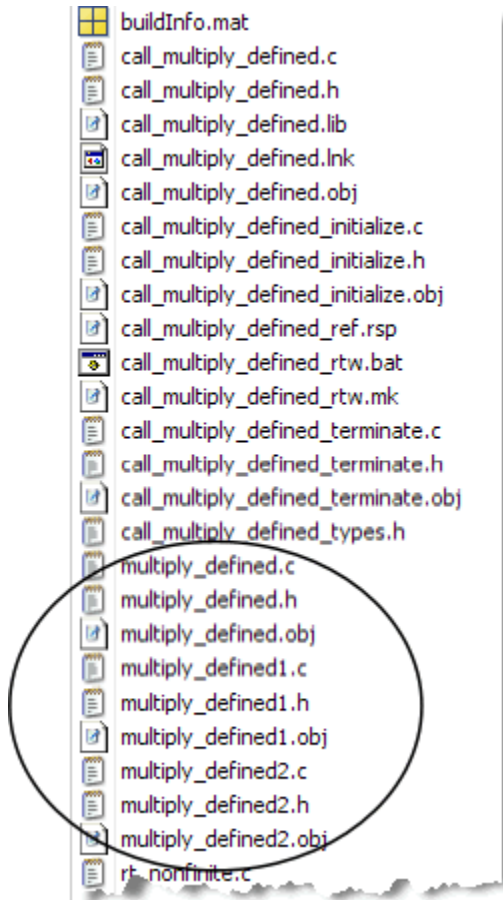
```
 %#codegen
function [y1,y2,y3] = call_multiply_defined

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	<code>codegen/mex/function_name</code>
HTML reports	<code>codegen/mex/function_name/html</code>

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	<code>codegen/lib/function_name</code>
HTML reports	<code>codegen/lib/function_name/html</code>

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	<code>codegen/dll/function_name</code>
HTML reports	<code>codegen/dll/function_name/html</code>

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	<code>codegen/exe/function_name</code>

Type of Files	Location
HTML reports	codegen/exe/function_name/html

Changing Names and Locations of Generated Files

Using the MATLAB Coder App

To change	Action
The output file name	<ol style="list-style-type: none"> 1 To open the Generate dialog box, on the Generate Code page, click the Generate arrow . 2 In the Output file name field, enter the file name.
The output file location	<ol style="list-style-type: none"> 1 To open the Generate dialog box, on the Generate Code page, click the Generate arrow . 2 Click More Settings. 3 On the Paths tab, set Build folder to Specified folder. 4 For the Build folder name field, either browse to the output file location or enter the full path. The output file location must not contain: <ul style="list-style-type: none"> • Spaces (Spaces can lead to code generation failures in certain operating system configurations). • Tabs • \, \$, #, *, ? • Non-7-bit ASCII characters, such as Japanese characters.

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.
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 21-126.
Generate one C/C++ file for each MATLAB file and enable inlining	<p>Places inlined functions in the same C/C++ file as the function into which they are inlined.</p> <p>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.</p>

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.


If You Generate	Generated C/C++ Code	Advantages	Disadvantages
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 21-125.
- Explicitly disable inlining, either globally or for individual MATLAB functions.

How to Disable Inlining Globally Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **All Settings** tab, under **Function Inlining** set the **Inline threshold** to 0.

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 21-135.


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 Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **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 21-139

“Customize Build Using Post-Code-Generation Command” on page 21-139

“Build Information Object” on page 21-140

“Build Information Methods” on page 21-140

“Write Post-Code-Generation Command” on page 21-174

“Use Post-Code-Generation Command to Customize Build” on page 21-175

“Write and Use Post-Code-Generation Command at the Command Line” on page 21-176

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” on page 21-140.
- 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” on page 21-174. 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” on page 21-175.

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” on page 21-140 to access this information in the build information object. “Write Post-Code-Generation Command” on page 21-174 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 groups 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.
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 <i>paths</i> .

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 groups 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.

If You Specify <i>paths</i> or <i>groups</i> as a...	The Function...
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.

To	Specify <i>groups</i> as a.
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
-
- 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 groups 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 as a The Function

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 <code>-D</code> 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> .

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
true	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
<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 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: • <code>mlrFiles.zip</code> — files in your <code>matlabroot</code> folder tree	'nestedZipFiles'	true (default)

To	Specify Property	With Value
<ul style="list-style-type: none"> • <code>sDirFiles.zip</code> — files in and under your build folder • <code>otherFiles.zip</code> — required files not in the <i>matlabroot</i> or <i>start</i> folder trees <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.
 - The function does not package example main source and header files that you generate using the default configuration settings. To package the example main files, configure code generation to generate and compile the example main function, generate your code, and then package the build files.
 - 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 Code for Other Development Environments” on page 25-44

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[®]) 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 21-176

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 MATLAB Coder App.

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Custom Code** tab, set the **Post-code-generation command** parameter.

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 21-175 and “Use Post-Code-Generation Command in the MATLAB Coder App.” on page 21-175.

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 21-175 and “Use Post-Code-Generation Command in the MATLAB Coder App.” on page 21-175.

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
```


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, in 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.

Verify Generated C/C++ Code

- “Generation of Traceable Code” on page 22-2
- “Code Generation Reports” on page 22-10
- “Enable Code Generation Reports” on page 22-27
- “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 22-28
- “Generate Standalone Code That Detects and Reports Run-Time Errors” on page 22-30

Generation of Traceable Code

In this section...

“Code Traceability” on page 22-2

“Generate Traceable Code” on page 22-2

“Format of Traceability Tags” on page 22-4

“Location of Comments in Generated Code” on page 22-5

“Traceability Limitations” on page 22-9

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 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 22-4.

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.

Use the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .

- 2 Click **More Settings**.
- 3 On the **Code Appearance** tab, under **Comments**, select the **MATLAB source code as comments** check box.

Use the Command-Line Interface

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 At 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 by 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 by 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 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 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 can still appear in the generated code.
 - If a complete function or code block is eliminated, comments can be eliminated from the generated code.
 - For certain optimizations, the comments can 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.

Code Generation Reports

In this section...

“About Code Generation Reports” on page 22-10

“View Your MATLAB Code in a Report” on page 22-13

“Viewing Call Stack Information” on page 22-14

“View Generated C and C++ Code in a Report” on page 22-16

“View the Build Summary Information” on page 22-17

“View Errors and Warnings in a Report” on page 22-17

“Viewing Variables in Your MATLAB Code” on page 22-18

“View Target Build Information” on page 22-24

“Keyboard Shortcuts for the Code Generation Report” on page 22-25

“Report Limitations” on page 22-26

About Code Generation Reports

At code generation time, MATLAB Coder produces reports to help you debug your MATLAB code. These reports also help you to verify that your MATLAB code is suitable for code generation.

Report Generation

If MATLAB Coder detects errors or warnings, the software produces a code generation report. Even if MATLAB Coder does not detect errors or warnings, you can use an option to request reports.

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 “Configure Build Settings” on page 21-27.

Opening Code Generation Reports

Opening Code Generation Reports with the MATLAB Coder App

After code generation, on the **Generate Code** page, the **Build Log** tab 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, this report provides links to your MATLAB code and generated C/C++ files. It also provides 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

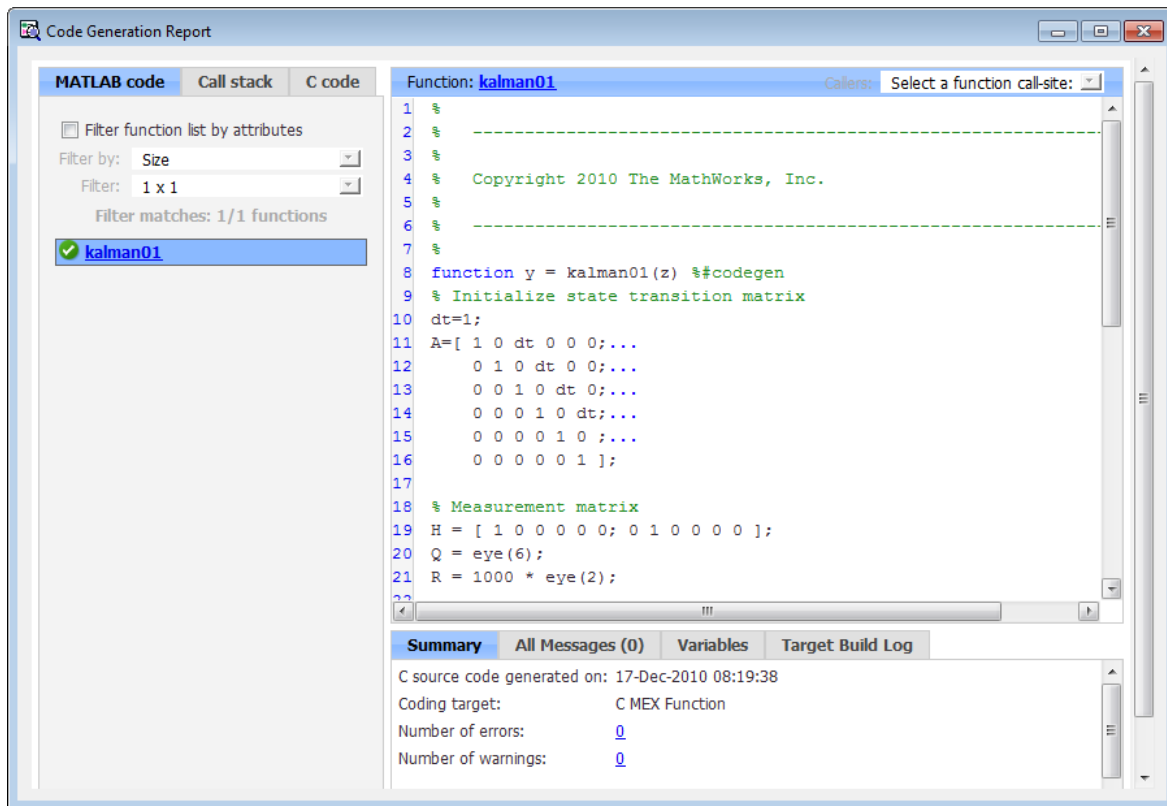
If you specify the `-launchreport` option, the code generation report opens automatically.

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 **View Error Report** link.

Description of Code Generation Reports

When you generate code for MATLAB files from a MATLAB Coder project, or at the command line by using the `codegen -report` option, MATLAB Coder generates a report. The following example report 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

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.

The **MATLAB code** tab provides:

- A list of the MATLAB functions and classes that you 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

In the function list on the **MATLAB code** tab, the code generation report annotates the local function with the name of the parent function.

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:

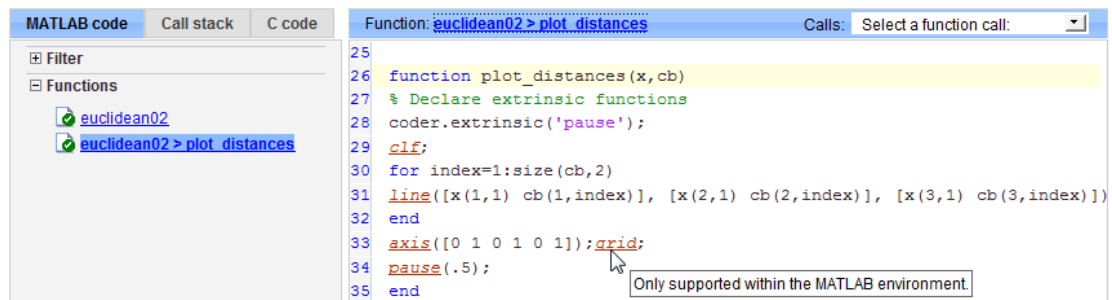
```
function y = fcn(u) %#codegen
% Specializations
y = y + subfcn(single(u));
y = y + subfcn(double(u));
```

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 more than one function calls a function, this list provides details of each call-site. Otherwise, the list is disabled.

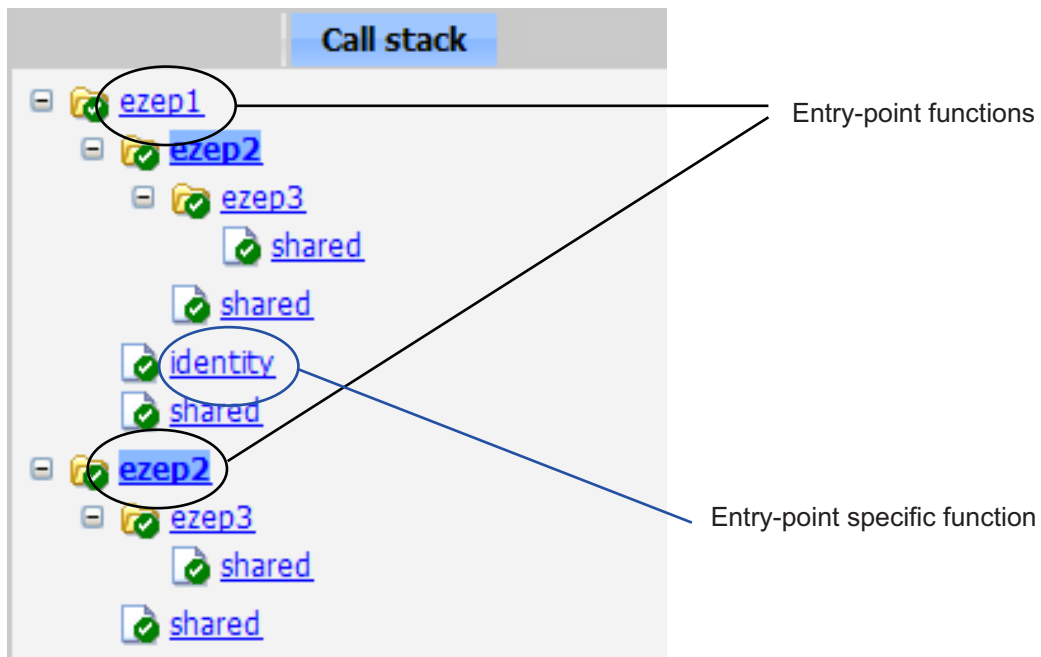
Viewing Call Stack Information on the Call stack Tab

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.

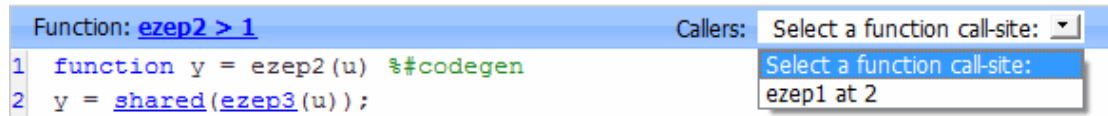


Viewing Call Sites in the Callers List

If more than one function or method calls a 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`.



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.

When you generate an example main function, by default, the code generation report does not include the generated example main files. If you configure code generation to generate and compile an example main function, and then you generate code, example main files appear in the code generation report. A list of source and header files for the example main function appears in the **Example Source Files** pane of the **C code** tab.

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 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 22-2.

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 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/.

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 as either:



Error



Warning

To locate the incorrect line of code for an error or warning, click the message in the list. The code generation report highlights errors in the list. It highlights MATLAB code in red and 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 the 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 a constant

argument includes the value. The information about constant arguments helps you to understand generated function signatures. It also helps you to see when code generation created function specializations for 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”.

The report displays only 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-size arrays, see “Variable-Size Data Definition for Code Generation” on page 7-3.

Viewing Renamed Variables in the Variables Tab

If your MATLAB function reuses a variable with a 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” on page 5-11. The report numbers the renamed variables in the list on the **Variables** tab. When you place your cursor 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

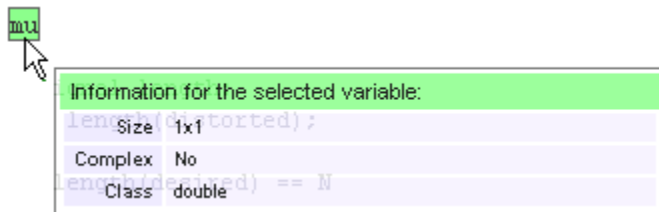
```

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	

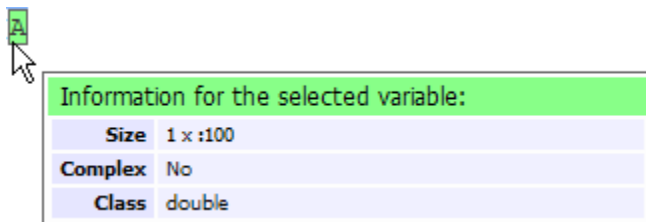
Viewing Information About Variables and Expressions in Your MATLAB Function Code

To view information about a particular variable or expression in your MATLAB function code, on the MATLAB code pane, place your cursor 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



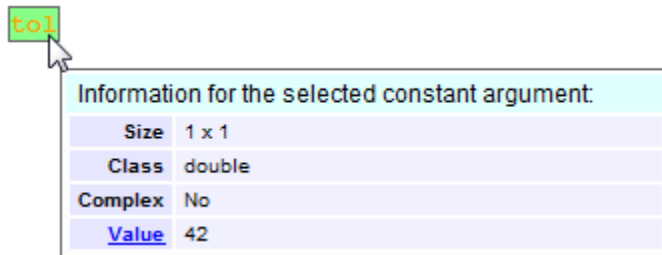
For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon `:`. Here the array `A` is variable-size with a maximum computed size of `1 x 100`.



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.



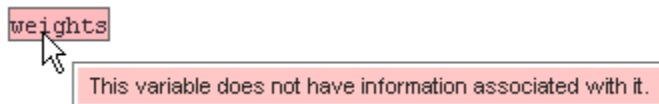
If you export the value as a variable to the base workspace, you can use the Workspace browser to view detailed information about the variable.

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**.

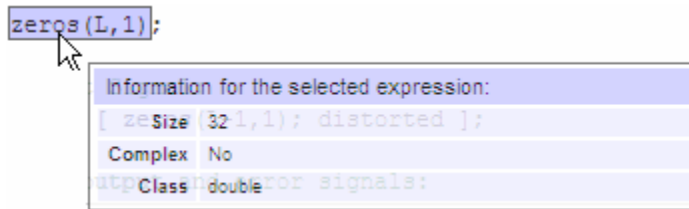
The variable and its value appear in the Workspace browser.

Pink, when the variable has no data type information



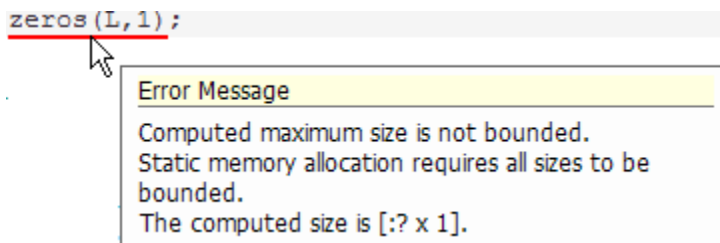
Purple, information about expressions

You can also view information about expressions in your MATLAB code. On the MATLAB code pane, place your cursor 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 to the **Target Build Log** tab so that you can view the build log.

Summary	All Messages (12)	Variables	Target Build Log
Build Parameters			
Build directory	C:\Work\emcpri\mexfcn\warn		
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\Aslrtw\~\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 select a pane, use the **Tab** key to advance through data in that pane.

To select	Use
MATLAB code Tab	Ctrl+m
Call stack Tab	Ctrl+k
C code Tab	Ctrl+c
Code Pane	Ctrl+w
Summary Tab	Ctrl+s
All Messages Tab	Ctrl+a
Variables Tab	Ctrl+v
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.

- The report does not provide information about all structure fields in the `struct()` constructor.
- 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 the 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 cursor 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];
```


Enable Code Generation Reports

In this section...

“Enable Code Generation Reports with the MATLAB Coder App” on page 22-27

“Enable Code Generation Reports at the Command Line” on page 22-27

Enable Code Generation Reports with the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Debugging** tab, select the **Always create a code generation report** check box.
- 4 If you want the app to open the report, select the **Automatically launch a report if one is generated** check box.

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.

Run-Time Error Detection and Reporting in Standalone C/C++ Code

You can generate standalone libraries and executables that detect and report run-time errors, such as out-of-bounds array indexing. If the generated code detects an error, it reports the error and terminates the program.

During development, before you generate C/C++ code, it is a best practice to test the generated code by running the MEX version of your algorithm. However, some errors occur only on the target hardware. To detect these errors, generate the standalone C/C++ code with run-time error detection enabled. Run-time error detection can affect the performance of the generated code. If performance is a consideration for your application, do not generate production code with run-time error detection enabled.

By default, run-time error detection is disabled for standalone libraries and executables. To enable run-time error detection and reporting for standalone libraries and executables:

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

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

- In the MATLAB Coder app, in the project settings dialog box, on the **Debugging** pane, select the **Generate run-time error checks** check box.

Run-time error detection and reporting in standalone code has these requirements and limitations:

- The error reporting software uses `fprintf` to write error messages to `stderr`. It uses `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. The `abort` function abruptly terminates the program. If your system supports signals, you can catch the abort signal (`SIGABRT`) so that you can control the program termination.
- Error messages are in English only.
- Some error checks require double-precision support. Therefore, the hardware on which the generated code runs must support double-precision operations.
- If the program terminates, the error detection and reporting software does not display the run-time stack. To inspect the stack, attach a debugger. Additionally, the

error detection and reporting software does not release resources, such as allocated memory.

- If the program terminates, the error detection and reporting software does not release resources, such as allocated memory.
- In standalone code, the function `error` does not report an error and does not terminate execution.
- In standalone code, if called with more than 1 argument, the function `assert` does not report an error and does not terminate execution. If called with a single argument, for example, `assert(cond)`, if `cond` is not a constant `true` value, reports an error and terminates execution.

Related Examples

- “Generate Standalone Code That Detects and Reports Run-Time Errors” on page 22-30

More About

- “Why Test MEX Functions in MATLAB?” on page 20-4

Generate Standalone Code That Detects and Reports Run-Time Errors

This example shows how to generate C/C++ libraries or executables that detect and report run-time errors such as out-of-bounds array indexing. If the generated code detects an error, it reports a message and terminates the program. You can detect and fix errors that occur only on the target hardware.

Write the function `getelement` that indexes into one structure field using the value of the other structure field.

```
function y = getelement(S)
y = S.A(S.u);
end
```

Create a code configuration object for a standalone library or executable. For example, create a code configuration object for a static library. Enable the code generation report.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
```

Enable generation of run-time error detection and reporting.

```
cfg.RuntimeChecks = true;
```

Define an example input that you can use to specify the properties of the input argument.

```
S.A = ones(2,2);
S.u = 0;
```

Generate code.

```
codegen -config cfg getelement -args {S}
```

To open the code generation report, click the **View report** link.

When the report opens, you see the generated C code. You can see the code that checks for an error and calls a function to report the error. For example, if the code detects an out-of-bounds array indexing error, it calls `rtDynamicBoundsError` to report the error and terminate the program.

```
/* Include Files */
#include "rt_nonfinite.h"
#include "getelement.h"
```



```

#include "getelement_rtwutil.h"
#include <stdio.h>
#include <stdlib.h>

/* Variable Definitions */
static rtBoundsCheckInfo emlrtBCI = { 1, 4, 2, 5, "S.A", "getelement",
    "C:\\\\coder\\\\runtime checks\\\\getelement.m", 0 };

static rtDoubleCheckInfo emlrtDCI = { 2, 5, "getelement",
    "C:\\\\coder\\\\runtime checks\\\\getelement.m", 1 };

/* Function Definitions */

/*
 * Arguments      : const struct0_T *S
 * Return Type    : double
 */
double getelement(const struct0_T *S)
{
    double d0;
    int i0;
    d0 = S->u;
    if (d0 != (int)floor(d0)) {
        rtIntegerError(d0, &emlrtDCI);
    }

    i0 = (int)d0;
    if (!(i0 >= 1) && (i0 <= 4)) {
        rtDynamicBoundsError(i0, 1, 4, &emlrtBCI);
    }

    return S->A[i0 - 1];
}

```

The error reporting software uses `fprintf` to write error messages to `stderr`. It uses `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. The `abort` function abruptly terminates the program. If your system supports signals, you can catch the abort signal (`SIGABRT`) so that you can control the program termination.

More About

- “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 22-28

Code Replacement for MATLAB Code

- “What Is Code Replacement?” on page 23-2
- “Code Replacement Libraries” on page 23-4
- “Code Replacement Terminology” on page 23-6
- “Code Replacement Limitations” on page 23-9
- “Replace Code Generated from MATLAB Code” on page 23-10
- “Choose a Code Replacement Library” on page 23-12

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—GNU¹ 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[®] 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” on page 23-10
- “Choose a Code Replacement Library” on page 23-12

More About

- “Code Replacement Libraries” on page 23-4
- “Code Replacement Terminology” on page 23-6
- “Code Replacement Limitations” on page 23-9

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.

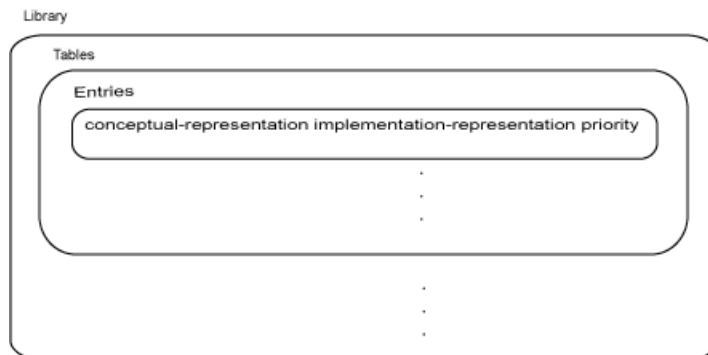


Table Entry Component	Description
Conceptual representation	<p>Identifies the table entry and contains match criteria for the code generator. Consists of:</p> <ul style="list-style-type: none"> • Function name or a key. The function name identifies most functions. For operators and some functions, a string called a key identifies a function or operator. For example, function name 'COS' and operator key 'RTW_OP_ADD'. • Conceptual arguments that observe code generator naming ('y1', 'u1', 'u2', ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.

Table Entry Component	Description
Implementation representation	Specifies replacement code. Consists of: <ul style="list-style-type: none"> • Function name. For example, 'cos_dbl' or 'u8_add_u8_u8'. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.
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” on page 23-10
- “Choose a Code Replacement Library” on page 23-12

More About

- “What Is Code Replacement?” on page 23-2
- “Code Replacement Terminology” on page 23-6

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"> • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
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"> • Function name (for example, 'cos_db1' or 'u8_add_u8_u8') • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
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.

Term	Definition
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?” on page 23-2
- “Code Replacement Libraries” on page 23-4

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” on page 23-10

More About

- “Code Replacement Libraries” on page 23-4

Replace Code Generated from MATLAB Code

This example shows how to replace generated code using a code replacement library. Code replacement is a technique for changing the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

- 1 Make sure that you have installed required software. Required software is:
 - MATLAB
 - MATLAB Coder
 - C compiler

Some code replacement libraries available in your development environment require Embedded Coder.

For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.

- 2 Identify an existing MATLAB function 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. Do one of the following:
 - In a project, on the **Custom Code** tab, set the **Code replacement library** parameter.
 - In a code configuration object, set the `CodeReplacementLibrary` parameter.
- 2 Configure the code generator to produce only code. Before you build an executable, verify your code replacements. Do one of the following:
 - In a project, in the **Generate** dialog box, select the **Generate code only** check box.

- 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 helps you verify code replacements. For more information, see “Verify Code Replacements” in the Embedded Coder documentation.

Generate Replacement Code

Generate C/C++ code from the MATLAB code. If you configured the code generator to produce a report, generate a code generation report. For example, in the MATLAB Coder app, on the **Generate Code** page, click **Generate**. 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 can sometimes 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” on page 23-12
- “Configure Build Settings” on page 21-27

More About

- “What Is Code Replacement?” on page 23-2
- “Code Replacement Libraries” on page 23-4
- “Code Replacement Terminology” on page 23-6
- “Code Replacement Limitations” on page 23-9

External Websites

- Supported Compilers

Choose a Code Replacement Library

In this section...

“About Choosing a Code Replacement Library” on page 23-12

“Explore Available Code Replacement Libraries” on page 23-12

“Explore Code Replacement Library Contents” on page 23-12

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?” on page 23-2.
 - See “Explore Available Code Replacement Libraries” on page 23-12.
- 2 Explore the contents of the library. See “Explore Code Replacement Library Contents” on page 23-12.

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 **Custom Code** 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 `crviewer`.

```
>> crviewer
```

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:

```
>> crviewer('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” on page 23-10

More About

- “Code Replacement Libraries” on page 23-4
- “Code Replacement Terminology” on page 23-6
- “Code Replacement Limitations” on page 23-9

Custom Toolchain Registration

- “Custom Toolchain Registration” on page 24-2
- “About `coder.make.ToolchainInfo`” on page 24-6
- “Create and Edit Toolchain Definition File” on page 24-8
- “Toolchain Definition File with Commentary” on page 24-10
- “Create and Validate `ToolchainInfo` Object” on page 24-16
- “Register the Custom Toolchain” on page 24-17
- “Use the Custom Toolchain” on page 24-19
- “Troubleshooting Custom Toolchain Validation” on page 24-20
- “Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers” on page 24-24

Custom Toolchain Registration

In this section...
“What Is a Custom Toolchain?” on page 24-2
“What Is a Factory Toolchain?” on page 24-2
“What is a Toolchain Definition?” on page 24-3
“Key Terms” on page 24-4
“Typical Workflow” on page 24-4

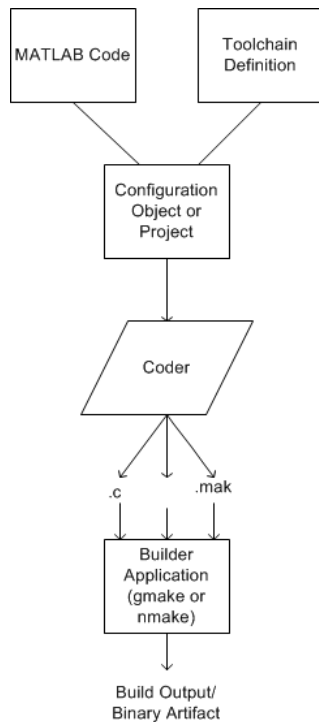
What Is a Custom Toolchain?

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” on page 24-8

- a** Create a toolchain definition file that returns a `coder.make.ToolchainInfo` object.
 - b** Update the file with information about the custom toolchain.
 - 2** “Create and Validate ToolchainInfo Object” on page 24-16
 - a** Use the toolchain definition file to create a `ToolchainInfo` object in the MATLAB workspace.
 - b** Validate the `ToolchainInfo` object.
 - c** Fix validation issues by updating the toolchain definition file, and creating/validating the updated `ToolchainInfo` object.
 - d** Create a valid `ToolchainInfo` object and save it to a MAT-file.
 - 3** “Register the Custom Toolchain” on page 24-17
 - a** Create an `rtwTargetInfo.m` file and update it with information about the MAT-file.
 - b** Register the custom toolchain in MATLAB Coder software using the `rtwTargetInfo.m` file.
 - 4** “Use the Custom Toolchain” on page 24-19
 - a** Configure MATLAB Coder software to use the custom toolchain.
 - b** Build and run an executable using the custom toolchain.

This workflow requires an iterative approach, with multiple cycles to arrive at a finished version of the custom `ToolchainInfo` object. You will need access to detailed information about the custom toolchain.

For a tutorial example of this workflow, see “Adding a Custom Toolchain”.

For more information about the `ToolchainInfo` object, see “About `coder.make.ToolchainInfo`” on page 24-6.

About `coder.make.ToolchainInfo`

The following properties in `coder.make.ToolchainInfo` represent your custom toolchain:

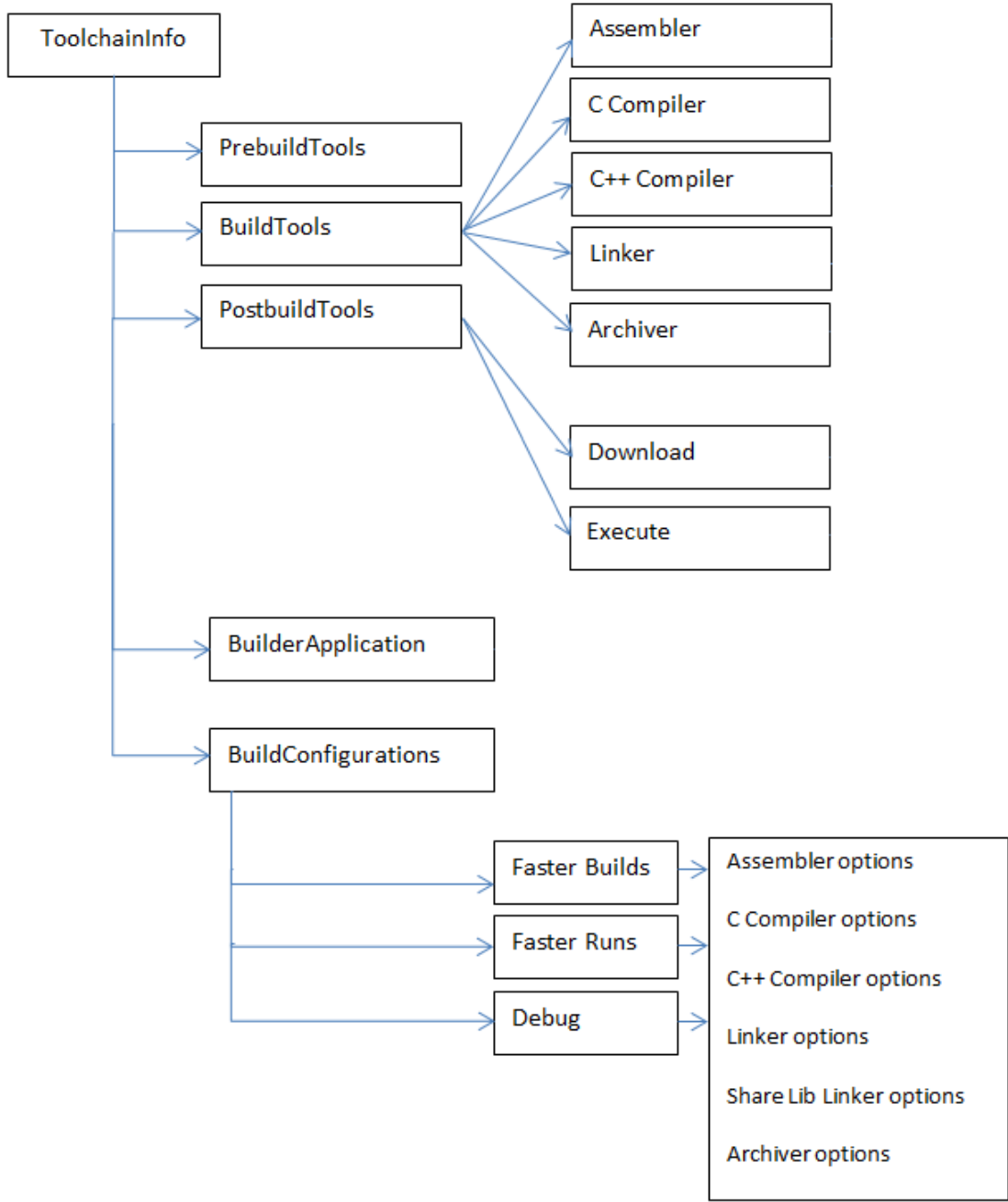
- `coder.make.ToolchainInfo.PrebuildTools` – Tools used before compiling the source files into object files.
- `coder.make.ToolchainInfo.BuildTools` – Tools used for compiling source files and linking/archiving them to form a binary.
- `coder.make.ToolchainInfo.PostbuildTools` – Tools used after the 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 24-4.

- 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 24-10.

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 24-16

Toolchain Definition File with Commentary

In this section...

“Steps Involved in Writing a Toolchain Definition File” on page 24-10
 “Write a Function That Creates a ToolchainInfo Object” on page 24-10
 “Setup” on page 24-11
 “Macros” on page 24-11
 “C Compiler” on page 24-12
 “C++ Compiler” on page 24-12
 “Linker” on page 24-13
 “Archiver” on page 24-13
 “Builder” on page 24-14
 “Build Configurations” on page 24-14

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 24-4.

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 24-8.

- 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 24-17.

For more information, see “Troubleshooting Custom Toolchain Validation” on page 24-20.

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 24-16.

- 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 24-19.

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 24-17.

- 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 24-2.

Troubleshooting Custom Toolchain Validation

In this section...

“Build Tool Command Path Incorrect” on page 24-20

“Build Tool Not in System Path” on page 24-20

“Tool Path Does Not Exist” on page 24-21

“Unsupported Platform” on page 24-21

“Toolchain is Not installed” on page 24-22

“Project or Configuration is Using the Template Makefile” on page 24-22

“Skipped Validation of Build Tool “Download” or “Execute”” on page 24-23

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 `coder.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 `coder.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 `coder.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 'pltrma' platform. However, you are running on a 'pltrmb' platform.
```

To correct this issue:

- Check the `coder.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 24-19.

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 24-8
- “Create and Validate ToolchainInfo Object” on page 24-16
- “Register the Custom Toolchain” on page 24-17

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 25-2
- “Call a C/C++ Static Library Function from MATLAB Code” on page 25-4
- “Call Generated C/C++ Functions” on page 25-6
- “Use a C Dynamic Library in a Microsoft Visual Studio Project” on page 25-9
- “Specify External File Locations” on page 25-12
- “Code Generation of Matrices and Arrays” on page 25-16
- “Incorporate Generated Code Using an Example Main Function” on page 25-18
- “Use an Example C Main in an Application” on page 25-21
- “Package Code for Other Development Environments” on page 25-44
- “Structure of Generated Example C/C++ Main Function” on page 25-49
- “Troubleshoot Failures in Deployed Code” on page 25-53

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 25-4.

- 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 25-12.

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 21-29.

- 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 21-51.

- 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 25-6.

- 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 25-6

“How to Call C/C++ Functions from MATLAB Code” on page 25-6

“Calling Initialize and Terminate Functions” on page 25-7

“Calling C/C++ Functions with Multiple Outputs” on page 25-8

“Calling C/C++ Functions that Return Arrays” on page 25-8

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” on page 29-6.

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 25-7.
- 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 25-8.
- 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 25-8.

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 C Dynamic Library in a Microsoft Visual Studio Project

This example shows how to create and configure a simple Microsoft Visual Studio® Win32 Console Application project that calls a dynamic library (DLL) that MATLAB Coder generates. This example uses Microsoft Visual Studio 2013. In other versions of Microsoft Visual Studio, you might encounter a different procedure.

Generate a C Dynamic Library

- 1 Create a MATLAB function `foo`.

```
function c = foo(a) %#codegen
    c = sqrt(a);
end
```

- 2 Save it as `foo.m` in a local writable folder, for example, `c:\dll_test`.
- 3 Generate a DLL for the MATLAB function `foo`. Use 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`.

Create a Microsoft Visual Studio Project

In Microsoft Visual Studio, create an empty Win32 Console Application project. In Microsoft Visual Studio 2013:

- 1 On the Start page window, select **File > New > Project**.
- 2 In the New Product dialog box, select **Installed > Templates > Visual C++ > Win32 > Win32 Console Application** and enter a name.
- 3 In the Win32 Application Wizard, select **Application Settings**. Select the **Empty project** check box.
- 4 Click **Finish**.

Configure the Platform

Verify that the project configuration specifies the 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 2013:

- 1 Select **Build > Configuration Manager**.
- 2 In the Configuration Manager, set **Active solution platform** to match your platform.

Configure the Solution Version

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. However, by default, the DLL that MATLAB Coder generates uses the release version. In Microsoft Visual Studio 2013:

- 1 Select **Build > Configuration Manager**.
- 2 In the Configuration Manager, set **Active solution configuration** to **Release**.

Configure Additional Directories and Dependencies

- 1 Select **Project > Properties**.
- 2 Under **Configuration Properties > C/C++ > General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Include Directories**.
- 3 Under **Configuration Properties > Linker > General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Library Directories**.
- 4 Under **Configuration Properties > Linker > Input**, add `foo.lib` to **Additional Dependencies**.

Create a main.c File

Create a `main.c` file that calls `foo.dll`. The `main.c` 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;
}
```

Add the main.c File to the Project

- 1** Select **Project > Add Existing Item**.
- 2** Navigate to the folder that contains the main.c file.
- 3** Select the main.c file.

Build and Run the Executable

- 1** Build the executable. Select **Build > Build Solution**.
- 2** Make the .dll accessible to the executable. Either copy foo.dll to the folder containing the executable or add the folder containing foo.dll to your path.
- 3** Run the executable.

Specify External File Locations

In this section...

“External File Locations for External Code Integration” on page 25-12

“Specify External Files in a Class Derived from `coder.ExternalDependency`” on page 25-12

“Specify External Files in MATLAB Code Using `coder.updateBuildInfo`” on page 25-12

“Specify External Files Using the MATLAB Coder App” on page 25-13

“Specify External Files at the Command Line” on page 25-13

“Specify External Files with Configuration Objects” on page 25-14

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 Using the MATLAB Coder App

- 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set the **Build Type** to one of the following:
 - Source Code
 - Static Library
 - Dynamic Library
 - Executable
- 3 Click **More Settings**.
- 4 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 Additional files and directories to be built , provide an absolute path or a path relative to the project folder.	
Include directories	Specifies a list of folders that contain custom header, source, object, or library files. Separate list items with a semicolon.
Source files	Specifies additional custom C/C++ files to be compiled with the MATLAB file. Separate list items with a semicolon.
Libraries	Specifies the names of object or library files to be linked with the generated code. Separate list items with a semicolon.
Under Custom C-code to include in generated files	
Source file	Specifies code to appear at the top of generated C/C++ source files.
Header file	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 21-36.

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>Note: 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” on page 21-46

```
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);
...
```

Code Generation of Matrices and Arrays

MATLAB and MATLAB Coder software store matrix data and arrays (1-D, 2-D, ...) in column-major format as a vector. Column-major format orders elements in a matrix starting from the first column, top to bottom, and then moving to the next column. For example, in the following 3x3 matrix:

```
A =  
    1    2    3  
    4    5    6  
    7    8    9
```

translates to an array of length 9 in the following order:

```
A(1) = A(1,1) = 1;  
A(2) = A(2,1) = 4;  
A(3) = A(3,1) = 7;  
A(4) = A(1,2) = 2;  
A(5) = A(2,2) = 5;  
and so on.
```

In column-major format, the software accesses the next element of an array in memory by incrementing the first index of the array. For example, the software stores these element pairs sequentially in memory:

- $A(i)$ and $A(i+1)$
- $B(i, j)$ and $B(i+1, j)$
- $C(i, j, k)$ and $C(i+1, j, k)$

For more information on the internal representation of MATLAB data, see “MATLAB Data” in the MATLAB External Interfaces document.

Code generation software uses column-major format because:

- Much of the software that supports signal and array processing uses column-major format: MATLAB, LAPack, Fortran90, DSP libraries.
- A column is equivalent to a channel in frame-based processing. In this case, column-major storage is more efficient than row-major storage.
- A column-major array is self-consistent with its component submatrices:
 - A column-major 2-D array is a simple concatenation of 1-D arrays.
 - A column-major 3-D array is a simple concatenation of 2-D arrays.

- The stride is the number of memory locations to index to the next element in the same dimension. The stride of the first dimension is one element. The stride of the n th dimension element is the product of sizes of the lower dimensions.
- Row-major n -D arrays have their stride of 1 for the highest dimension. Submatrix manipulations typically access a scattered data set in memory, which does not allow for efficient indexing.

C typically uses row-major format. MATLAB uses column-major format. You cannot configure the code generation software to generate code with row-major ordering. If you are integrating legacy C code with the generated code, consider transposing the row-major data in your legacy C code into column-major format as a 1-D array.

Incorporate Generated Code Using an Example Main Function

In this section...

“Workflow for Using an Example Main Function” on page 25-18

“Control Example Main Generation Using the MATLAB Coder App” on page 25-19

“Control Example Main Generation Using the Command-Line Interface” on page 25-19

When you build an application that uses generated C/C++ code, you must provide a C/C++ main function that calls the generated code.

By default, for code generation of C/C++ source code, static libraries, dynamic libraries, and executables, MATLAB Coder generates an example C/C++ main function. This function is a template that can help you incorporate generated C/C++ code into your application. The example main function declares and initializes data, including dynamically allocated data. It calls entry-point functions but does not use values that the entry point functions return.

MATLAB Coder generates source and header files for the example main function in the `examples` subfolder of the build folder. For C code generation, it generates the files `main.c` and `main.h`. For C++ code generation, it generates the files `main.cpp` and `main.h`.

Do not modify the files `main.c` and `main.h` in the `examples` subfolder. If you do, when you regenerate code, MATLAB Coder does not regenerate the example main files. It warns you that it detects changes to the generated files. Before using the example main function, copy the example main source and header files to a location outside of the build folder. Modify the files in the new location to meet the requirements of your application.

The `packNGo` function and the **Package** option of the MATLAB Coder app do not package the example main source and header files when you generate the files using the default configuration settings. To package the example main files, configure code generation to generate and compile the example main function, generate your code, and then package the build files.


Workflow for Using an Example Main Function

- 1 Prepare your MATLAB code for code generation.
- 2 Check for run-time issues.

- 3 Make sure that example main generation is enabled.
- 4 Generate C/C++ code for the entry-point functions.
- 5 Copy the example main files from the `examples` subfolder to a different folder.
- 6 Modify the example main files in the new folder to meet the requirements of your application.
- 7 Deploy the example main and generated code for the platform that you want.
- 8 Build the application.

For an example that shows how to generate an example main and use it to build an executable, see “Use an Example C Main in an Application” on page 25-21.

Control Example Main Generation Using the MATLAB Coder App

- 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set the **Build Type** to one of the following:
 - Source Code
 - Static Library
 - Dynamic Library
 - Executable
- 3 Click **More Settings**.
- 4 On the **All Settings** tab, under **Advanced**, set **Generate example main** to one of the following:

Set To	For
Do not generate an example main function	Not generating an example C/C++ main function
Generate, but do not compile, an example main function (default)	Generating an example C/C++ main function but not compiling it
Generate and compile an example main function	Generating an example C/C++ main function and compiling it

Control Example Main Generation Using the Command-Line Interface

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

- ```
cfg = coder.config('lib'); % or dll or exe
```
- 2 Set the `GenerateExampleMain` property.

| Set To                       | For                                                            |
|------------------------------|----------------------------------------------------------------|
| 'DoNotGenerate'              | Not generating an example C/C++ main function                  |
| 'GenerateCodeOnly' (default) | Generating an example C/C++ main function but not compiling it |
| 'GenerateCodeAndCompile'     | Generating an example C/C++ main function and compiling it     |

For example:

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

## Related Examples

- “Structure of Generated Example C/C++ Main Function” on page 25-49
- “Call a C Static Library Function from C Code” on page 25-2

## More About

- “Specifying main Functions for C/C++ Executables” on page 21-25

## Use an Example C Main in an Application

This example shows how to build a C executable from MATLAB code that implements a simple Sobel filter to perform edge detection on images. The executable reads an image from the disk, applies the Sobel filtering algorithm, and then saves the modified image.

The example shows how to generate and modify an example main function that you can use when you build the executable.

### In this section...

“Prerequisites” on page 25-21

“Create a Folder and Copy Relevant Files” on page 25-22

“Run the Sobel Filter on the Image” on page 25-24

“Generate and Test a MEX Function” on page 25-26

“Generate an Example Main Function for sobel.m” on page 25-26

“Copy the Example Main Files” on page 25-29

“Modify the Generated Example Main Function” on page 25-29

“Generate the Sobel Filter Application” on page 25-42

“Run the Sobel Filter Application” on page 25-42

“Display the Resulting Image” on page 25-42

## Prerequisites

To complete this example, install the following products:

- MATLAB
- MATLAB Coder
- 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 “Change Default Compiler”.

## Create a Folder and Copy Relevant Files

The files you use in this example are:

| File Name | File Type     | Description                                                                                                                                                                                                                                                                                              |
|-----------|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| sobel.m   | Function code | MATLAB implementation of a Sobel filtering algorithm. <code>sobel.m</code> takes an image (represented as a double matrix) and a threshold value as inputs. The algorithm detects edges in the image (based on the threshold value). <code>sobel.m</code> returns a modified image displaying the edges. |
| hello.jpg | Image file    | Image that the Sobel filter modifies.                                                                                                                                                                                                                                                                    |

### Contents of File `sobel.m`

```
function edgeImage = sobel(originalImage, threshold) %#codegen

% 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.

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);
```

### Contents of hello.jpg



To copy the example files to a local working folder:

- 1 Create a local working folder. For example, `c:\coder\edge_detection`.
- 2 Navigate to the working folder.
- 3 Copy the files `sobel.m` and `hello.jpg` from the examples folder `sobel` to your working folder.

```
copyfile(fullfile(docroot, 'toolbox', 'coder', 'examples', 'sobel'))
```

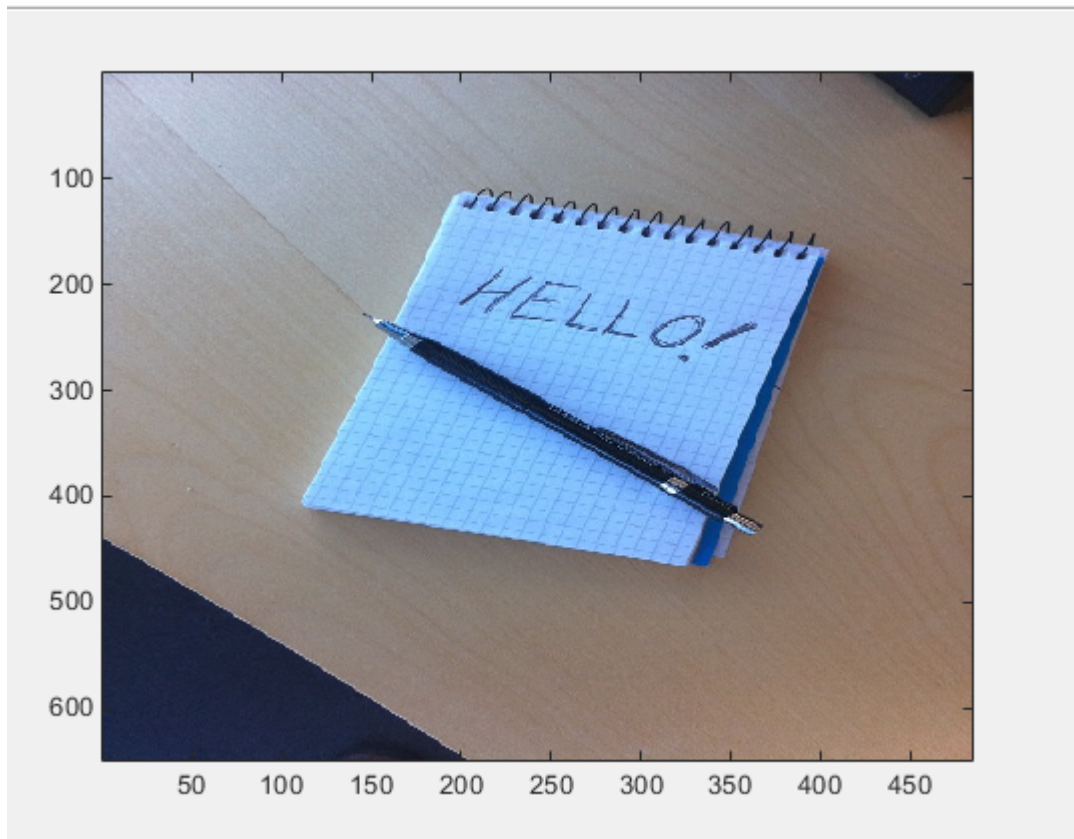
## Run the Sobel Filter on the Image

- 1 Read the original image into a MATLAB matrix and display it.

```
im = imread('hello.jpg');
```

- 2 Display the image as a basis for comparison to the result of the Sobel filter.

```
image(im);
```



- 3 The Sobel filtering algorithm operates on grayscale images. Convert the color image 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)))/255;
```

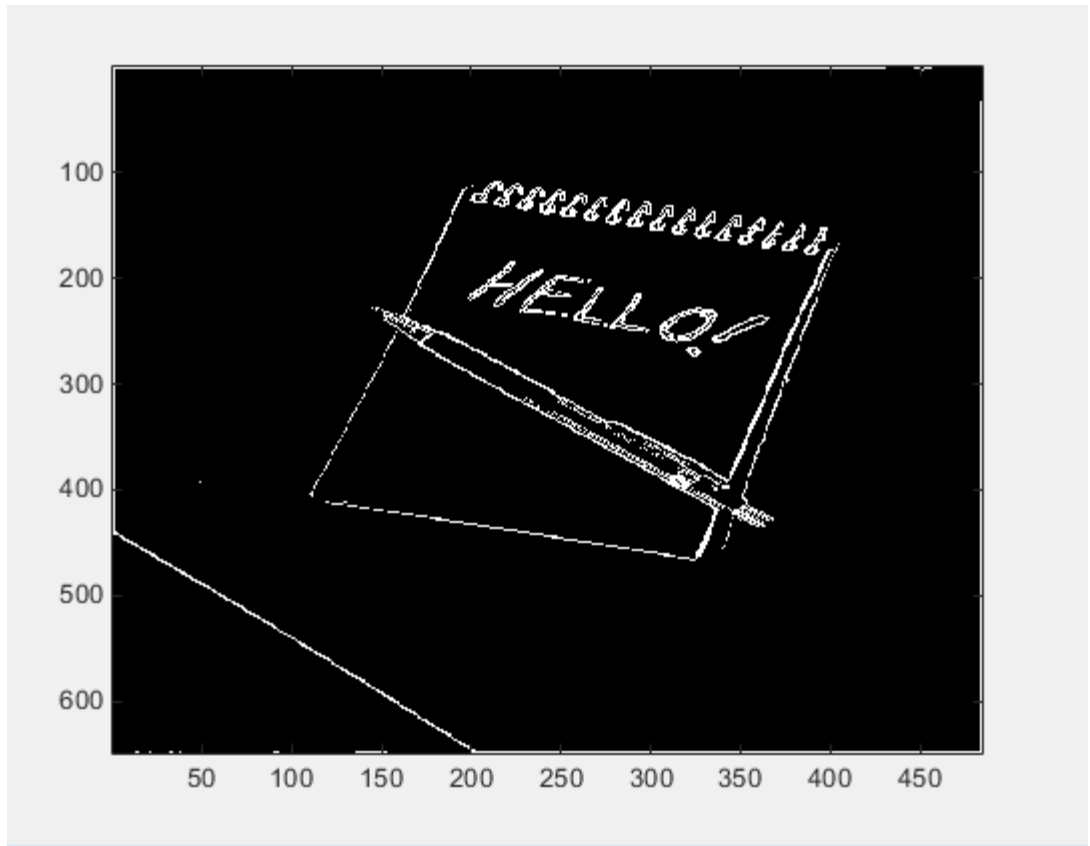


- 4 To run the MATLAB function for the Sobel filter, pass the grayscale image matrix `gray` and a threshold value to the function `sobel`. This example uses 0.7 for a threshold value.

```
edgeIm = sobel(gray, 0.7);
```

- 5 To display the modified image, reformat the matrix `edgeIm` with the function `repmat` so that you can pass it to the `image` command.

```
im3 = repmat(edgeIm, [1 1 3]);
image(im3);
```



## Generate and Test a MEX Function

- 1 To test that generated code is functionally equivalent to the original MATLAB code and that run-time errors do not occur, generate a MEX function.

```
codegen -report sobel
```

codegen generates a MEX function named `sobel_mex` in the current working folder.

- 2 To run the MEX function for the Sobel filter, pass the grayscale image matrix `gray` and a threshold value to the function `sobel_mex`. This example uses 0.7 for a threshold value.

```
edgeImMex = sobel_mex(gray, 0.7);
```

- 3 To display the modified image, reformat the matrix `edgeImMex` with the function `repmat` so that you can pass it to the `image` command.

```
im3Mex = repmat(edgeImMex, [1 1 3]);
image(im3Mex);
```

This image is the same as the image created using the MATLAB function.

## Generate an Example Main Function for `sobel.m`

Although you can write a custom main function for your application, an example main function provides a template to help you incorporate the generated code.

To generate an example main function for the Sobel filter:

- 1 Create a configuration object for a C static library.

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

For configuration objects for C/C++ source code, static libraries, dynamic libraries, and executables, the setting `GenerateExampleMain` controls generation of the example main function. The setting is set to `'GenerateCodeOnly'` by default, which generates the example main function but does not compile it. For this example, do not change the value of the `GenerateExampleMain` setting.

- 2 Generate a C static library using the configuration object.

```
codegen -report -config cfg sobel
```

The generated files for the static library are in the folder `codegen/lib/sobel`. The example main files are in the subfolder `codegen/lib/sobel/examples`.

### Contents of Example Main File `main.c`

```

/*
 * main.c
 *
 * Code generation for function 'main'
 *
 */

/*****
/* This automatically generated example C main file shows how to call
/* entry-point functions that MATLAB Coder generated. You must customize
/* this file for your application. Do not modify this file directly.
/* Instead, make a copy of this file, modify it, and integrate it into
/* your development environment.
/*
/* This file initializes entry-point function arguments to a default
/* size and value before calling the entry-point functions. It does
/* not store or use any values returned from the entry-point functions.
/* If necessary, it does pre-allocate memory for returned values.
/* You can use this file as a starting point for a main function that
/* you can deploy in your application.
/*
/* After you copy the file, and before you deploy it, you must make the
/* following changes:
/* * For variable-size function arguments, change the example sizes to
/* the sizes that your application requires.
/* * Change the example values of function arguments to the values that
/* your application requires.
/* * If the entry-point functions return values, store these values or
/* otherwise use them as required by your application.
/*
/*****
/* Include files */
#include "rt_nonfinite.h"
#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"

```

```
/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(void);
static double argInit_real_T(void);
static void main_sobel(void);

/* Function Definitions */
static emxArray_real_T *argInit_d1024xd1024_real_T(void)
{
 emxArray_real_T *result;
 static int iv2[2] = { 2, 2 };

 int b_j0;
 int b_j1;

 /* Set the size of the array.
 Change this size to the value that the application requires. */
 result = emxCreateND_real_T(2, iv2);

 /* Loop over the array to initialize each element. */
 for (b_j0 = 0; b_j0 < result->size[0U]; b_j0++) {
 for (b_j1 = 0; b_j1 < result->size[1U]; b_j1++) {
 /* Set the value of the array element.
 Change this value to the value that the application requires. */
 result->data[b_j0 + result->size[0] * b_j1] = argInit_real_T();
 }
 }

 return result;
}

static double argInit_real_T(void)
{
 return 0.0;
}

static void main_sobel(void)
{
 emxArray_uint8_T *edgeImage;
 emxArray_real_T *originalImage;
 emxInitArray_uint8_T(&edgeImage, 2);

 /* Initialize function 'sobel' input arguments. */
 /* Initialize function input argument 'originalImage'. */
 originalImage = argInit_d1024xd1024_real_T();
}
```

```

 /* Call the entry-point 'sobel'. */
 sobel(originalImage, argInit_real_T(), edgeImage);
 emxDestroyArray_uint8_T(edgeImage);
 emxDestroyArray_real_T(originalImage);
}

int main(int argc, const char * const argv[])
{
 (void)argc;
 (void)argv;

 /* Initialize the application.
 You do not need to do this more than one time. */
 sobel_initialize();

 /* Invoke the entry-point functions.
 You can call entry-point functions multiple times. */
 main_sobel();

 /* Terminate the application.
 You do not need to do this more than one time. */
 sobel_terminate();
 return 0;
}

/* End of code generation (main.c) */

```

## Copy the Example Main Files

Do not modify the files `main.c` and `main.h` in the `examples` subfolder. If you do, when you regenerate code, MATLAB Coder does not regenerate the example main files. It warns you that it detects changes to the generated files.

Copy the files `main.c` and `main.h` from the folder `codegen/lib/sobel/examples` to another location. For this example, copy the files to the current working folder. Modify the files in the new location.

## Modify the Generated Example Main Function

- “Modify the Function main” on page 25-30
- “Modify the Initialization Function `argInit_d1024xd1024_real_T`” on page 25-32

- “Write the Function `saveImage`” on page 25-34
- “Modify the Function `main_sobel`” on page 25-36
- “Modify the Function Declarations” on page 25-37
- “Modify the Include Files” on page 25-37
- “Contents of Modified File `main.c`” on page 25-38

The example main function declares and initializes data, including dynamically allocated data, to zero values. It calls entry-point functions with arguments set to zero values, but it does not use values returned from the entry-point functions.

The C main function must meet the requirements of your application. This example modifies the example main function to meet the requirements of the Sobel filter application.

This example modifies the file `main.c` so that the Sobel filter application:

- Reads in the grayscale image from a binary file.
- Applies the Sobel filtering algorithm.
- Saves the modified image to a binary file.

### **Modify the Function `main`**

Modify the function `main` to:

- Accept the file containing the grayscale image data and a threshold value as input arguments.
- Call the function `main_sobel` with the address of the grayscale image data stream and the threshold value as input arguments.

In the function `main`:

- 1** Remove the declarations `void(argc)` and `(void)argv`.
- 2** Declare the variable `filename` to hold the name of the binary file containing the grayscale image data.  

```
const char *filename;
```
- 3** Declare the variable `threshold` to hold the threshold value.  

```
double threshold;
```

- 4 Declare the variable `fd` to hold the address of the grayscale image data that the application reads in from `filename`.

```
FILE *fd;
```

- 5 Add an `if` statement that checks for three arguments.

```
if (argc != 3) {
 printf("Expected 2 arguments: filename and threshold\n");
 exit(-1);
}
```

- 6 Assign the input argument `argv[1]` for the file containing the grayscale image data to `filename`.

```
filename = argv[1];
```

- 7 Assign the input argument `argv[2]` for the threshold value to `threshold`, converting the input from a string to a numeric double.

```
threshold = atof(argv[2]);
```

- 8 Open the file containing the grayscale image data whose name is specified in `filename`. Assign the address of the data stream to `fd`.

```
fd = fopen(filename, "rb");
```

- 9 To verify that the executable can open `filename`, write an `if`-statement that exits the program if the value of `fd` is `NULL`.

```
if (fd == NULL) {
 exit(-1);
}
```

- 10 Replace the function call for `main_sobel` by calling `main_sobel` with input arguments `fd` and `threshold`.

```
main_sobel(fd, threshold);
```

- 11 Close the grayscale image file after calling `sobel_terminate`.

```
fclose(fd);
```

### Modified Function `main`

```
int main(int argc, const char * const argv[])
{
 const char *filename;
 double threshold;
```

```
FILE *fd;

if (argc != 3) {
 printf("Expected 2 arguments: filename and threshold\n");
 exit(-1);
}

filename = argv[1];
threshold = atof(argv[2]);
fd = fopen(filename, "rb");
if (fd == NULL) {
 exit(-1);
}
/* Initialize the application.
 You do not need to do this more than one time. */
sobel_initialize();

/* Invoke the entry-point functions.
 You can call entry-point functions multiple times. */
main_sobel(fd, threshold);

/* Terminate the application.
 You do not need to do this more than one time. */
sobel_terminate();

fclose(fd);

return 0;
}
```

### **Modify the Initialization Function `argInit_d1024xd1024_real_T`**

In the example main file, the function `argInit_d1024xd1024_real_T` creates a dynamically allocated variable-size array (`emxArray`) for the image that you pass to the Sobel filter. This function initializes the `emxArray` to a default size and the elements of the `emxArray` to 0. It returns the initialized `emxArray`.

For the Sobel filter application, modify the function to read the grayscale image data from a binary file into the `emxArray`.

In the function `argInit_d1024xd1024_real_T`:

- 1 Replace the input argument `void` with the argument `FILE *fd`. This variable points to the grayscale image data that the function reads in.



```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
```

- 2 Change the values of the variable `iv2` to match the dimensions of the grayscale image matrix `gray`. `iv2` holds the size values for the dimensions of the `emxArray` that `argInit_d1024xd1024_real_T` creates.

```
static int iv2[2] = { 484, 648 };
```

MATLAB stores matrix data in column-major format, while C stores matrix data in row-major format. Declare the dimensions accordingly.

- 3 Define a variable `element` to hold the values read in from the grayscale image data.

```
double element;
```

- 4 Change the `for`-loop construct to read data points from the normalized image into `element` by adding an `fread` command to the inner `for`-loop.

```
fread(&element, 1, sizeof(element), fd);
```

- 5 Inside the `for`-loop, assign `element` as the value set for the `emxArray` data.

```
result->data[b_j0 + result->size[0] * b_j1] = element;
```

### Modified Initialization Function `argInit_d1024xd1024_real_T`

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
 emxArray_real_T *result;
 static int iv2[2] = { 484, 648 };

 int b_j0;
 int b_j1;
 double element;

 /* Set the size of the array.
 Change this size to the value that the application requires. */
 result = emxCreateND_real_T(2, iv2);

 /* Loop over the array to initialize each element. */
 for (b_j0 = 0; b_j0 < result->size[0]; b_j0++) {
 for (b_j1 = 0; b_j1 < result->size[1]; b_j1++) {
 /* Set the value of the array element.
 Change this value to the value that the application requires. */
 fread(&element, 1, sizeof(element), fd);
 result->data[b_j0 + result->size[0] * b_j1] = element;
 }
 }
}
```

```
 }
 }

 return result;
}
```

### Write the Function `saveImage`

The MATLAB function `sobel.m` interfaces with MATLAB arrays, but the Sobel filter application interfaces with binary files.

To save the image modified by the Sobel filtering algorithm to a binary file, create a function `saveImage`. The function `saveImage` writes data from an `emxArray` into a binary file. It uses a construction that is similar to the one used by the function `argInit_d1024xd1024_real_T`.

In the file `main.c`:

- 1 Define the function `saveImage` that takes the address of `emxArray` `edgeImage` as an input and has output type `void`.

```
static void saveImage(emxArray_uint8_T *edgeImage)
{
}
```

- 2 Define the variables `b_j0` and `b_j1` like they are defined in the function `argInit_d1024xd1024_real_T`.

```
int b_j0;
int b_j1;
```

- 3 Define the variable `element` to store data read from the `emxArray`.

```
uint8_T element;
```

- 4 Open a binary file `edge.bin` for writing the modified image. Assign the address of `edge.bin` to `FILE *fd`.

```
FILE *fd = fopen("edge.bin", "wb");
```

- 5 To verify that the executable can open `edge.bin`, write an `if`-statement that exits the program if the value of `fd` is `NULL`.

```
if (fd == NULL) {
 exit(-1);
}
```

- 6 Write a nested for-loop construct like the one in the function `argInit_d1024xd1024_real_T`.

```
for (b_j0 = 0; b_j0 < edgeImage->size[0U]; b_j0++)
{
 for (b_j1 = 0; b_j1 < edgeImage->size[1U]; b_j1++)
 {
 }
}
```

- 7 Inside the inner for-loop, assign the values from the modified image data to `element`.

```
element = edgeImage->data[b_j0 + edgeImage->size[0] * b_j1];
```

- 8 After the assignment for `element`, write the value from `element` to the file `edge.bin`.

```
fwrite(&element, 1, sizeof(element), fd);
```

- 9 After the for-loop construct, close `fd`.

```
fclose(fd);
```

### Function `saveImage`

```
static void saveImage(emxArray_uint8_T *edgeImage)
{
 int b_j0;
 int b_j1;
 uint8_T element;

 FILE *fd = fopen("edge.bin", "wb");
 if (fd == NULL) {
 exit(-1);
 }
 /* Loop over the array to save each element. */
 for (b_j0 = 0; b_j0 < edgeImage->size[0U]; b_j0++) {
 for (b_j1 = 0; b_j1 < edgeImage->size[1U]; b_j1++) {
 element = edgeImage->data[b_j0 + edgeImage->size[0] * b_j1];
 fwrite(&element, 1, sizeof(element), fd);
 }
 }
 fclose(fd);
}
```

### Modify the Function `main_sobel`

In the example main function, the function `main_sobel` creates `emxArrays` for the data for the grayscale and modified images. It calls the function `argInit_d1024xd1024_real_T` to initialize the `emxArray` for the grayscale image. `main_sobel` passes both `emxArrays` and the threshold value of 0 that the initialization function `argInit_real_T` returns to the function `sobel`. When the function `main_sobel` ends, it discards the result of the function `sobel`.

For the Sobel filter application, modify the function `main_sobel` to:

- Take the address of the grayscale image data and the threshold value as inputs.
- Read the data from the address using `argInit_d1024xd1024_real_T`.
- Pass the data to the Sobel filtering algorithm with the threshold value `threshold`.
- Save the result using `saveImage`.

In the function `main_sobel`:

- 1 Replace the input arguments to the function with the arguments `FILE *fd` and `double threshold`.

```
static void main_sobel(FILE *fd, double threshold)
```

- 2 Pass the input argument `fd` to the function call for `argInit_d1024xd1024_real_T`.

```
originalImage = argInit_d1024xd1024_real_T(fd);
```

- 3 Replace the threshold value input in the function call to `sobel` with `threshold`.

```
sobel(originalImage, threshold, edgeImage);
```

- 4 After calling the function `sobel`, call the function `saveImage` with the input `edgeImage`.

```
saveImage(edgeImage);
```

### Modified Function `main_sobel`

```
static void main_sobel(FILE *fd, double threshold)
{
 emxArray_uint8_T *edgeImage;
 emxArray_real_T *originalImage;
 emxInitArray_uint8_T(&edgeImage, 2);
```

```

/* Initialize function 'sobel' input arguments. */
/* Initialize function input argument 'originalImage'. */
originalImage = argInit_d1024xd1024_real_T(fd);

/* Call the entry-point 'sobel'. */
sobel(originalImage, threshold, edgeImage);

saveImage(edgeImage);

emxDestroyArray_uint8_T(edgeImage);
emxDestroyArray_real_T(originalImage);
}

```

### Modify the Function Declarations

To match the changes that you made to the function definitions, make the following changes to the function declarations:

- 1 Change the input of the function `*argInit_d1024xd1024_real_T` to `FILE *fd`.

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
```

- 2 Change the inputs of the function `main_sobel` to `FILE *fd` and `double threshold`.

```
static void main_sobel(FILE *fd, double threshold);
```

- 3 Add the function `saveImage`.

```
static void saveImage(emxArray_uint8_T *edgeImage);
```

### Modified Function Declarations

```

/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
static void saveImage(emxArray_uint8_T *edgeImage);
static double argInit_real_T(void);
static void main_sobel(FILE *fd, double threshold);

```

### Modify the Include Files

For input/output functions that you use in `main.c`, add the header file `stdio.h` to the included files list.

```
#include <stdio.h>
```

**Modified Include Files**

```
/* Include Files */
#include <stdio.h>

#include "rt_nonfinite.h"
#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"
```

**Contents of Modified File main.c****main.c**

```
/*
 * main.c
 *
 * Code generation for function 'main'
 */

/*****
/* This automatically generated example C main file shows how to call
/* entry-point functions that MATLAB Coder generated. You must customize
/* this file for your application. Do not modify this file directly.
/* Instead, make a copy of this file, modify it, and integrate it into
/* your development environment.
/*
/* This file initializes entry-point function arguments to a default
/* size and value before calling the entry-point functions. It does
/* not store or use any values returned from the entry-point functions.
/* If necessary, it does pre-allocate memory for returned values.
/* You can use this file as a starting point for a main function that
/* you can deploy in your application.
/*
/* After you copy the file, and before you deploy it, you must make the
/* following changes:
/* * For variable-size function arguments, change the example sizes to
/* the sizes that your application requires.
/* * Change the example values of function arguments to the values that
/* your application requires.
/* * If the entry-point functions return values, store these values or
/* otherwise use them as required by your application.
*****/
```

```
/* */
/***** */
/* Include Files */
#include <stdio.h>

#include "rt_nonfinite.h"
#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"

/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
static void saveImage(emxArray_uint8_T *edgeImage);
static double argInit_real_T(void);
static void main_sobel(FILE *fd, double threshold);

/* Function Definitions */

static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
 emxArray_real_T *result;
 static int iv2[2] = { 484, 648 };

 int b_j0;
 int b_j1;
 double element;

 /* Set the size of the array.
 Change this size to the value that the application requires. */
 result = emxCreateND_real_T(2, iv2);

 /* Loop over the array to initialize each element. */
 for (b_j0 = 0; b_j0 < result->size[0U]; b_j0++) {
 for (b_j1 = 0; b_j1 < result->size[1U]; b_j1++) {
 /* Set the value of the array element.
 Change this value to the value that the application requires. */
 fread(&element, 1, sizeof(element), fd);
 result->data[b_j0 + result->size[0] * b_j1] = element;
 }
 }

 return result;
}
```

```
}

static void saveImage(emxArray_uint8_T *edgeImage)
{
 int b_j0;
 int b_j1;
 uint8_T element;

 FILE *fd = fopen("edge.bin", "wb");
 if (fd == NULL) {
 exit(-1);
 }
 /* Loop over the array to save each element. */
 for (b_j0 = 0; b_j0 < edgeImage->size[0U]; b_j0++) {
 for (b_j1 = 0; b_j1 < edgeImage->size[1U]; b_j1++) {
 element = edgeImage->data[b_j0 + edgeImage->size[0] * b_j1];
 fwrite(&element, 1, sizeof(element), fd);
 }
 }
 fclose(fd);
}

/*
 * Arguments : void
 * Return Type : double
 */
static double argInit_real_T(void)
{
 return 0.0;
}

static void main_sobel(FILE *fd, double threshold)
{
 emxArray_uint8_T *edgeImage;
 emxArray_real_T *originalImage;
 emxInitArray_uint8_T(&edgeImage, 2);

 /* Initialize function 'sobel' input arguments. */
 /* Initialize function input argument 'originalImage'. */
 originalImage = argInit_d1024xd1024_real_T(fd);

 /* Call the entry-point 'sobel'. */
 sobel(originalImage, threshold, edgeImage);
}
```



```
 saveImage(edgeImage);

 emxDestroyArray_uint8_T(edgeImage);
 emxDestroyArray_real_T(originalImage);
}

int main(int argc, const char * const argv[])
{
 const char *filename;
 double threshold;
 FILE *fd;

 if (argc != 3) {
 printf("Expected 2 arguments: filename and threshold\n");
 exit(-1);
 }

 filename = argv[1];
 threshold = atof(argv[2]);
 fd = fopen(filename, "rb");
 if (fd == NULL) {
 exit(-1);
 }
 /* Initialize the application.
 You do not need to do this more than one time. */
 sobel_initialize();

 /* Invoke the entry-point functions.
 You can call entry-point functions multiple times. */
 main_sobel(fd, threshold);

 /* Terminate the application.
 You do not need to do this more than one time. */
 sobel_terminate();

 fclose(fd);

 return 0;
}

/* End of code generation (main.c) */
```

## Generate the Sobel Filter Application

- 1 Navigate to the working folder if you are not currently in it.
- 2 Create a configuration object for a C standalone executable.
- 3 Generate a C standalone executable for the Sobel filter using the configuration object and the modified main function.

```
cfg = coder.config('exe');
```

```
codegen -report -config cfg sobel main.c main.h
```

By default, if you are running MATLAB on a Windows platform, the executable `sobel.exe` is generated in the current working folder. If you are running MATLAB on a platform other than Windows, the file extension is the corresponding extension for that platform. By default, the code generated for the executable is in the folder `codegen/exe/sobel`.

## Run the Sobel Filter Application

- 1 Create the MATLAB matrix `gray` if it is not currently in your MATLAB workspace:

```
im = imread('hello.jpg');
```

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
```

- 2 Write the matrix `gray` into a binary file using the `fopen` and `fwrite` commands. The application reads in this binary file.

```
fid = fopen('gray.bin', 'w');
fwrite(fid, gray, 'double');
fclose(fid);
```

- 3 Run the executable, passing to it the file `gray.bin` and the threshold value 0.7.

To run the example in MATLAB on a Windows platform:

```
system('sobel.exe gray.bin 0.7');
```

The executable generates the file `edge.bin`.

## Display the Resulting Image

- 1 Read the file `edge.bin` into a MATLAB matrix `edgeImExe` using the `fopen` and `fread` commands.

```
fd = fopen('edge.bin', 'r');
edgeImExe = fread(fd, size(gray), 'uint8');
fclose(fd);
```

- 2 Pass the matrix `edgeImExe` to the function `repmat` and display the image.

```
im3Exe = repmat(edgeImExe, [1 1 3]);
image(im3Exe);
```

The image matches the images from the MATLAB and MEX functions.

## Related Examples

- “Structure of Generated Example C/C++ Main Function” on page 25-49
- “Incorporate Generated Code Using an Example Main Function” on page 25-18
- “Call a C Static Library Function from C Code” on page 25-2

## Package Code for Other Development Environments

### In this section...

“When to Package Code” on page 25-44

“Package Generated Code Using the MATLAB Coder App” on page 25-44

“Package Generated Code at the Command Line” on page 25-45

“Specify packNGo Options” on page 25-47

### When to Package Code

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 the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

See “Package Generated Code Using the MATLAB Coder App” on page 25-44 and “Package Generated Code at the Command Line” on page 25-45.

### Package Generated Code Using the MATLAB Coder App

This example shows how to package generated code into a zip file for relocation using the **Package** option in the MATLAB Coder app. By default, MATLAB Coder creates the zip file in the current working 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 Open the MATLAB Coder app. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.
- 3 On the **Select Source Files** page, enter the name of the entry-point function `foo`. Click **Next** to go to the **Define Input Types** page.
- 4 Specify that inputs `A` and `B` are scalar doubles. Click **Next** to go to the **Check for Run-Time Issues** page.

- 5 Check for run-time issues. In the **Check for Run-Time Issues** dialog box, enter code that calls `foo` with scalar double inputs. For example:

```
foo(1,2)
```

Click **Check for Issues**.

To check for run-time issues, the app generates and runs a MEX function. The app does not find issues for `foo`. Click **Next** to go to the **Generate Code** page.

- 6 In the **Generate** dialog box, set the **Build Type** to **Source Code**, **Static Library**, **Dynamic Library**, or **Executable**. You cannot package the code generated for MEX targets.
- 7 Click **Generate**. Click **Next** to go to the **Finish Workflow** page.
- 8 On the **Finish Workflow** page, click **Package**.
- 9 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, in this case `foo_pkg.zip`, 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
  - Makefiles
  - Example main files, unless you configure code generation to generate and compile the example main function. See “Incorporate Generated Code Using an Example Main Function” on page 25-18.
- 10 Inspect the contents of `foo_pkg.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially open and inspect the file without unpacking it.

You can now relocate the resulting zip file to the desired 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
- Makefiles
- Example main files, unless you configure code generation to generate and compile the example main function. See “Incorporate Generated Code Using an Example Main Function” on page 25-18.

In this example, you specify only the file name. Optionally, you can specify additional packaging options. See “Specify `packNGo` Options” on page 25-47.

- 5 Inspect the contents of `foo.zip` to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially 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 desired 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'...<br/>'fileName' 'zippedsrcs'});</code> |
| Include all header files found on the include path in the zip file (rather than the minimal header files required to build the code) | <code>packNGo(buildInfo, {'minimalHeaders' false});</code>                                   |
| Generate warnings for parse errors and missing files                                                                                 | <code>packNGo(buildInfo, {'ignoreParseError' true...<br/>'ignoreFileMissing' true});</code>  |

For more information, see `packNGo` in “Build Information Methods” on page 21-140.

### Choose a Structure for the Zip File

Before you generate and package the files, decide whether you want to package the files 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                                                                                                                                                          | Use                             |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 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 | A single, flat folder structure |

| If                                                                                                                                                                                                 | Use                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| 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 | A 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.



## Structure of Generated Example C/C++ Main Function

### In this section...

“Contents of the File `main.c` or `main.cpp`” on page 25-49

“Contents of the File `main.h`” on page 25-52

When you build an application that uses generated C/C++ code, you must provide a C/C++ main function that calls the generated code.

By default, for code generation of C/C++ source code, static libraries, dynamic libraries, and executables, MATLAB Coder generates an example C/C++ main function. This function is a template that can help you incorporate generated C/C++ code into your application. The example main function declares and initializes data, including dynamically allocated data. It calls entry-point functions but does not use values that the entry point functions return. To use the example main function, copy the example main source and header files to a location outside of the build folder, and then modify the files in the new location to meet the requirements of your application.

MATLAB Coder generates source and header files for the example main function in the `examples` subfolder of the build folder. For C code generation, it generates the files `main.c` and `main.h`. For C++ code generation, it generates the files `main.cpp` and `main.h`.

### Contents of the File `main.c` or `main.cpp`

For the example main source file `main.c` or `main.cpp`, MATLAB Coder generates the following sections:

- “Include Files” on page 25-50
- “Function Declarations” on page 25-50
- “Argument Initialization Functions” on page 25-50
- “Entry-Point Functions” on page 25-51
- “Main Function” on page 25-51

By default, MATLAB Coder also generates comments in the example main source file that can help you modify the example main function to use in your application.

## Include Files

This section includes the header files required to call code that is not in the example main source file. If you call external functions when you modify the example main source file, include any other required header files.

## Function Declarations

This section declares the function prototypes for the argument initialization and entry-point functions that are defined in the example main source file. Modify the function prototypes to match modifications that you make in the function definitions. Declare new function prototypes for functions that you define in the example main source file.

## Argument Initialization Functions

This section defines an initialization function for each data type that the entry-point functions use as an argument. The argument initialization function initializes the size of the argument to a default value and the values of the data to zero. The function then returns the initialized data. Change these size and data values to meet the requirements of your application.

For an argument with dimensions of size `<dimSizes>` and MATLAB C/C++ data type `<baseType>`, the example main source file defines an initialization function with the name `argInit_<dimSizes>_<baseType>`. For example, for a 5-by-5 array with data of MATLAB type `double`, the example main source file defines the argument initialization function `argInit_5x5_real_T`.

MATLAB Coder alters the name of the argument initialization functions as follows:

- If any of the dimensions are variable-size, MATLAB Coder designates the size of these dimensions as `d<maxSize>`, where `<maxSize>` is the maximum size of that dimension. For example, for an array with data of MATLAB type `double` with a first dimension of static size 2 and a second dimension that can vary in size up to 10, the example main source file defines the argument initialization function `argInit_2xd10_real_T`.
- If any of the dimensions are unbounded, MATLAB Coder designates the size of these dimensions as `Unbounded`.
- If the return type of the initialization function is an `emxArray`, MATLAB Coder defines the function as returning a pointer to the `emxArray`.
- If the length of the initialization function name exceeds the maximum number of characters set for function names in the configuration settings, MATLAB Coder

prepends an identifier string to the front of the function name. MATLAB Coder then truncates the function name to the maximum allowed number of characters for identifier length.

---

**Note:** By default, the maximum number of characters allowed for generated identifiers is 31. To specify the value set for the maximum identifier length using the MATLAB Coder app, select the **Maximum identifier length** value on the **Code Appearance** tab of the code generation settings. To specify the value set for the maximum identifier using the command-line interface, change the value of the `MaxIdLength` configuration object setting.

---

### Entry-Point Functions

This section defines a function for each MATLAB entry-point function. For a MATLAB function `foo.m`, the example main source file defines an entry-point function `main_foo`. This function creates the variables and calls the data initialization functions that the C/C++ source function `foo.c` or `foo.cpp` requires. It calls this C/C++ source function but does not return the result. Modify `main_foo` so that it takes inputs and returns outputs as required by your application.

### Main Function

This section defines a `main` function that does the following:

- If your output language is C, it declares and names the variables `argc` and `argv` but casts them to void. If your output language is C++, the generated example main declares, but does not name, the variables `argc` and `argv`.
- Calls the initialize function `foo_initialize`, which is named for the alphabetically first entry-point function `foo` declared for code generation. Call the initialize function only once, even if you have multiple entry-point functions called in the function `main`.
- Calls each of the entry-point functions once.
- Calls the terminate function `foo_terminate`, which is named for the alphabetically first entry-point function `foo` declared for code generation. Call the terminate function only once, even if you have multiple entry-point functions called in the function `main`.
- Returns zero.

Modify the function `main`, including the inputs and outputs of `main` and of the entry-point functions, to meet the requirements of your application.

## Contents of the File `main.h`

For the example main header file `main.h`, MATLAB Coder generates the following:

- “Include Guard” on page 25-52
- “Include Files” on page 25-52
- “Function Declarations” on page 25-52

By default, MATLAB Coder also generates comments in `main.h` that can help you modify the example main function to use in your application.

### Include Guard

`main.h` uses an include guard to prevent the contents of the file from being included multiple times. The include guard contains the include files and function declarations within an `#ifndef` construct.

### Include Files

`main.h` includes the header files required to call code that is not defined within it.

### Function Declarations

`main.h` declares the function prototype for the main function that is defined in the example main source file `main.c` or `main.cpp`.

## Related Examples

- “Incorporate Generated Code Using an Example Main Function” on page 25-18
- “Use an Example C Main in an Application” on page 25-21

## More About

- “How MATLAB Coder Infers C/C++ Data Types” on page 27-9

## Troubleshoot Failures in Deployed Code

If your deployed code fails, consider regenerating the code with run-time error detection enabled. When you enable run-time error detection, the generated code includes code that detects and reports errors, such as out-of-bounds array indexing. If the code detects one of these errors, it reports a message and terminates the program. Running the code that includes the error checks helps you to see if one of these errors caused the failure.

Run-time error detection can affect the performance of the generated code. If performance is a consideration for your application, when you finish troubleshooting, regenerate the code with run-time error detection disabled.

See “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 22-28 and “Generate Standalone Code That Detects and Reports Run-Time Errors” on page 22-30.

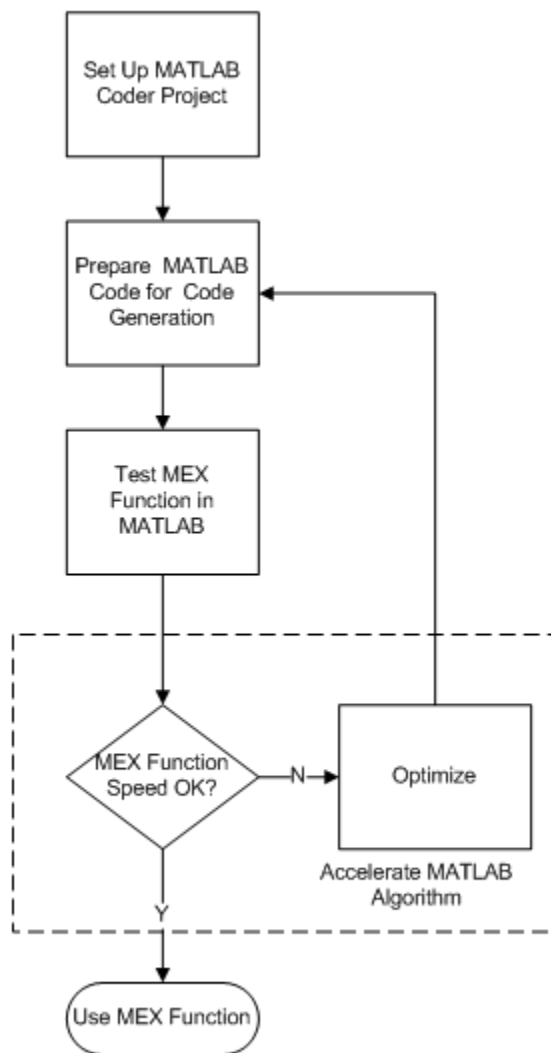


# Accelerating MATLAB Algorithms

---

- “Workflow for Accelerating MATLAB Algorithms” on page 26-2
- “Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms” on page 26-4
- “Edge Detection on Images” on page 26-7
- “Accelerate MATLAB Algorithms” on page 26-14
- “Modifying MATLAB Code for Acceleration” on page 26-15
- “Control Run-Time Checks” on page 26-16
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 26-19
- “Control Compilation of parfor-Loops” on page 26-25
- “Reduction Assignments in parfor-Loops” on page 26-26
- “Classification of Variables in parfor-Loops” on page 26-27
- “Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)” on page 26-36
- “Specify Maximum Number of Threads in parfor-Loops” on page 26-37
- “Troubleshooting parfor-Loops” on page 26-38
- “Accelerating Simulation of Bouncing Balls” on page 26-39

## Workflow for Accelerating MATLAB Algorithms





## See Also

- “Set Up a MATLAB Coder Project” on page 18-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 19-2
- “Workflow for Testing MEX Functions in MATLAB” on page 20-2
- “Modifying MATLAB Code for Acceleration” on page 26-15

## Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms

### In this section...

“Accelerate Code That Dominates Execution Time” on page 26-4

“Include Loops Inside MEX Function” on page 26-4

“Avoid Generating MEX Functions from Unsupported Functions” on page 26-5

“Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time” on page 26-6

“Minimize MEX Function Calls” on page 26-6

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 MATLAB code, see “Profile to Improve Performance”.

### 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” on page 4-2.

---

**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” on page 14-11.

---

## 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” on page 14-11.

---

## 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.

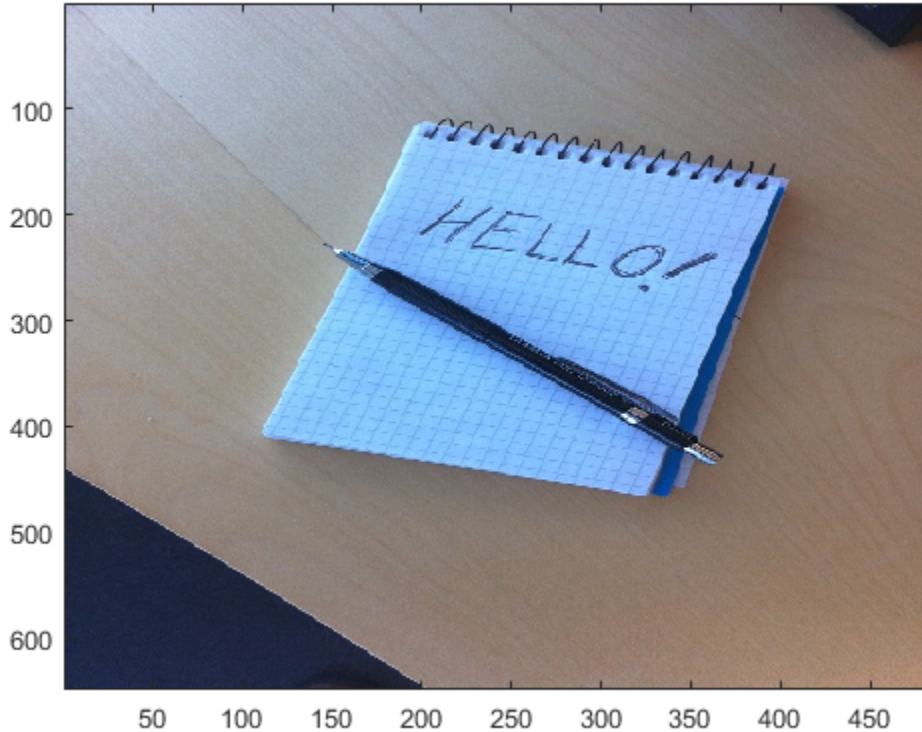
```
codegen sobel
```

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)))
```

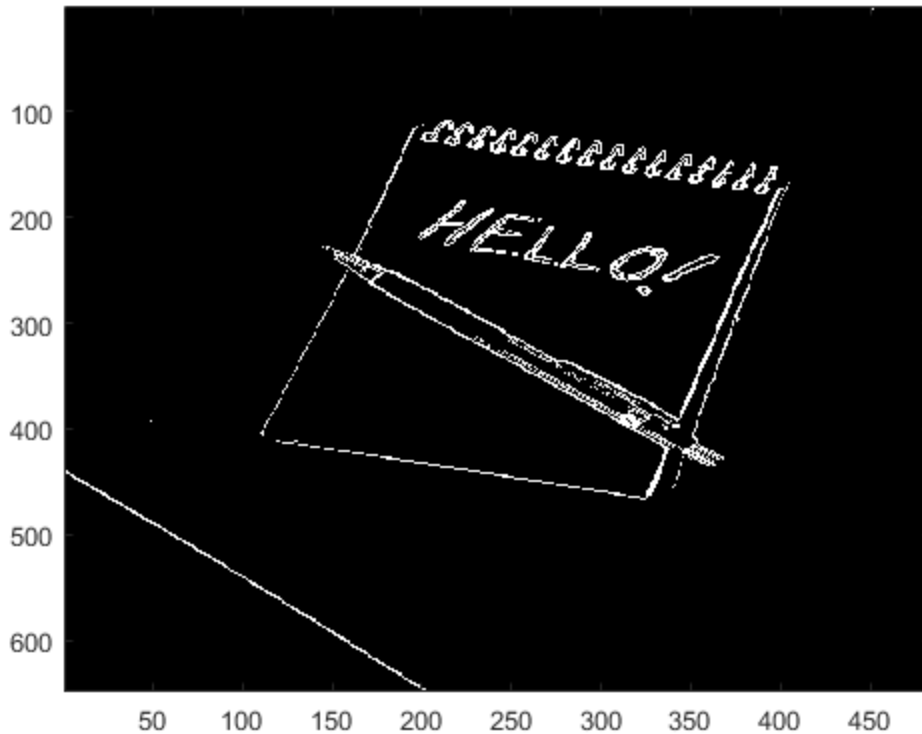
### Run the MEX Function (The Sobel Filter)

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 : 3.0
 * C/C++ source code generated on : 31-Jul-2015 04:32:18
 */

/* 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 emxArray_real_T *originalImage
```

```
* double threshold
* mxArray_uint8_T *edgeImage
* Return Type : void
*/
void sobel(const mxArray_real_T *originalImage, double threshold,
 mxArray_uint8_T *edgeImage)
{
 mxArray_real_T *H;
 mxArray_real_T *V;
 int b_H;
 int c_H;
 mxArray_init_real_T(&H, 2);
 mxArray_init_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];
 mxArray_ensure_capacity((mxArray__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];
 }

 mxArray_free_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];
 mxArray_ensure_capacity((mxArray__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);
 }

 mxArray_free_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 26-15
- “Control Run-Time Checks” on page 26-16

# 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 29-36
- “Inline Code” on page 29-9
- “Eliminate Redundant Copies of Function Inputs” on page 29-6

## Control Run-Time Checks

**In this section...**

“Types of Run-Time Checks” on page 26-16

“When to Disable Run-Time Checks” on page 26-16

“How to Disable Run-Time Checks” on page 26-17

### 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” on page 14-12.

### 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 Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to MEX.
- 3 Click **More Settings**.
- 4 On the **Speed** tab, clear **Ensure memory integrity**, **Enable responsiveness to CTRL+C** and **graphics refreshing**, or **Keep 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 26-19

“How parfor-Loops Improve Execution Speed” on page 26-20

“When to Use parfor-Loops” on page 26-20

“When Not to Use parfor-Loops” on page 26-20

“parfor-Loop Syntax” on page 26-21

“parfor Restrictions” on page 26-21

### 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 26-20.

---

**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” on page 26-2.

---

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” on page 26-21.

## 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 26-27. 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 26-29.

- 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:step:endVal)
parfor i=initVal:step:endVal
```

- 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” on page 24-22.
- 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 variables**

You cannot write to a global variable inside a `parfor`-loop.

- **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 26-27.

- **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 26-29.

### 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 26-27

“Sliced Variables” on page 26-28

“Broadcast Variables” on page 26-29

“Reduction Variables” on page 26-29

“Temporary Variables” on page 26-34

### 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
```

## 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 26-33                                |
| <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><br>Function <code>f</code> must be a user-defined function. | <code>X = f(expr, X)</code><br>See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 26-33 |

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) + ... + 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();         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

### Global or Persistent Declarations in parfor-Loop

The body of a `parfor`-loop cannot contain a `global` or `persistent` variable declaration.

### 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, use 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

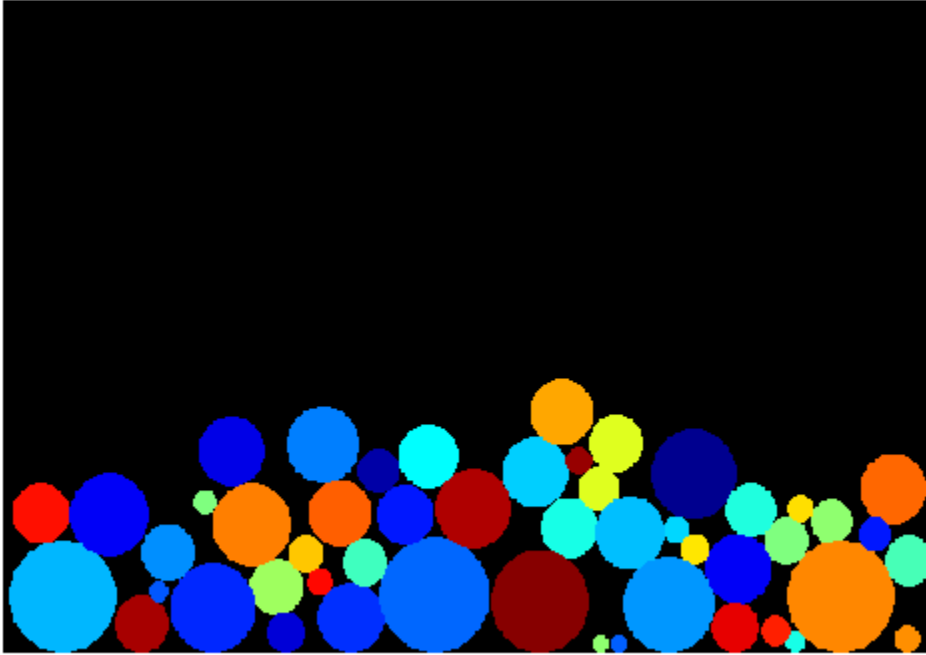
Run and time the original 'run\_balls' function followed by the generated MEX function.

```
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 ~2.8
```

### **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 27-2
- “Call External Functions Using `coder.ceval`” on page 27-6
- “Return Multiple Values from C Functions” on page 27-8
- “How MATLAB Coder Infers C/C++ Data Types” on page 27-9

## External Function Calls from Generated Code

### In this section...

- “Calling External Functions from Generated Code” on page 27-2
- “Why Call External Functions from Generated Code?” on page 27-2
- “How To Call External Functions” on page 27-2
- “Pass Arguments by Reference to External Functions” on page 27-3
- “Manipulate C Data” on page 27-4

### 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” on page 25-12.

## 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 27-6

“Best Practices for Calling External Code from Generated Code” on page 27-7

### 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 27-9.

---

**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 27-3).

- 4 Include the custom C/C++ functions in the build. See “Specify External File Locations” on page 25-12.
- 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 27-9).

- 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” on page 25-12.

## 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 27-9

“Mapping 64-Bit Integer Types to C/C++” on page 27-10

“Mapping Fixed-Point Types to C/C++” on page 27-11

“Mapping Arrays to C/C++” on page 27-11

“Mapping Complex Values to C/C++” on page 27-12

“Mapping Structures to C/C++ Structures” on page 27-13

“Mapping Strings to C/C++” on page 27-13

“Mapping Multiword Types to C/C++” on page 27-14

### 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 Types Used in Generated Code” on page 21-39.

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++” on page 27-10.                                                                                                                    |                                           |
| 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++” on page 27-10.                                                                                                                    |                                           |
| double          | real_T                                                                                                                                                                        | real_T *                                  |
| single          | real32_T                                                                                                                                                                      | real32_T *                                |
| char            | char_T                                                                                                                                                                        | char *                                    |
| logical         | boolean_T                                                                                                                                                                     | boolean_T *                               |
| fi              | numerictype 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++” on page 27-11. |                                           |
| struct          | The MATLAB Coder software translates structures to C/C++ types field-by-field. See “Mapping Structures to C/C++ Structures” on page 27-13 .                                   |                                           |
| complex         | See “Mapping Complex Values to C/C++” on page 27-12.                                                                                                                          |                                           |
| Multiword types | See “Mapping Multiword Types to C/C++” on page 27-14.                                                                                                                         |                                           |

## 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++” on page 27-14.

## 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++” on page 27-10. |                                           |
| <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++” on page 27-10. |                                           |
| <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 28-2
- “Encapsulating the Interface to External Code” on page 28-3
- “Best Practices for Using `coder.ExternalDependency`” on page 28-4
- “Encapsulate Interface to an External C Library” on page 28-6
- “Update Build Information from MATLAB code” on page 28-9
- “Call External Functions Encapsulated by `coder.ExternalDependency`” on page 28-10

## 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” on page 28-3
- “Specify External File Locations” on page 25-12
- “External Function Calls from Generated Code” on page 27-2

## 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” on page 28-6

### More About

- “Best Practices for Using `coder.ExternalDependency`” on page 28-4

## Best Practices for Using `coder.ExternalDependency`

### In this section...

“Terminate Code Generation for Unsupported External Dependency” on page 28-4

“Parameterize Methods for MATLAB and Generated Code” on page 28-4

“Parameterize `updateBuildInfo` for Multiple Platforms” on page 28-5

### 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” on page 28-6

## 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” on page 28-3
- “Build Information Object” on page 21-140
- “Build Information Methods” on page 21-140



## 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” on page 28-6

### More About

- “Encapsulating the Interface to External Code” on page 28-3

# Generate Efficient and Reusable Code

---

- “Optimization Strategies” on page 29-2
- “Modularize MATLAB Code” on page 29-5
- “Eliminate Redundant Copies of Function Inputs” on page 29-6
- “Inline Code” on page 29-9
- “Control Inlining” on page 29-11
- “Fold Function Calls into Constants” on page 29-14
- “Control Stack Space Usage” on page 29-16
- “Stack Allocation and Performance” on page 29-17
- “Dynamic Memory Allocation and Performance” on page 29-18
- “Minimize Dynamic Memory Allocation” on page 29-19
- “Provide Maximum Size for Variable-Size Arrays” on page 29-20
- “Disable Dynamic Memory Allocation During Code Generation” on page 29-26
- “Set Dynamic Memory Allocation Threshold” on page 29-27
- “Excluding Unused Paths from Generated Code” on page 29-29
- “Prevent Code Generation for Unused Execution Paths” on page 29-30
- “Generate Code with Parallel for-Loops (parfor)” on page 29-32
- “Minimize Redundant Operations in Loops” on page 29-34
- “Unroll for-Loops” on page 29-36
- “Support for Integer Overflow and Non-Finites” on page 29-39
- “Integrate Custom Code” on page 29-41
- “MATLAB Coder Optimizations in Generated Code” on page 29-47
- “Generate Reusable Code” on page 29-50
- “Reuse Large Arrays and Structures” on page 29-51
- “Variable Reuse in Generated Code” on page 29-52

## 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” on page 29-47.

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> )” on page 29-32 |
| You have variable-size arrays in your MATLAB code.                                                                                                       | “Minimize Dynamic Memory Allocation” on page 29-19                            |
| You have multiple variable-size arrays in your MATLAB code. You want dynamic memory allocation for larger arrays and static allocation for smaller ones. | “Set Dynamic Memory Allocation Threshold” on page 29-27                       |
| You want your generated function to be called by reference.                                                                                              | “Eliminate Redundant Copies of Function Inputs” on page 29-6                  |
| You are calling small functions in your MATLAB code.                                                                                                     | “Inline Code” on page 29-9                                                    |
| You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.                   | “Control Inlining” on page 29-11                                              |
| You do not want to generate code for expressions that contain constants only.                                                                            | “Fold Function Calls into Constants” on page 29-14                            |
| You have loop operations in your MATLAB code that do not depend on the loop index.                                                                       | “Minimize Redundant Operations in Loops” on page 29-34                        |

| Condition                                                                                                                                          | Action                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| You have integer operations in your MATLAB code. You know beforehand that integer overflow does not occur during execution of your generated code. | “Disable Support for Integer Overflow” on page 29-39   |
| You know beforehand that Infs and NaNs do not occur during execution of your generated code.                                                       | “Disable Support for Non-Finite Numbers” on page 29-40 |
| You have for-loops with few iterations.                                                                                                            | “Unroll for-Loops” on page 29-36                       |
| You already have legacy C/C++ code optimized for your target environment.                                                                          | “Integrate Custom Code” on page 29-41                  |

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” on page 29-30 |
| You want your generated function to be called by reference.                                                                                                            | “Eliminate Redundant Copies of Function Inputs” on page 29-6       |
| You have limited stack space for your generated code.                                                                                                                  | “Control Stack Space Usage” on page 29-16                          |
| You are calling small functions in your MATLAB code.                                                                                                                   | “Inline Code” on page 29-9                                         |
| You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.                                 | “Control Inlining” on page 29-11                                   |
| You do not want to generate code for expressions that contain constants only.                                                                                          | “Fold Function Calls into Constants” on page 29-14                 |
| You have loop operations in your MATLAB code that do not depend on the loop index.                                                                                     | “Minimize Redundant Operations in Loops” on page 29-34             |
| You have integer operations in your MATLAB code. You know beforehand that                                                                                              | “Disable Support for Integer Overflow” on page 29-39               |

| <b>Condition</b>                                                                                                                                                                                                                                                      | <b>Action</b>                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| integer overflow does not occur during execution of your generated code.                                                                                                                                                                                              |                                                        |
| You know beforehand that Inf-s and NaN-s does not occur during execution of your generated code.                                                                                                                                                                      | “Disable Support for Non-Finite Numbers” on page 29-40 |
| Your MATLAB code has variables that are large arrays or structures. The variable reuse optimization preserves your variable names. You want to see if the extra memory required to preserve the variable names of the large arrays or structures affects performance. | “Reuse Large Arrays and Structures” on page 29-51      |

## 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++ code for the individual sections. You can reuse the code for the individual sections 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, the generated code passes the variable by reference 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 that you rewrite function `foo` without the optimization:

```
function y = foo2(A, B) %#codegen
y = A * B;
end
```

The generated code 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
```

The reference parameter optimization does not apply to constant inputs. If the same variable is an input and an output, and the input is constant, the code generation software treats the output as a separate variable. For example, consider the function `foo`:

```
function A = foo(A, B) %#codegen
A = A * B;
end
```

Generate code in which `A` has a constant value 2.

```
codegen -config:lib foo -args {coder.Constant(2) 3} -report
```

The generated code defines the constant A and returns the value of the output.

```
...
#define A (2.0)
...
double foo(double B)
{
 return A * B;
}
...
```

### **Related Examples**

- “Pass Structure Arguments by Reference or by Value” on page 8-13

## Inline Code

MATLAB uses internal heuristics to determine whether 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 29-9

“Use Inlining in Control Flow Statements” on page 29-9

### 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” on page 29-11

# Control Inlining

Restrict inlining when:

- The size of generated code exceeds desired limits due to excessive inlining of functions. Suppose that you include the statement, `coder.inline('always')`, inside a certain function. You then call that function at many 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 29-11

“Control Size of Functions After Inlining” on page 29-12

“Control Stack Size Limit on Inlined Functions” on page 29-12

## Control Size of Functions Inlined

You can use the MATLAB Coder app or the command-line interface to control the maximum size of functions that can be inlined. 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.

- Using the app, 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 use the MATLAB Coder app or the command-line interface to control the maximum size of functions after inlining. 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.

- Using the app, 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 a certain value of `InlineStackLimit` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

You can use the MATLAB Coder app or the command-line interface to control the stack size limit on inlined functions.

- Using the app, 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, `InlineStackLimit`, to the maximum size that you want.

```
cfg = coder.config('lib');
cfg.InlineStackLimit = 2000;
```

Generate code using this configuration object.

## **Related Examples**

- “Inline Code” on page 29-9

## 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 that the generated code uses. Set the maximum stack usage when:

- You have limited stack space, for instance, in embedded targets.
- Your C compiler reports a run-time stack overflow.

The value of the property, `StackUsageMax`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that a certain value of `StackUsageMax` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

### Control Stack Space Usage Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
- 3 Click **More Settings**.
- 4 On the **Memory** tab, set **Stack usage max** to the value that you want.

### Control Stack Space Usage 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 Set the property, `StackUsageMax`, to the value that you want.

```
cfg.StackUsageMax=400000;
```

### More About

- “Stack Allocation and Performance” on page 29-17

## 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” on page 29-16.

## 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” on page 7-4.

### 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 can 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 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 29-19.

## Minimize Dynamic Memory Allocation

When possible, minimize dynamic memory allocation because 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.

If you know the maximum size of a variable-size array, you can avoid dynamic memory allocation. Follow these steps:

- 1 “Provide Maximum Size for Variable-Size Arrays” on page 29-20.
- 2 Depending on your requirements, do one of the following:
  - “Disable Dynamic Memory Allocation During Code Generation” on page 29-26.
  - “Set Dynamic Memory Allocation Threshold” on page 29-27

---

**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” on page 29-18

## 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.ysize` 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.ysize`:

```
function out=ExpandArray() % Expand an array by five elements
 out = zeros(1,5);
 coder.ysize('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. varsize` with explicit upper bounds. This example shows how to use `coder. varsize` with explicit upper bounds:

### Restrict Concatenations Using `coder. varsize` 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. varsize('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 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. varsize` 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.varsize('avg',[1 8]);
```

with:

```
coder.varsize('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.varsize('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” on page 29-19
- “Disable Dynamic Memory Allocation During Code Generation” on page 29-26
- “Set Dynamic Memory Allocation Threshold” on page 29-27

## **More About**

- “Dynamic Memory Allocation and Performance” on page 29-18

## Disable Dynamic Memory Allocation During Code Generation

To disable dynamic memory allocation using the MATLAB Coder app:

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Memory** tab, under **Variable Sizing Support**, set **Dynamic memory allocation** to **Never**.

To disable dynamic memory allocation at 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';
```

If a variable-size array in the MATLAB code does not have a maximum upper bound, disabling dynamic memory allocation leads to a code generation error. Therefore, you can identify variable-size arrays in your MATLAB code that do not have a maximum upper bound. These arrays are the arrays that are dynamically allocated in the generated code.

### Related Examples

- “Minimize Dynamic Memory Allocation” on page 29-19
- “Provide Maximum Size for Variable-Size Arrays” on page 29-20
- “Set Dynamic Memory Allocation Threshold” on page 29-27

### More About

- “Dynamic Memory Allocation and Performance” on page 29-18


## Set Dynamic Memory Allocation Threshold

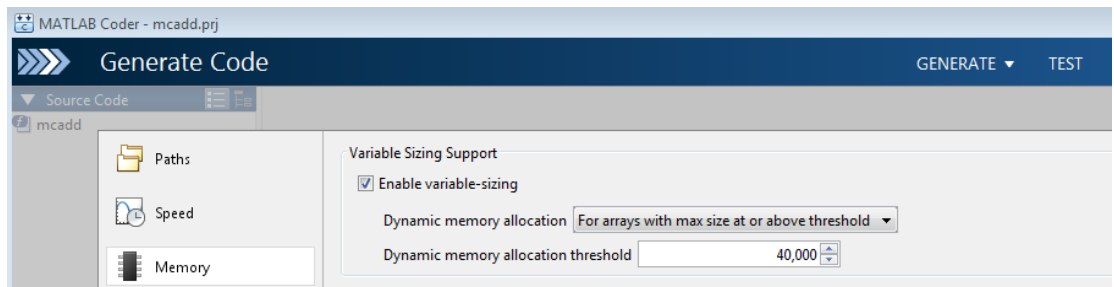
This example shows how to specify a dynamic memory allocation threshold for variable-size arrays. Dynamic memory allocation optimizes storage requirements for variable-size arrays, but causes slower execution of generated code. Instead of disabling dynamic memory allocation for all variable-size arrays, you can disable dynamic memory allocation for arrays less than a certain size.

Specify this threshold when you want to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, static memory allocation can speed up generated code. Static memory allocation can lead to unused storage space. However, you can decide that the unused storage space is not a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, when you use dynamic memory allocation, you can significantly reduce storage requirements.

### Set Dynamic Memory Allocation Threshold Using the MATLAB Coder App

- 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
- 2 Click **More Settings**.
- 3 On the **Memory** tab, select the **Enable variable-sizing** check box.
- 4 Set **Dynamic memory allocation** to **For arrays with max size at or above threshold**.
- 5 Set **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 a certain value of `DynamicMemoryAllocationThreshold` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

## Set Dynamic Memory Allocation Threshold 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 Set `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 a certain value of `DynamicMemoryAllocationThreshold` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

## Related Examples

- “Minimize Dynamic Memory Allocation” on page 29-19
- “Provide Maximum Size for Variable-Size Arrays” on page 29-20
- “Disable Dynamic Memory Allocation During Code Generation” on page 29-26

## More About

- “Dynamic Memory Allocation and Performance” on page 29-18

## 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/if` 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 the control-flow variable chooses.

### Related Examples

- “Prevent Code Generation for Unused Execution Paths” on page 29-30

## Prevent Code Generation for Unused Execution Paths

### In this section...

“Prevent Code Generation When Local Variable Controls Flow” on page 29-30

“Prevent Code Generation When Input Variable Controls Flow” on page 29-31

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 29-30.
- 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 29-31.

### 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” on page 29-29

## 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`)” on page 26-19
- “Classification of Variables in `parfor`-Loops” on page 26-27

- “Reduction Assignments in parfor-Loops” on page 26-26

## 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;
% 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 dounroll 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 dounroll 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


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.
- An operation generates non-finite values (`inf` and `NaN`). The supporting code is contained in the files `rt_nonfinite.c`, `rtGetInf.c`, and `rtGetNaN.c` (with corresponding header files).

If you know that these situations do not occur, you can suppress generation of the supporting code. You therefore reduce the size of the generated code and increase its speed. However, if one of these situations occurs, it is possible that the generated code will not match the behavior of the original MATLAB code.

### Disable Support for Integer Overflow

You can use the MATLAB Coder app or the command-line interface to disable support for integer overflow. When you disable this support, the overflow behavior of your generated code depends on your target C compiler. Most C compilers wrap on overflow.


- Using the app:
  - 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
  - 2 Click **More Settings**.
  - 3 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-Finite Numbers

You can use the MATLAB Coder app or the command-line interface to disable support for non-finite numbers(`inf` and `NaN`).

- Using the app:
  - 1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow .
  - 2 Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
  - 3 Click **More Settings**.
  - 4 On the **Speed** tab, clear the **Support non-finite numbers** check box.
- 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® CUDA® 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
 end
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 `cublasSgemm` 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” on page 28-6

### **More About**

- “Encapsulating the Interface to External Code” on page 28-3



## MATLAB Coder Optimizations in Generated Code

### In this section...

“Constant Folding” on page 29-47

“Loop Fusion” on page 29-48

“Successive Matrix Operations Combined” on page 29-48

“Unreachable Code Elimination” on page 29-49

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” on page 29-14.

## 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;
```

- Using the app, 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” on page 14-2.

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.

## Reuse Large Arrays and Structures

To reduce memory usage, when possible, variables share names and memory in the generated code. By default, for readability, this variable reuse optimization preserves your name for a variable. It does not reuse your variable name for another variable or reuse another variable name for your variable. If your code has large arrays or structures, the extra memory required to preserve your variable names can affect performance. In this case, to reduce memory usage, specify that the variable reuse optimization does not have to preserve variable names. The optimization can, therefore, reuse any variables that meet the requirements for variable reuse.

- Using the MATLAB Coder app, in the project settings dialog box, on the **All Settings** tab, set **Preserve variable names** to **None**.
- Using the command-line interface:

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

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

- 2 Set the `PreserveVariableNames` property to `None`.

```
cfg.PreserveVariableNames = 'None';
```

### More About

- “Variable Reuse in Generated Code” on page 29-52
- “Optimization Strategies” on page 29-2

## Variable Reuse in Generated Code

To reduce memory usage, MATLAB Coder can reuse the name and memory of one variable for another variable when:

- Both variables have the same memory requirements.
- Memory access for one variable does not interfere with memory access for the other.

By default, this variable reuse optimization preserves your variable names in the generated code. It does not reuse your variable name for another variable or reuse another variable name for your variable. For example, for code such as:

```
if (s>0)
 myvar1 = 0;
 ...
else
 myvar2 = 0;
 ...
end
```

the generated code might look like:

```
if (s > 0.0) {
 myvar1 = 0.0;
 ...
} else {
 myvar2 = 0.0;
 ...
}
```

You can configure the variable reuse optimization so that it does not preserve your variable names. For an example, see “Reuse Large Arrays and Structures” on page 29-51. When the optimization does not preserve your variable names, the generated code might look like:

```
if (s > 0.0) {
 myvar2 = 0.0;
 ...
} else {
 myvar2 = 0.0;
 ...
}
```

In this case, the optimization uses `myvar2` for `myvar1`.

# Generating Reentrant C Code from MATLAB Code

---

- “Generate Reentrant C Code from MATLAB Code” on page 30-2
- “Reentrant Code” on page 30-10
- “Specify Generation of Reentrant Code” on page 30-12
- “Generated Code API” on page 30-14
- “Call Reentrant Code in a Single-Thread Environment” on page 30-15
- “Call Reentrant Code in a Multithreaded Environment” on page 30-16
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 30-17
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 30-22
- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 30-27

## Generate Reentrant C Code from MATLAB Code

**In this section...**

“About This Tutorial” on page 30-2  
“Copying Files Locally” on page 30-3  
“About the Example” on page 30-4  
“Providing a C main Function” on page 30-5  
“Configuring Build Parameters” on page 30-6  
“Generating the C Code” on page 30-7  
“Viewing the Generated C Code” on page 30-7  
“Running the Code” on page 30-8  
“Key Points to Remember” on page 30-8  
“Learn More” on page 30-9

### About This Tutorial

#### Learning Objectives

This tutorial shows you how to:

- Generate reentrant code from MATLAB code that does not use persistent or global data.
- Automatically generate C code from your MATLAB code.
- Define function input properties at the command line.
- Specify code generation properties.
- Generate a code generation report that you can use to view and debug your MATLAB code.

---

**Note:** This example runs on Windows only.

---

#### Prerequisites

To complete this example, install the following products:



- MATLAB
- MATLAB Coder
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). MATLAB Coder locates and uses a supported installed compiler. For the current list of supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/) on the MathWorks website.

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

### Required Files

| Type            | Name                      | Description                                                                                                               |
|-----------------|---------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Function code   | <code>matrix_exp.m</code> | MATLAB function that computes matrix exponential of the input matrix using Taylor series and returns the computed output. |
| C main function | <code>main.c</code>       | Calls the reentrant code.                                                                                                 |

### Copying Files Locally

Copy the tutorial files to a local working folder.

- 1 Create a local working folder, for example, `c:\coder\work`.
- 2 Change to the `matlabroot\help\toolbox\coder\examples` folder. At the MATLAB command prompt, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

- 3 Copy the `reentrant_win` folder to your local working folder.

Your work folder now contains the files for the tutorial.

- 4 Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command prompt, enter:

```
cd work
```

`work` is the full path of the work folder containing your files.

## About the Example

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a simple, multithreaded example that does not use persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

### Contents of `matrix_exp.m`

```
function Y = matrix_exp(X) %#codegen
%
% The function matrix_exp computes matrix exponential of
% the input matrix using Taylor series and returns the
% computed output.
E = zeros(size(X));
F = eye(size(X));
k = 1;
while norm(E+F-E,1) > 0
 E = E + F;
 F = X*F/k;
 k = k+1;
end
Y = E;
```

When you generate reusable, reentrant code, MATLAB Coder supports dynamic allocation of:

- Function variables that are too large for the stack
- Persistent variables
- Global variables

MATLAB Coder generates a header file, `primary_function_name_types.h`, that you must include when using the generated code. This header file contains the following structures:

- `primary_function_nameStackData`

Contains the user allocated memory. Pass a pointer to this structure as the first parameter to functions that use it:

- Directly (the function uses a field in the structure)
- Indirectly (the function passes the structure to a called function)

If the algorithm uses persistent or global data, the `primary_function_nameStackData` structure also contains a pointer to the

*primary\_function\_name*PersistentData structure. If you include this pointer, you have to pass only one parameter to each calling function.

- *primary\_function\_name*PersistentData

If your algorithm uses persistent or global variables, MATLAB Coder provides a separate structure for them. The memory allocation structure contains a pointer to this persistent data structure. Because you have a separate structure for persistent and global variables, you can allocate memory for these variables once and share them with all threads. However, if the threads do not communicate, you can allocate memory for these variables per thread or per application.

## Providing a C main Function

To call the reentrant code, provide a main function that:

- Includes the generated header file `matrix_exp.h`. This file includes the generated header file, `matrix_exp_types.h`.
- For each thread, allocates memory for stack data.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions” on page 25-7.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees up the for stack data memory.

### Contents of `main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
 real_T in[NUMELEMENTS];
 real_T out[NUMELEMENTS];
 matrix_expStackData* spillData;
} IODATA;

/* The thread_function calls the matrix_exp function written in MATLAB */
DWORD WINAPI thread_function(PVOID dummyPtr) {
 IODATA *myIOData = (IODATA*)dummyPtr;
 matrix_exp_initialize();
 matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
}
```

```
 matrix_exp_terminate();
 return 0;
}

void main() {
 HANDLE thread1, thread2;
 IODATA data1;
 IODATA data2;
 int32_T i;

 /*Initializing data for passing to the 2 threads*/
 matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
 matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

 data1.spillData = sd1;
 data2.spillData = sd2;

 for (i=0;i<NUMELEMENTS;i++) {
 data1.in[i] = 1;
 data1.out[i] = 0;
 data2.in[i] = 1.1;
 data2.out[i] = 0;
 }

 /*Initializing the 2 threads and passing data to the thread functions*/
 printf("Starting thread 1...\n");
 thread1 = CreateThread(NULL, 0, thread_function, (PVOID) &data1, 0, NULL);
 if (thread1 == NULL){
 perror("Thread 1 creation failed.");
 exit(EXIT_FAILURE);
 }

 printf("Starting thread 2...\n");
 thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
 if (thread2 == NULL){
 perror("Thread 2 creation failed.");
 exit(EXIT_FAILURE);
 }

 /*Wait for both the threads to finish execution*/
 if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
 perror("Thread 1 join failed.");
 exit(EXIT_FAILURE);
 }

 if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
 perror("Thread 2 join failed.");
 exit(EXIT_FAILURE);
 }

 free(sd1);
 free(sd2);

 printf("Finished Execution!\n");
 exit(EXIT_SUCCESS);
}
```

## Configuring Build Parameters

You can enable generation of reentrant code using a code generation configuration object.

- 1 Create a configuration object.
 

```
cfg = coder.config('exe');
```
- 2 Enable reentrant code generation.
 

```
cfg.MultiInstanceCode = true;
```

## Generating the C Code

Call the `codegen` function to generate C code, with the following options:

- `-config` to pass in the code generation configuration object `cfg`.
- `main.c` to include this file in the compilation.
- `-report` to create a code generation report.
- `-args` to specify the class, size, and complexity of input arguments using example data.

```
codegen -config cfg main.c -report matrix_exp.m -args ones(160,160)
```

`codegen` generates a C executable, `matrix_exp.exe`, in the current folder and C code in the `/codegen/exe/matrix_exp` subfolder. Because you selected report generation, `codegen` provides a link to the report.

## Viewing the Generated C Code

`codegen` generates a header file `matrix_exp_types.h`, which defines the `matrix_expStackData` global structure. This structure contains local variables that are too large to fit on the stack.

To view this header file:

- 1 Click the **View report** link to open the code generation report.
- 2 In the report, click the **C code** tab.
- 3 On this tab, click the link to `matrix_exp_types.h`.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */
```

```
#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
 struct {
 double F[25600];
 double Y[25600];
 double X[25600];
 } f0;
} matrix_expStackData;

#endif /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

## Running the Code

Verify that the example is running on Windows platforms and call the code.

```
% This example can only be run on Windows platforms
if ~ispc
 error('This example requires Windows-specific libraries and can only be run on Windows.');
```

end  
system('matrix\_exp.exe')

The executable runs and reports completion.

## Key Points to Remember

- Create a `main` function that:
  - Includes the generated header file, `primary_function_name_types.h`. This file defines the `primary_function_nameStackData` global structure. This structure contains local variables that are too large to fit on the stack.
  - For each thread, allocates memory for stack data.
  - Calls `primary_function_name_initialize`.
  - Calls `primary_function_name`.

- Calls *primary\_function\_name\_terminate*.
- Frees the stack data memory.
- Use the `-config` option to pass the code generation configuration object to the `codegen` function.
- Use the `-args` option to specify input parameters at the command line.
- Use the `-report` option to create a code generation report.

## Learn More

| To                                                            | See                                                                                     |
|---------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Learn more about the generated code API                       | “Generated Code API” on page 30-14                                                      |
| Call reentrant code without persistent or global data on UNIX | “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 30-17       |
| Call reentrant code with persistent data on Windows           | “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 30-22 |
| Call reentrant code with persistent data on UNIX              | “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 30-27    |

## Reentrant Code

Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Operating systems and other system software that use multithreading to handle concurrent events use reentrant code. In a concurrent environment, multiple threads or processes can attempt to read and write static data simultaneously. Therefore, sharing code that uses persistent or static data is difficult. Reentrant code does not contain static data. Calling programs maintain their state variables and pass them into the function. Therefore, any number of threads or processes can share one copy of a reentrant routine.

Generate reentrant code when you want to:

- Deploy your code in multithreaded environments.
- Use an algorithm with persistent data belonging to different processes or threads.
- Compile code that uses function variables that are too large to fit on the stack.

If you do not specify reentrant code, MATLAB Coder generates code that uses statically allocated memory for:

- Function variables that are too large to fit on the stack
- Global variables
- Persistent variables

If the generated code uses static memory allocation for these variables, you cannot deploy the generated code in environments that require reentrant code. If you cannot adjust the static memory allocation size, the generated code can result in static memory size overflow.

When you generate reentrant code, MATLAB Coder creates input data structures for:

- Function variables that are too large to fit on the stack
- Persistent variables
- Global variables

You can then dynamically allocate memory for these input structures. The use of dynamic memory allocation means that you can deploy the code in reentrant environments.



## Related Examples

- “Specify Generation of Reentrant Code” on page 30-12
- “Generate Reentrant C Code from MATLAB Code” on page 30-2
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 30-17
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 30-22
- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 30-27


## Specify Generation of Reentrant Code

### In this section...

“Specify Generation of Reentrant Code Using the MATLAB Coder App” on page 30-12

“Specify Generation of Reentrant Code Using the Command-Line Interface” on page 30-12

### Specify Generation of Reentrant Code Using the MATLAB Coder App

- 1 On the **Generate Code** page, click the **Generate** arrow .
- 2 Set **Build type** to one of the following:
  - Source Code
  - Static Library (.lib)
  - Dynamic Library (.dll)
  - Executable (.exe)
- 3 Click **More Settings**.
- 4 On the **Memory** tab, select the **Generate re-entrant code** check box.

### Specify Generation of Reentrant Code Using the Command-Line Interface

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

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

- 2 Set the `MultiInstanceCode` property to `true`. For example:

```
cfg.MultiInstanceCode = true;
```

### Related Examples

- “Generate Reentrant C Code from MATLAB Code” on page 30-2
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 30-17
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 30-22

- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 30-27

## **More About**

- “Reentrant Code” on page 30-10

## Generated Code API

When you generate reusable code, MATLAB Coder supports dynamic allocation of:

- Function variables that are too large for the stack
- Persistent variables
- Global variables

It generates a header file, *primary\_function\_name\_types.h*, that you must include when using the generated code. This header file contains the following structures:

- *primary\_function\_nameStackData*

This structure contains the user-allocated memory. You must pass a pointer to this structure as the first parameter to all functions that use it:

- Directly, because the function uses a field in the structure.
- Indirectly, because the function passes the structure to a called function.

If the algorithm uses persistent or global data, the *primary\_function\_nameStackData* structure also contains a pointer to the *primary\_function\_namePersistentData* structure. If you include this pointer, you only have to pass one parameter to each calling function.

- *primary\_function\_namePersistentData*

If your algorithm uses persistent or global variables, MATLAB Coder provides a separate structure for them. The memory allocation structure contains a pointer to this structure. Because you have a separate structure for persistent and global variables, you can allocate memory for these variables once and share them with all threads. However, if there is no communication between threads, you can choose to allocate memory for these variables per thread or per application.

For more information on using these global structures, see “Multithreaded Examples” on page 30-16.

## Call Reentrant Code in a Single-Thread Environment

To call reentrant code in a single-thread environment, create a `main` function that:

- Includes the header file `primary_function_name.h`.
- Allocates memory for the global memory allocation structure `primary_function_nameStackData`.
- If the algorithm uses persistent or global data, allocates memory for the global structure `primary_function_namePersistentData`.
- Calls these functions:
  - `primary_function_name_initialize`.
  - `primary_function_name`.
  - `primary_function_name_terminate`.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder generates two housekeeping functions. Call these housekeeping functions in the code that calls the generated C/C++ function. For more information, see “Calling Initialize and Terminate Functions” on page 25-7.

- Frees the memory used for global structures.

## Call Reentrant Code in a Multithreaded Environment

To call reentrant code, create a `main` function that:

- Includes the header file `primary_function_name.h`.
- For each thread, allocates memory for the global memory allocation structure `primary_function_nameStackData`.
- If the algorithm uses persistent or global data, allocates memory for the global structure `primary_function_namePersistentData`. If the threads communicate, allocate this memory once for the application. Otherwise, you can choose to allocate memory per thread or per application.
- Contains a thread function that calls these functions:
  - `primary_function_name_initialize`.
  - `primary_function_name`.
  - `primary_function_name_terminate`.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder generates two housekeeping functions. Call these functions in the code that calls the generated C/C++ function. For more information, see “Calling Initialize and Terminate Functions” on page 25-7.

- Initializes each thread and passes in a pointer to the memory allocation structure as the first parameter to the thread function.
- Frees up the memory used for global structures.

### Multithreaded Examples

| Type of Reentrant Code                          | Platform | Reference                                                                               |
|-------------------------------------------------|----------|-----------------------------------------------------------------------------------------|
| Multithreaded without persistent or global data | Windows  | “Generate Reentrant C Code from MATLAB Code” on page 30-2                               |
|                                                 | UNIX     | “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 30-17       |
| Multithreaded with persistent or global data    | Windows  | “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 30-22 |
|                                                 | UNIX     | “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 30-27    |

## Call Reentrant Code with No Persistent or Global Data (UNIX Only)

### In this section...

- “Provide a Main Function” on page 30-17
- “Generate Reentrant C Code” on page 30-19
- “Examine the Generated Code” on page 30-20
- “Run the Code” on page 30-21

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a simple multithreaded example that uses no persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

### Provide a Main Function

To call the reentrant code, provide a main function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions” on page 25-7.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
 real_T in[NUMELEMENTS];
 real_T out[NUMELEMENTS];
 matrix_expStackData* spillData;
} IODATA;
```

```
/* The thread_function calls the matrix_exp function written in MATLAB */
void *thread_function(void *dummyPtr) {
 IODATA *myIOData = (IODATA*)dummyPtr;
 matrix_exp_initialize();
 matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
 matrix_exp_terminate();
}

int main() {
 pthread_t thread1, thread2;
 int iret1, iret2;
 IODATA data1;
 IODATA data2;
 int32_T i;

 /*Initializing data for passing to the 2 threads*/
 matrix_expStackData* sd1=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
 matrix_expStackData* sd2=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

 data1.spillData = sd1;
 data2.spillData = sd2;

 for (i=0;i<NUMELEMENTS;i++) {
 data1.in[i] = 1;
 data1.out[i] = 0;
 data2.in[i] = 1.1;
 data2.out[i] = 0;
 }

 /*Initializing the 2 threads and passing required data to the thread functions*/
 printf("Starting thread 1...\n");
 iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
 if (iret1 != 0){
 perror("Thread 1 creation failed.");
 exit(EXIT_FAILURE);
 }

 printf("Starting thread 2...\n");
 iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
 if (iret2 != 0){
 perror("Thread 2 creation failed.");
 exit(EXIT_FAILURE);
 }

 /*Wait for both the threads to finish execution*/
 iret1 = pthread_join(thread1, NULL);
 if (iret1 != 0){
 perror("Thread 1 join failed.");
 exit(EXIT_FAILURE);
 }

 iret2 = pthread_join(thread2, NULL);
 if (iret2 != 0){
 perror("Thread 2 join failed.");
 }
}
```



```

 exit(EXIT_FAILURE);
}

 free(sd1);
 free(sd2);

 printf("Finished Execution!\n");
 exit(EXIT_SUCCESS);
}

```

## Generate Reentrant C Code

To generate code, run the following script at the MATLAB command prompt.

```

% This example can only be run on Unix platforms
if ~isunix
 error('This example requires pthread libraries and can only be run on Unix.');
```

end

```

% Setting the options for the Config object

% Create a code gen configuration object
cfg = coder.config('exe');
```

% Enable reentrant code generation

```

cfg.MultiInstanceCode = true;
```

% Set the post code generation command to be the 'setbuildargs' function

```

cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';
```

% Compiling

```

codegen -config cfg main.c matrix_exp.m -report -args ones(160,160)

```

This script:

- Generates an error message if the example is not running on a UNIX platform.
- Creates a code configuration object for generation of an executable.
- Enables the `MultiInstanceCode` option to generate reusable, reentrant code.
- Uses the `PostCodeGenCommand` option to set the post code generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

```

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'};

```

```
addLinkFlags(buildInfo, linkFlags);
```

For more information, see “Customize the Post-Code-Generation Build Process” on page 21-139.

- Invokes `codegen` with the following options:
  - `-config` to pass in the code generation configuration object `cfg`.
  - `main.c` to include this file in the compilation.
  - `-report` to create a code generation report.
  - `-args` to specify an example input with class, size, and complexity.

## Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines the `matrix_expStackData` global structure. This structure contains local variables that are too large to fit on the stack.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
 struct {
 double F[25600];
 double Y[25600];
 double X[25600];
 } f0;
} matrix_expStackData;

#endif
/*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

## Run the Code

Call the code using the command:

```
system('./matrix_exp')
```

The executable runs and reports completion.

## Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)

### In this section...

“MATLAB Code for This Example” on page 30-22

“Provide a Main Function” on page 30-23

“Generate Reentrant C Code” on page 30-25

“Examine the Generated Code” on page 30-25

“Run the Code” on page 30-26

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

### MATLAB Code for This Example

```
function [Y,numTimes] = matrix_exp(X) %#codegen
%
% The function matrix_exp computes matrix exponential
% of the input matrix using Taylor series and returns
% the computed output. It also returns the number of
% times this function has been called.
%
persistent count;
if isempty(count)
 count = 0;
end
count = count+1;

E = zeros(size(X));
F = eye(size(X));
k = 1;
while norm(E+F-E,1) > 0
 E = E + F;
 F = X*F/k;
 k = k+1;
end
Y = E ;

numTimes = count;
```

## Provide a Main Function

To call reentrant code that uses persistent data, provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions” on page 25-7.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
 real_T in[NUMELEMENTS];
 real_T out[NUMELEMENTS];
 real_T numTimes;
 matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
DWORD WINAPI thread_function(PVOID dummyPtr) {
 IODATA *myIOData = (IODATA*)dummyPtr;
 matrix_exp_initialize(myIOData->spillData);
 matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData->numTimes);
 printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
 matrix_exp_terminate();
 return 0;
}

void main() {
 HANDLE thread1, thread2;
 IODATA data1;
 IODATA data2;
 int32_T i;
```

```
/*Initializing data for passing to the 2 threads*/
matrix_expPersistentData* pd1 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
matrix_expPersistentData* pd2 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

sd1->pd = pd1;
sd2->pd = pd2;
data1.spillData = sd1;
data2.spillData = sd2;

for (i=0;i<NUMELEMENTS;i++) {
 data1.in[i] = 1;
 data1.out[i] = 0;
 data2.in[i] = 1.1;
 data2.out[i] = 0;
}

data1.numTimes = 0;
data2.numTimes = 0;

/*Initializing the 2 threads and passing required data to the thread functions*/
printf("Starting thread 1...\n");
thread1 = CreateThread(NULL, 0, thread_function, (PVOID) &data1, 0, NULL);
if (thread1 == NULL){
 perror("Thread 1 creation failed.");
 exit(EXIT_FAILURE);
}

printf("Starting thread 2...\n");
thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
if (thread2 == NULL){
 perror("Thread 2 creation failed.");
 exit(EXIT_FAILURE);
}

/*Wait for both the threads to finish execution*/
if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
 perror("Thread 1 join failed.");
 exit(EXIT_FAILURE);
}

if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
 perror("Thread 2 join failed.");
 exit(EXIT_FAILURE);
}

free(sd1);
free(sd2);
free(pd1);
free(pd2);

printf("Finished Execution!\n");
exit(EXIT_SUCCESS);
```

```
}
```

## Generate Reentrant C Code

Run the following script at the MATLAB command prompt to generate code.

```
% This example can only be run on Windows platforms
if ~ispc
 error...
 ('This example requires Windows-specific libraries and can only be run on Windows.');
```

```
end

% Setting the options for the Config object
% Create a code gen configuration object
cfg = coder.config('exe');
```

```
% Enable reentrant code generation
cfg.MultiInstanceCode = true;
```

```
% Compiling
codegen -config cfg main.c -report matrix_exp.m -args ones(160,160)
```

This script:

- Generates an error message if the example is not running on a Windows platform.
- Creates a code generation configuration object for generation of an executable.
- Enables the `MultiInstanceCode` option to generate reusable, reentrant code.
- Invokes `codegen` with the following options:
  - `-config` to pass in the code generation configuration object `cfg`.
  - `main.c` to include this file in the compilation.
  - `-report` to create a code generation report.
  - `-args` to specify an example input with class, size, and complexity.

## Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, that defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expPersistentData
#define typedef_matrix_expPersistentData

typedef struct {
 double count;
} matrix_expPersistentData;

#endif /*typedef_matrix_expPersistentData*/

#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
 struct {
 double F[25600];
 double Y[25600];
 double X[25600];
 } f0;

 matrix_expPersistentData *pd;
} matrix_expStackData;

#endif /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

## Run the Code

Call the code using the command:

```
system('matrix_exp.exe')
The executable runs and reports completion.
```



## Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)

### In this section...

“MATLAB Code for This Example” on page 30-27

“Provide a Main Function” on page 30-28

“Generate Reentrant C Code” on page 30-30

“Examine the Generated Code” on page 30-31

“Run the Code” on page 30-32

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

### MATLAB Code for This Example

```
function [Y,numTimes] = matrix_exp(X) %#codegen
%
% The function matrix_exp computes matrix exponential
% of the input matrix using Taylor series and returns
% the computed output. It also returns the number of
% times this function has been called.
%
persistent count;
if isempty(count)
 count = 0;
end
count = count+1;

E = zeros(size(X));
F = eye(size(X));
k = 1;
while norm(E+F-E,1) > 0
 E = E + F;
 F = X*F/k;
 k = k+1;
end
Y = E ;

numTimes = count;
```

## Provide a Main Function

To call reentrant code that uses persistent data, provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions” on page 25-7.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
 real_T in[NUMELEMENTS];
 real_T out[NUMELEMENTS];
 real_T numTimes;
 matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
void *thread_function(void *dummyPtr) {
 IODATA *myIOData = (IODATA*)dummyPtr;
 matrix_exp_initialize(myIOData->spillData);
 matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData->numTimes);
 printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
 matrix_exp_terminate();
}

int main() {
 pthread_t thread1, thread2;
 int iret1, iret2;
 IODATA data1;
 IODATA data2;
 int32_T i;
```

```

/*Initializing data for passing to the 2 threads*/
matrix_expPersistentData* pd1 =
 (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
matrix_expPersistentData* pd2 =
 (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

sd1->pd = pd1;
sd2->pd = pd2;
data1.spillData = sd1;
data2.spillData = sd2;

for (i=0;i<NUMELEMENTS;i++) {
 data1.in[i] = 1;
 data1.out[i] = 0;
 data2.in[i] = 1.1;
 data2.out[i] = 0;
}

data1.numTimes = 0;
data2.numTimes = 0;

/*Initializing the 2 threads and passing required data to the thread functions*/
printf("Starting thread 1...\n");
iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
if (iret1 != 0){
 perror("Thread 1 creation failed.");
exit(EXIT_FAILURE);
}

printf("Starting thread 2...\n");
iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
if (iret2 != 0){
 perror("Thread 2 creation failed.");
 exit(EXIT_FAILURE);
}

/*Wait for both the threads to finish execution*/
iret1 = pthread_join(thread1, NULL);
if (iret1 != 0){
 perror("Thread 1 join failed.");
exit(EXIT_FAILURE);
}

iret2 = pthread_join(thread2, NULL);
if (iret2 != 0){
 perror("Thread 2 join failed.");
exit(EXIT_FAILURE);
}

free(sd1);
free(sd2);
free(pd1);
free(pd2);

```

```
printf("Finished Execution!\n");
return(0);
}
```

## Generate Reentrant C Code

To generate code, run the following script at the MATLAB command prompt.

```
% This example can only be run on Unix platforms
if ~isunix
 error('This example requires pthread libraries and can only be run on Unix.');
```

end

```
% Setting the options for the Config object

% Specify an ERT target
cfg = coder.config('exe');
```

```
% Enable reentrant code generation
cfg.MultiInstanceCode = true;
```

```
% Set the post code generation command to be the 'setbuildargs' function
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config cfg main.c -report matrix_exp.m -args ones(160,160)
```

This script:

- Generates an error message if the example is not running on a UNIX platform.
- Creates a code generation configuration object for generation of an executable.
- Enables the `MultiInstanceCode` option to generate reusable, reentrant code.
- Uses the `PostCodeGenCommand` option to set the post-code-generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

```
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'};
 addLinkFlags(buildInfo, linkFlags);
```

For more information, see “Customize the Post-Code-Generation Build Process” on page 21-139.

- Invokes `codegen` with the following options:

- `-config` to pass in the code generation configuration object `cfg`.
- `main.c` to include this file in the compilation.
- `-report` to create a code generation report.
- `-args` to specify an example input with class, size, and complexity.

## Examine the Generated Code

codegen generates a header file `matrix_exp_types.h`, which defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```

/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expPersistentData
#define typedef_matrix_expPersistentData

typedef struct {
 double count;
} matrix_expPersistentData;

#endif /*typedef_matrix_expPersistentData*/

#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
 struct {
 double F[25600];
 double Y[25600];
 double X[25600];
 } f0;
}

```

```
 matrix_expPersistentData *pd;
} matrix_expStackData;

#endif /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

### Run the Code

Call the code using the command:

```
system('./matrix_exp')
```