

Code Generation from MATLAB®

User's Guide

R2012a

MATLAB®

How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

Code Generation from MATLAB User's Guide

© COPYRIGHT 2007–2012 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

March 2007	Online only	New for Release 2007a
September 2007	Online only	Revised for Release 2007b
March 2008	Online only	Revised for Release 2008a
October 2008	Online only	Revised for Release 2008b
March 2009	Online only	Revised for Release 2009a
September 2009	Online only	Revised for Release 2009b
March 2010	Online only	Revised for Release 2010a
September 2010	Online only	Revised for Release 2010b
April 2011	Online only	Revised for Release 2011a
September 2011	Online only	Revised for Release 2011b
March 2012	Online only	Revised for Release 2012a

About Code Generation from MATLAB Algorithms

1

Direct Translation of MATLAB Algorithms to C/C++ Code	1-2
Prerequisites for Code Generation from MATLAB	1-3
Preparing MATLAB Code for C/C++ and MEX Code Generation	1-4
Expected Differences in Behavior After Compiling Your MATLAB Code	1-5
Why Are There Differences?	1-5
Character Size	1-5
Order of Evaluation in Expressions	1-5
Termination Behavior	1-6
Size of Variable-Size N-D Arrays	1-6
Size of Empty Arrays	1-7
Floating-Point Numerical Results	1-7
NaN and Infinity Patterns	1-8
Code Generation Target	1-8
MATLAB Class Initial Values	1-8
MATLAB Language Features Supported for Code Generation	1-9
MATLAB Language Features Not Supported for Code Generation	1-11
Related Products That Support Code Generation from MATLAB	1-12

Functions Supported for Code Generation

2

About Code Generation for Supported Functions	2-2
Functions Supported for Code Generation —	
Alphabetical List	2-3
Functions Supported for Code Generation —	
Categorical List	2-65
Aerospace Toolbox Functions	2-66
Arithmetic Operator Functions	2-66
Bit-Wise Operation Functions	2-67
Casting Functions	2-67
Communications System Toolbox Functions	2-68
Complex Number Functions	2-68
Computer Vision System Toolbox Functions	2-69
Data Type Functions	2-70
Derivative and Integral Functions	2-70
Discrete Math Functions	2-70
Error Handling Functions	2-71
Exponential Functions	2-71
Filtering and Convolution Functions	2-72
Fixed-Point Toolbox Functions	2-72
Histogram Functions	2-81
Image Processing Toolbox Functions	2-81
Input and Output Functions	2-82
Interpolation and Computational Geometry	2-82
Linear Algebra	2-83
Logical Operator Functions	2-83
MATLAB Compiler Functions	2-84
Matrix and Array Functions	2-84
Nonlinear Numerical Methods	2-88
Polynomial Functions	2-88
Relational Operator Functions	2-88
Rounding and Remainder Functions	2-89
Set Functions	2-89
Signal Processing Functions in MATLAB	2-90
Signal Processing Toolbox Functions	2-91
Special Values	2-95
Specialized Math	2-96
Statistical Functions	2-97

String Functions	2-97
Structure Functions	2-98
Trigonometric Functions	2-98

System Objects Supported for Code Generation

3

About Code Generation for System Objects	3-2
Computer Vision System Toolbox System Objects	3-3
Communications System Toolbox System Objects	3-8
DSP System Toolbox System Objects	3-14

Defining MATLAB Variables for C/C++ Code Generation

4

Why Define Variables Differently for Code Generation?	4-2
Best Practices for Defining Variables for C/C++ Code Generation	4-3
Define Variables By Assignment Before Using Them	4-3
Use Caution When Reassigning Variables	4-6
Use Type Cast Operators in Variable Definitions	4-6
Define Matrices Before Assigning Indexed Variables	4-6
When You Can Reassign Variable Properties for C/C++ Code Generation	4-7
Eliminating Redundant Copies of Variables in Generated Code	4-8

When Redundant Copies Occur	4-8
How to Eliminate Redundant Copies by Defining Uninitialized Variables	4-8
Defining Uninitialized Variables	4-9
Defining and Initializing Persistent Variables	4-10
Reusing the Same Variable with Different Properties	4-11
When You Can Reuse the Same Variable with Different Properties	4-11
When You Cannot Reuse Variables	4-12
Limitations of Variable Reuse	4-14
Avoiding Overflows in for-Loops	4-16
Supported Variable Types	4-19

Defining Data for Code Generation

5

How Working with Data is Different for Code Generation	5-2
Code Generation for Complex Data	5-4
Restrictions When Defining Complex Variables	5-4
Expressions Containing Complex Operands Yield Complex Results	5-5
Code Generation for Characters	5-6

Code Generation for MATLAB Structures

6

How Working with Structures Is Different for Code Generation	6-2
Structure Operations Allowed for Code Generation ...	6-3
Defining Scalar Structures for Code Generation	6-4
Restrictions When Using <code>struct</code>	6-4
Restrictions When Defining Scalar Structures by Assignment	6-4
Adding Fields in Consistent Order on Each Control Flow Path	6-4
Restriction on Adding New Fields After First Use	6-5
Defining Arrays of Structures for Code Generation ...	6-7
Ensuring Consistency of Fields	6-7
Using <code>repmat</code> to Define an Array of Structures with Consistent Field Properties	6-7
Defining an Array of Structures Using Concatenation	6-8
Making Structures Persistent	6-9
Indexing Substructures and Fields	6-10
Assigning Values to Structures and Fields	6-12
Passing Large Structures as Input Parameters	6-13

Code Generation for Enumerated Data

7

How Working with Enumerated Data Is Different for Code Generation	7-2
See Also	7-2

Enumerated Types Supported for Code Generation ...	7-3
Enumerated Type Based on int32	7-3
Enumerated Type Based on Simulink.IntEnumType	7-4
When to Use Enumerated Data for Code Generation ..	7-6
Workflows for Using Enumerated Data for Code	
Generation	7-7
Workflow for Generating Code for Enumerated Data from MATLAB Algorithms	7-7
Workflow for Generating Code for Enumerated Data from MATLAB Function Blocks	7-8
How to Define Enumerated Data for Code	
Generation	7-9
Naming Enumerated Types for Code Generation	7-10
How to Instantiate Enumerated Types for Code	
Generation	7-11
How to Generate Code for Enumerated Data	7-12
See Also	7-12
Defining and Using Enumerated Types for Code	
Generation	7-13
About the Example	7-13
Class Definition: sysMode	7-13
Class Definition: LEDcolor	7-14
Function: displayState	7-14
Operations on Enumerated Data Allowed for Code	
Generation	7-15
Assignment Operator, =	7-15
Relational Operators, < > <= >= == ~=	7-15
Cast Operation	7-16
Indexing Operation	7-16
Control Flow Statements: if, switch, while	7-17
Using Enumerated Data in Control Flow Statements ..	7-18
Using the if Statement on Enumerated Data Types	7-18

Using the switch Statement on Enumerated Data Types	7-19
Using the while Statement on Enumerated Data Types ..	7-22
Customizing Enumerated Data Types	7-24
Customizing Enumerated Types Based on int32	7-24
Customizing Enumerated Types Based on Simulink.IntEnumType	7-31
Changing and Reloading Enumerated Data Types	7-32
Restrictions on Use of Enumerated Data in for-Loops	7-33
Toolbox Functions That Support Enumerated Types for Code Generation	7-34

Code Generation for Variable-Size Data

8

What Is Variable-Size Data?	8-2
How Working with Variable-Size Data Is Different for Code Generation	8-3
See Also	8-3
Bounded Versus Unbounded Variable-Size Data	8-4
How to Control Memory Allocation of Variable-Size Data	8-5
How to Generate Code for MATLAB Functions with Variable-Size Data	8-6
Generating Code for a MATLAB Function That Expands a Vector in a Loop	8-8

About the MATLAB Function <code>unique_tol</code>	8-8
Step 1: Add Compilation Directive for Code Generation ..	8-8
Step 2: Address Issues Detected by the Code Analyzer ...	8-9
Step 3: Generate MEX Code	8-9
Step 4: Fix the Size Mismatch Error	8-11
Step 5: Generate C Code	8-13
Step 6: Change the Dynamic Memory Allocation Threshold	8-14
Using Variable-Size Data Without Dynamic Memory	
Allocation	8-16
Fixing Upper Bounds Errors	8-16
Specifying Upper Bounds for Variable-Size Data	8-16
Variable-Size Data in Code Generation Reports	8-20
What Reports Tell You About Size	8-20
How Size Appears in Code Generation Reports	8-21
How to Generate a Code Generation Report	8-21
Defining Variable-Size Data for Code Generation	8-22
When to Define Variable-Size Data Explicitly	8-22
Using a Matrix Constructor with Nonconstant Dimensions	8-23
Inferring Variable Size from Multiple Assignments	8-23
Defining Variable-Size Data Explicitly Using <code>coder.varsize</code>	8-24
C Code Interface for Arrays	8-29
C Code Interface for Statically Allocated Arrays	8-29
C Code Interface for Dynamically Allocated Arrays	8-30
Utility Functions for Creating <code>emxArray</code> Data Structures	8-31
Troubleshooting Issues with Variable-Size Data	8-33
Diagnosing and Fixing Size Mismatch Errors	8-33
Diagnosing and Fixing Errors in Detecting Upper Bounds	8-35
Limitations with Variable-Size Support for Code	
Generation	8-37
Limitation on Scalar Expansion	8-37

Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays	8-38
Incompatibility with MATLAB in Determining Size of Empty Arrays	8-39
Limitation on Vector-Vector Indexing	8-40
Limitations on Matrix Indexing Operations for Code Generation	8-41
Dynamic Memory Allocation Not Supported for MATLAB Function Blocks	8-42
 Restrictions on Variable Sizing in Toolbox Functions	
Supported for Code Generation	8-43
Common Restrictions	8-43
Toolbox Functions with Variable Sizing Restrictions	8-44

Code Generation for MATLAB Classes

9

About Code Generation for MATLAB Classes	9-2
See Also	9-2
 How Working with MATLAB Classes is Different for	
Code Generation	9-3
Language Limitations	9-3
Code Generation Features Not Compatible with Classes ..	9-5
Defining Class Properties for Code Generation	9-6
Calls to Base Class Constructor	9-7
 Memory Allocation Requirements	
9-9	
 Generate Code for MATLAB Value Classes	9-10
 Generate Code for MATLAB Handle Classes and System Objects	9-16
 MATLAB Classes in Code Generation Reports	9-19
What Reports Tell You About Classes	9-19
How Classes Appear in Code Generation Reports	9-19

How to Generate a Code Generation Report	9-21
Troubleshooting	9-22
Class <i>cClass</i> does not have a property with name <i>name</i> ...	9-22

Code Generation for Function Handles

10

How Working with Function Handles is Different for Code Generation	10-2
Example: Defining and Passing Function Handles for Code Generation	10-3
Limitations with Function Handles for Code Generation	10-6

Defining Functions for Code Generation

11

Specifying Variable Numbers of Arguments	11-2
Supported Index Expressions	11-3
Using <code>varargin</code> and <code>varargout</code> in for-Loops	11-4
When to Force Loop Unrolling	11-4
Example: Using Variable Numbers of Arguments in a for-Loop	11-5
Implementing Wrapper Functions with <code>varargin</code> and <code>varargout</code>	11-7
Example: Passing Variable Numbers of Arguments from One Function to Another	11-7

Passing Property/Value Pairs with varargin	11-8
Rules for Using Variable Length Argument Lists for Code Generation	11-10

Calling Functions for Code Generation

12

How MATLAB Resolves Function Calls in Generated Code	12-2
Key Points About Resolving Function Calls	12-4
Compile Path Search Order	12-4
When to Use the Code Generation Path	12-5
 How MATLAB Resolves File Types on the Path for Code Generation	12-6
 Adding the Compilation Directive %#codegen	12-8
 Calling Subfunctions	12-9
 Calling Supported Toolbox Functions	12-10
 Calling MATLAB Functions	12-11
Declaring MATLAB Functions as Extrinsic Functions ...	12-11
Calling MATLAB Functions Using feval	12-15
How MATLAB Resolves Extrinsic Functions During Simulation	12-15
Working with mxArray	12-16
Restrictions on Extrinsic Functions for Code Generation ..	12-17
Limit on Function Arguments	12-18

Generating Efficient and Reusable Code

13

Generating Efficient Code	13-2
Unrolling for-Loops	13-2
Inlining Functions	13-2
Eliminating Redundant Copies of Function Inputs	13-2
Generating Reusable Code	13-4

Examples

A

Data Management	A-2
Code Generation for Structures	A-3
Code Generation for Enumerated Data	A-4
Generating Code for Variable-Size Data	A-5
Code Generation for Variable-Size Data	A-6
Code Generation for Function Handles	A-7
Using Variable-Length Argument Lists	A-8
Optimizing Generated Code	A-9

Index

About Code Generation from MATLAB Algorithms

- “Direct Translation of MATLAB Algorithms to C/C++ Code” on page 1-2
- “Prerequisites for Code Generation from MATLAB” on page 1-3
- “Preparing MATLAB Code for C/C++ and MEX Code Generation” on page 1-4
- “Expected Differences in Behavior After Compiling Your MATLAB Code” on page 1-5
- “MATLAB Language Features Supported for Code Generation” on page 1-9
- “MATLAB Language Features Not Supported for Code Generation” on page 1-11
- “Related Products That Support Code Generation from MATLAB” on page 1-12

Direct Translation of MATLAB Algorithms to C/C++ Code

You can automatically generate MEX functions and standalone C/C++ code from MATLAB® algorithms. With this capability, you can design, implement, and test software in the MATLAB environment, then automatically translate the algorithms to readable, efficient, and compact C/C++ code for deployment to desktop and embedded systems.

The generated code contains optimizations tailored to meet the requirements of desktop and embedded applications for speed, memory use, and data type management.

To verify the generated code in the MATLAB environment, you can generate MEX functions to compare with your original algorithm to determine whether they are functionally equivalent.

In certain applications, you can also generate MEX functions to accelerate MATLAB algorithms.

Prerequisites for 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

For more information, see:

- “Why Test MEX Functions in MATLAB?” in the MATLAB Coder documentation.
- “Generating C/C++ Code from MATLAB Code” in the MATLAB Coder documentation.
- “Accelerating MATLAB Algorithms” in the MATLAB Coder documentation.

Preparing MATLAB Code for C/C++ and MEX Code Generation

By default, the MATLAB language uses dynamic typing. Properties of dynamically typed variables can change at runtime, allowing a single variable to hold a value of any class, size, or complexity. However, to generate efficient code for statically typed languages such as C and C++, the properties of all MATLAB variables must be determined at compile time. Therefore, to prepare MATLAB code for C, C++, and MEX code generation, you must specify the class, size, and complexity of inputs to the primary function (also known as the *top-level* or *entry-point* function). By determining these properties at compile time, the code generation process translates your MATLAB algorithms into code that is efficient and tailored to your specific application, rather than producing generic code that handles every possible set of MATLAB inputs.

Expected Differences in Behavior After Compiling Your MATLAB Code

In this section...

“Why Are There Differences?” on page 1-5

“Character Size” on page 1-5

“Order of Evaluation in Expressions” on page 1-5

“Termination Behavior” on page 1-6

“Size of Variable-Size N-D Arrays” on page 1-6

“Size of Empty Arrays” on page 1-7

“Floating-Point Numerical Results” on page 1-7

“NaN and Infinity Patterns” on page 1-8

“Code Generation Target” on page 1-8

“MATLAB Class Initial Values” on page 1-8

Why Are There Differences?

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

Character Size

MATLAB supports 16-bit characters, but the generated code represents characters in 8 bits, the standard size for most embedded languages like C. See “Code Generation for Characters” on page 5-6.

Order of Evaluation in Expressions

Generated code does not enforce order of evaluation in expressions. For most expressions, order of evaluation is not significant. However, for expressions with side effects, the generated code may produce the side effects in different

order from the original MATLAB code. Expressions that produce side effects include those that:

- Modify persistent or global variables
- Display data to the screen
- Write data to files
- Modify the properties of handle class objects

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

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

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

as

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

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

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, optimizations remove infinite loops from generated code if they have no 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 8-38.

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 8-39.

Floating-Point Numerical Results

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

When computer hardware uses extended precision registers

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

For certain advanced library functions

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

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

For implementation of BLAS library functions

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

NaN and Infinity Patterns

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

Code Generation Target

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

MATLAB Class Initial Values

MATLAB computes class initial values at class loading time before code generation. The code generation software uses the value that MATLAB computed, it does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generation software does not execute this function. If the function modifies a global state, for example, a persistent variable, code generation software might provide a different initial value than MATLAB. For more information, see “Defining Class Properties for Code Generation” on page 9-6.

MATLAB Language Features Supported for Code Generation

MATLAB supports the following language features in generated code:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see “How Working with Variable-Size Data Is Different for Code Generation” on page 8-3)
- Subscripting (see “Limitations on Matrix Indexing Operations for Code Generation” on page 8-41)
- Complex numbers (see “Code Generation for Complex Data” on page 5-4)
- Numeric classes (see “Supported Variable Types” on page 4-19)
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see “Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms” in the Fixed-Point Toolbox™ documentation)
- Program control statements `if`, `switch`, `for`, and `while`
- All arithmetic, relational, and logical operators
- Subfunctions (see Chapter 12, “Calling Functions for Code Generation”)
- Persistent variables (see “Defining and Initializing Persistent Variables” on page 4-10)
- Global variables (see “Specifying Global Variable Type and Initial Value in a Project” in the MATLAB Coder documentation.
- Structures (see Chapter 6, “Code Generation for MATLAB Structures”)
- Characters (see “Code Generation for Characters” on page 5-6)
- Function handles (see Chapter 10, “Code Generation for Function Handles”)
- Frames (see “Working with Frame-Based Signals” in the Simulink® documentation.
- Variable length input and output argument lists (see Chapter 8, “Code Generation for Variable-Size Data”)

- Subset of MATLAB toolbox functions (see Chapter 2, “Functions Supported for Code Generation”)
- MATLAB classes
- Ability to call functions (see “How MATLAB Resolves Function Calls in Generated Code” on page 12-2)

MATLAB Language Features Not Supported for Code Generation

MATLAB does not support the following features in generated code:

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

Related Products That Support Code Generation from MATLAB

You can also generate code from MATLAB using other MathWorks® products that require additional licenses:

To:	Do This:	Required Licenses	Details
Generate C/C++ code from MATLAB in a Simulink model	Add MATLAB Function blocks and MATLAB Truth Table blocks to the model.	<ul style="list-style-type: none"> • Simulink • MATLAB Coder • Simulink Coder and/or Embedded Coder™ 	See: <ul style="list-style-type: none"> • “Using the MATLAB Function Block” in the Simulink documentation • “Building a Model with a Stateflow Truth Table” in the Stateflow® documentation
Generate C/C++ code from a Stateflow chart	Add MATLAB functions and MATLAB Truth Table functions to the chart.	<ul style="list-style-type: none"> • Stateflow • Simulink • MATLAB Coder • Simulink Coder 	See “Using MATLAB Functions in Stateflow Charts” and “Truth Table Functions for Decision-Making Logic” in the Stateflow documentation.

To:	Do This:	Required Licenses	Details
Accelerate fixed-point algorithms in your MATLAB code	Compile the MATLAB code with the <code>fiaccel</code> function.	Fixed-Point Toolbox	See “Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms” in the Fixed-Point Toolbox documentation.
Write and simulate MATLAB functions that manipulate data associated with entities		<ul style="list-style-type: none"> • SimEvents® • Simulink • MATLAB Coder • Simulink Coder 	See “Working with Entities” in the SimEvents documentation.
Verify that the simulation behavior of a model satisfies test objectives	Use MATLAB functions for proving properties and generating tests	<ul style="list-style-type: none"> • Simulink Design Verifier™ • Simulink • MATLAB Coder • Simulink Coder 	See “About Property Proving” and “About Test Case Generation” in the Simulink Design Verifier documentation

Functions Supported for Code Generation

- “About Code Generation for Supported Functions” on page 2-2
- “Functions Supported for Code Generation — Alphabetical List” on page 2-3
- “Functions Supported for Code Generation — Categorical List” on page 2-65

About Code Generation for Supported Functions

You can generate efficient C/C++ code for a subset of MATLAB and toolbox functions that you call from MATLAB code. In generated code, each supported function has the same name, arguments, and functionality as its MATLAB or toolbox counterparts. However, to generate code for these functions, you must adhere to certain limitations when calling them from your MATLAB source code. These limitations appear in “Functions Supported for Code Generation — Alphabetical List” on page 2-3.

To see at a glance whether there is support for a function in a particular MATLAB category or toolbox, see “Functions Supported for Code Generation — Categorical List” on page 2-65.

Note For more information on code generation for fixed-point algorithms, refer to “Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms” in the Fixed-Point Toolbox documentation.

Functions Supported for Code Generation – Alphabetical List

Function	Product	Remarks/Limitations
abs	MATLAB	—
abs	Fixed-Point Toolbox	—
acos	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>.
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>.
acot	MATLAB	—
acotd	MATLAB	—
acoth	MATLAB	—
acsc	MATLAB	—
acscd	MATLAB	—
acsch	MATLAB	—
add	Fixed-Point Toolbox	—
all	MATLAB	—
all	Fixed-Point Toolbox	—
and	MATLAB	—

Function	Product	Remarks/Limitations
angle	MATLAB	—
any	MATLAB	—
any	Fixed-Point Toolbox	—
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 <code>complex(x)</code>.
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.
atan	MATLAB	—
atan2	MATLAB	—
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>.

Function	Product	Remarks/Limitations
barthannwin	Signal Processing Toolbox™	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox™ license to generate code.
bartlett	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
besselap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Filter order 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
beta	MATLAB	—
betainc	MATLAB	—
betaln	MATLAB	—
bi2de	Communications System Toolbox™	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
bin2dec	MATLAB	<ul style="list-style-type: none"> Does not match MATLAB when the input is empty.
bitand	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitand	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	Fixed-Point Toolbox	—
bitcmp	MATLAB	<ul style="list-style-type: none"> Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitcmp	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
bitconcat	Fixed-Point Toolbox	—
bitget	MATLAB	—
bitget	Fixed-Point Toolbox	—
bitmax	MATLAB	—
bitor	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point inputs. The arguments must belong to an integer class.
bitor	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	Fixed-Point Toolbox	—
bitreplicate	Fixed-Point Toolbox	—
bitrevorder	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Computation performed at run time. • Requires DSP System Toolbox license to generate code.
bitrol	Fixed-Point Toolbox	—
bitror	Fixed-Point Toolbox	—
bitset	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point input for the first argument. The first argument must belong to an integer class.
bitset	Fixed-Point Toolbox	—
bitshift	MATLAB	<ul style="list-style-type: none"> • Does not support floating-point input for the first argument. The first argument must belong to an integer class.

Function	Product	Remarks/Limitations
bitshift	Fixed-Point Toolbox	—
bitsliceget	Fixed-Point Toolbox	—
bitsll	Fixed-Point Toolbox	—
bitsra	Fixed-Point Toolbox	—
bitsrl	Fixed-Point Toolbox	—
bitxor	MATLAB	<ul style="list-style-type: none"> Does not support floating-point inputs. The arguments must belong to an integer class.
bitxor	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	Fixed-Point Toolbox	—
blackman	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Window length 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
blackmanharris	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
blanks	MATLAB	—
blkdiag	MATLAB	—
bohmanwin	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Window length 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
bsxfun	MATLAB	—

Function	Product	Remarks/Limitations
buttap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter order 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
butter	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter coefficients 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
buttord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
cart2pol	MATLAB	—
cart2sph	MATLAB	—
cast	MATLAB	—
cat	MATLAB	—
ceil	MATLAB	—
ceil	Fixed-Point Toolbox	—
cfirpm	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
char	MATLAB	—
cheb1ap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
cheb1ord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
cheb2ap	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
cheb2ord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
chebwin	Signal Processing Toolbox	<ul style="list-style-type: none">• Does not support variable-size inputs.• All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none">• Requires DSP System Toolbox license to generate code.
cheby1	Signal Processing Toolbox	<ul style="list-style-type: none">• Does not support variable-size inputs.• All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none">• Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
cheby2	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
chol	MATLAB	—
circshift	MATLAB	—
class	MATLAB	—
compan	MATLAB	—
complex	MATLAB	—
complex	Fixed-Point Toolbox	—
cond	MATLAB	—
conj	MATLAB	—
conj	Fixed-Point Toolbox	—
conv	MATLAB	—

Function	Product	Remarks/Limitations
conv	Fixed-Point Toolbox	<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 always 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	—
convergent	Fixed-Point Toolbox	—
convn	MATLAB	—
cordicabs	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicangle	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordicatan2	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordiccart2pol	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.
cordiccxp	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Variable-size signals are not supported.

Function	Product	Remarks/Limitations
cordiccos	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicpol2cart	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicrotate	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicsin	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-size signals are not supported.
cordicsincos	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-size signals are not supported.
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	—
cosd	MATLAB	—
cosh	MATLAB	—
cot	MATLAB	—
cotd	MATLAB	—
coth	MATLAB	—
cov	MATLAB	—
cross	MATLAB	<ul style="list-style-type: none"> If supplied, <code>dim</code> must be a constant.
csc	MATLAB	—
cscd	MATLAB	—
csch	MATLAB	—
ctranspose	MATLAB	—
ctranspose	Fixed-Point Toolbox	—
cumprod	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to double first.

Function	Product	Remarks/Limitations
cumsum	MATLAB	<ul style="list-style-type: none"> Logical inputs are not supported. Cast input to double first.
cumtrapz	MATLAB	—
dct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Requires DSP System Toolbox license to generate code. 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 codegen, use <code>coder.Constant</code>. For more information, see “Specifying Constant Inputs at the Command Line”.</p>
de2bi	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
deal	MATLAB	—
dec2bin	MATLAB	<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.

Function	Product	Remarks/Limitations
dec2hex	MATLAB	<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.
deconv	MATLAB	—
de12	MATLAB	—
det	MATLAB	—
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 $[1, n-2]$, where <code>n</code> is the number of elements in a column of input argument <code>X</code>, or the number of elements in <code>X</code> when <code>X</code> is a row vector ▪ Contain all unique values
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.
diag	Fixed-Point Toolbox	<ul style="list-style-type: none"> • If supplied, the index, <code>k</code>, 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.

Function	Product	Remarks/Limitations
disp	Fixed-Point Toolbox	—
divide	Fixed-Point Toolbox	<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. The syntax <code>T.divide(a,b)</code> is not supported.
dot	MATLAB	—
double	MATLAB	—
double	Fixed-Point Toolbox	—
downsample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs.
dpss	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
eig	MATLAB	<ul style="list-style-type: none"> • QZ algorithm used in all cases, whereas MATLAB might use different algorithms for different inputs. Consequently, V might represent a different basis of eigenvectors, and the eigenvalues in D might not be in the same order as in MATLAB. • With one input, $[V,D] = \text{eig}(A)$, the results will be 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', 'nobalance' are not supported for the standard eigenvalue problem, and 'chol' is not supported for the symmetric generalized eigenvalue problem. • Outputs are always of complex type.
ellip	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Inputs 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 codegen, use <code>coder.Constant</code>. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
ellipap	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
ellipke	MATLAB	—
ellipord	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
end	Fixed-Point Toolbox	—
eps	MATLAB	—
eps	Fixed-Point Toolbox	<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	—

Function	Product	Remarks/Limitations
eq	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
erf	MATLAB	—
erfc	MATLAB	—
erfcinv	MATLAB	—
erfcx	MATLAB	—
erfinv	MATLAB	—
error	MATLAB	<ul style="list-style-type: none"> This is an extrinsic call.
estimate Fundamental Matrix	Computer Vision System Toolbox™	—
exp	MATLAB	—
expint	MATLAB	—
expm	MATLAB	—
expm1	MATLAB	—
eye	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants.
factor	MATLAB	<ul style="list-style-type: none"> For double precision input, the maximum value of A is $2^{32} - 1$. For single precision input, the maximum value of A is $2^{24} - 1$.
factorial	MATLAB	—
false	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants.
fft	MATLAB	<ul style="list-style-type: none"> Length of input vector must be a power of 2.
fft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.
fftn	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2.

Function	Product	Remarks/Limitations
fftshift	MATLAB	—
fi	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Use to create a fixed-point constant or variable. • The default constructor syntax without any input arguments is not supported. • The syntax <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>. • Works for all input values when complete <code>numericType</code> information of the <code>fi</code> object is provided. • Works only for constant input values (value of input must be known at compile time) when complete <code>numericType</code> information of the <code>fi</code> object is not specified. • <code>numericType</code> object information must be available for non-fixed-point Simulink inputs.
filter	MATLAB	—
filter	Fixed-Point Toolbox	<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	—

Function	Product	Remarks/Limitations
filtfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Filter coefficients 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 codegen, use coder.Constant. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
fimath	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned the fimath object defined in the MATLAB Function dialog in the Model Explorer. • Use to create fimath objects in generated code.
find	MATLAB	<ul style="list-style-type: none"> • Issues an error if a variable-sized 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-sized inputs, the shape of empty outputs, 0-by-0, 0-by-1, or 1-by-0, depends on the upper bounds of the size of the input. The output might not match MATLAB when the input array is a scalar or [] at run time. If the input is a variable-length row vector, the size of an empty output is 1-by-0, otherwise it is 0-by-1.

Function	Product	Remarks/Limitations
fir1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
fir2	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
fircls	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
fircls1	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
firls	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
firpm	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
firpmord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
firrcos	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
fix	MATLAB	—
fix	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
flattopwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
flipdim	MATLAB	—
fliplr	MATLAB	—
flipud	MATLAB	—
floor	MATLAB	—
floor	Fixed-Point Toolbox	—
freqspace	MATLAB	—
freqz	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. <code>freqz</code> with no output arguments produces a plot only when the function call terminates in a semicolon. See “freqz With No Output Arguments”. Requires DSP System Toolbox license to generate code.
fspecial	Image Processing Toolbox™	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
full	MATLAB	—

Function	Product	Remarks/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). • Only supports the TolX and FunValCheck fields of an options input structure. Ignores all other options in an options input structure. You cannot use the optimset function to create the options structure. Create this structure directly, for example, <pre>opt.TolX = tol; opt.FunValCheck = 'on';</pre> The input structure field names must match exactly.
gamma	MATLAB	—
gammainc	MATLAB	—
gammaln	MATLAB	—
gaussfir	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 codegen, use coder.Constant. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
gausswin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
gcd	MATLAB	—
ge	MATLAB	—
ge	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
get	Fixed-Point Toolbox	<ul style="list-style-type: none"> The syntax <code>structure = get(o)</code> is not supported.
getlsb	Fixed-Point Toolbox	—
getmsb	Fixed-Point Toolbox	—
gradient	MATLAB	—
gt	MATLAB	—
gt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
hadamard	MATLAB	—

Function	Product	Remarks/Limitations
hamming	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
hankel	MATLAB	—
hann	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
hex2dec	MATLAB	—
hilb	MATLAB	—
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.

Function	Product	Remarks/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.
horzcat	Fixed-Point Toolbox	—
hypot	MATLAB	—
idct	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
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"> Length of input vector must be a power of 2. Output of <code>ifft</code> block is always complex. Does not support the 'symmetric' option.
ifft2	MATLAB	<ul style="list-style-type: none"> Length of input matrix dimensions must each be a power of 2. Does not support the 'symmetric' option.

Function	Product	Remarks/Limitations
ifftn	MATLAB	<ul style="list-style-type: none"> • Length of input matrix dimensions must each be a power of 2. • Does not support the 'symmetric' option.
ifftshift	MATLAB	—
imag	MATLAB	—
imag	Fixed-Point Toolbox	—
ind2sub	MATLAB	<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.
inf	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants.
int8, int16, int32	MATLAB	—
int8, int16, int32	Fixed-Point Toolbox	—
interp1	MATLAB	<ul style="list-style-type: none"> • Supports only linear and nearest interpolation methods. • Does not handle evenly spaced X indices separately. • X must be strictly monotonically increasing or strictly monotonically decreasing; does not reorder indices.

Function	Product	Remarks/Limitations
intersect	MATLAB	<ul style="list-style-type: none"> • When rows is not specified: <ul style="list-style-type: none"> ▪ Inputs must be row vectors. ▪ If a vector is variable-sized, its first dimension must have a fixed length of 1. ▪ The input [] is not supported. Use a 1-by-0 input, for example zeros(1,0), to represent the empty set. ▪ Empty outputs are always row vectors, 1-by-0, never 0-by-0. • When rows is specified, outputs ia and ib are always 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. • Inputs must already be sorted in ascending order. The first output is always sorted in ascending order. • Complex inputs must be single or double.
intfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 codegen, use coder.Constant. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
intmax	MATLAB	—
intmin	MATLAB	—

Function	Product	Remarks/Limitations
inv	MATLAB	Singular matrix inputs can produce nonfinite values that differ from MATLAB results.
invhilb	MATLAB	—
ipermute	MATLAB	—
isa	MATLAB	—
iscell	MATLAB	—
ischar	MATLAB	—
iscolumn	MATLAB	—
iscolumn	Fixed-Point Toolbox	—
isdeployed	MATLAB Compiler™	<ul style="list-style-type: none"> • Returns true and false as appropriate for MEX and SIM targets • Returns false for all other targets
isempty	MATLAB	—
isempty	Fixed-Point Toolbox	—
isequal	MATLAB	—
isequal	Fixed-Point Toolbox	—
isequaln	MATLAB	—
isfi	Fixed-Point Toolbox	—
isfield	MATLAB	<ul style="list-style-type: none"> • Does not support cell input for second argument
isfimath	Fixed-Point Toolbox	—
isfimathlocal	Fixed-Point Toolbox	—
isfinite	MATLAB	—

Function	Product	Remarks/Limitations
isfinite	Fixed-Point Toolbox	—
isfloat	MATLAB	—
isinf	MATLAB	—
isinf	Fixed-Point Toolbox	—
isinteger	MATLAB	—
islogical	MATLAB	—
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 all other targets.
ismember	MATLAB	<ul style="list-style-type: none"> • The second input, S, must be sorted in ascending order. • Complex inputs must be single or double.
isnan	MATLAB	—
isnan	Fixed-Point Toolbox	—
isnumeric	MATLAB	—
isnumeric	Fixed-Point Toolbox	—
isnumerictype	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
isprime	MATLAB	<ul style="list-style-type: none"> • For double precision input, the maximum value of A is $2^{32}-1$. • For single precision input, the maximum value of A is $2^{24}-1$.
isreal	MATLAB	—
isreal	Fixed-Point Toolbox	—
isrow	MATLAB	—
isrow	Fixed-Point Toolbox	—
isscalar	MATLAB	—
isscalar	Fixed-Point Toolbox	—
assigned	Fixed-Point Toolbox	—
issorted	MATLAB	—
issparse	MATLAB	—
isstruct	MATLAB	—
istrellis	Communications System Toolbox	<ul style="list-style-type: none"> • Requires a Communications System Toolbox license to generate code.
isvector	MATLAB	—

Function	Product	Remarks/Limitations
isvector	Fixed-Point Toolbox	—
kaiser	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
kaiserord	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Computation performed at run time. • Requires DSP System Toolbox license to generate code.
kron	MATLAB	—
label2rgb	Image Processing Toolbox	<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</code>-by-3, <code>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.

Function	Product	Remarks/Limitations
		<ul style="list-style-type: none"> If you supply a value for <code>order</code>, it must be <code>'noshuffle'</code>.
<code>lcm</code>	MATLAB	—
<code>ldivide</code>	MATLAB	—
<code>le</code>	MATLAB	—
<code>le</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
<code>length</code>	MATLAB	—
<code>length</code>	Fixed-Point Toolbox	—
<code>levinson</code>	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
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>All other options are equivalent to using <code>mldivide</code>.</p>
linspace	MATLAB	—
log	MATLAB	<ul style="list-style-type: none"> • Generates an error during simulation and returns NaN in generated code when the input value <code>x</code> 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 Toolbox	—

Function	Product	Remarks/Limitations
logspace	MATLAB	—
lowerbound	Fixed-Point Toolbox	—
lsb	Fixed-Point Toolbox	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, fi single and double signals.
lt	MATLAB	—
lt	Fixed-Point Toolbox	<ul style="list-style-type: none"> Not supported for fixed-point signals with different biases.
lu	MATLAB	—
magic	MATLAB	—
max	MATLAB	—
max	Fixed-Point Toolbox	—
maxflat	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
mean	MATLAB	—
mean	Fixed-Point Toolbox	—
median	MATLAB	—

Function	Product	Remarks/Limitations
median	Fixed-Point Toolbox	—
meshgrid	MATLAB	—
min	MATLAB	—
min	Fixed-Point Toolbox	—
minus	MATLAB	—
minus	Fixed-Point Toolbox	<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.
mldivide	MATLAB	—
mod	MATLAB	<ul style="list-style-type: none"> Performs all arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors.
mode	MATLAB	<ul style="list-style-type: none"> Does not support third output argument <code>C</code> (cell array)
mpower	MATLAB	—
mpower	Fixed-Point Toolbox	<ul style="list-style-type: none"> The exponent input, k, must be constant; that is, its value must be known at compile time. 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 always 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>

Function	Product	Remarks/Limitations
		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>mpy</code>	Fixed-Point Toolbox	<ul style="list-style-type: none"> 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>.
<code>mrdivide</code>	MATLAB	—
<code>mrdivide</code>	Fixed-Point Toolbox	—
<code>mtimes</code>	MATLAB	—
<code>mtimes</code>	Fixed-Point Toolbox	<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 always 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

Function	Product	Remarks/Limitations
		either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
NaN or nan	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants
nargchk	MATLAB	<ul style="list-style-type: none"> • Output structure does not include stack information.
nargin	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> <hr/>
nargoutchk	MATLAB	<ul style="list-style-type: none"> • Output structure does not include stack information.
nchoosek	MATLAB	—
ndgrid	MATLAB	—
ndims	MATLAB	—
ndims	Fixed-Point Toolbox	—
ne	MATLAB	—
ne	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Not supported for fixed-point signals with different biases.
nearest	Fixed-Point Toolbox	—
nextpow2	MATLAB	—
nnz	MATLAB	—

Function	Product	Remarks/Limitations
nonzeros	MATLAB	—
norm	MATLAB	—
normest	MATLAB	—
not	MATLAB	—
nthroot	MATLAB	—
null	MATLAB	<ul style="list-style-type: none"> • Might return a different basis than MATLAB • Does not support rational basis option (second input)
numberofelements	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Returns the number of elements of <code>fi</code> objects in the generated code (works the same as <code>numel</code> for <code>fi</code> objects in generated code).
numel	MATLAB	<ul style="list-style-type: none"> • Returns the number of elements of <code>fi</code> objects in the generated code, rather than always returning 1.
numerictype	Fixed-Point Toolbox	<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 non-fixed-point signal. • Use to create <code>numerictype</code> objects in the generated code.

Function	Product	Remarks/Limitations
nuttallwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
ones	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants
or	MATLAB	—
orth	MATLAB	<ul style="list-style-type: none"> Might return a different basis than MATLAB
parzenwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
pascal	MATLAB	—
permute	MATLAB	—
permute	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
pi	MATLAB	—
pinv	MATLAB	—
planerot	MATLAB	—
plus	MATLAB	—
plus	Fixed-Point Toolbox	<ul style="list-style-type: none"> Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
pol2cart	MATLAB	—
poly	MATLAB	<ul style="list-style-type: none"> Does not discard nonfinite input values Complex input always produces complex output
poly2trellis	Communications System Toolbox	<ul style="list-style-type: none"> Requires a Communications System Toolbox license to generate code.
polyfit	MATLAB	—
polyval	MATLAB	—
pow2	Fixed-Point Toolbox	—
power	MATLAB	<ul style="list-style-type: none"> Generates an error during simulation and returns NaN in generated code when both X and Y are real, but $\text{power}(X, Y)$ is complex. To get the complex result, make the input value X complex by passing in <code>complex(X)</code>. For example, <code>power(complex(X), Y)</code>. Generates an error during simulation and returns NaN in generated code when both X and Y are real, but $X.^Y$ is complex. To get the complex result, make the input value X complex by using <code>complex(X)</code>. For example, <code>complex(X).^Y</code>.
power	Fixed-Point Toolbox	<ul style="list-style-type: none"> The exponent input, k, must be constant; that is, its value must be known at compile time.

Function	Product	Remarks/Limitations
primes	MATLAB	—
prod	MATLAB	—
qr	MATLAB	—
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	—
quatconj	Aerospace Toolbox	—
quatdivide	Aerospace Toolbox	—
quatinv	Aerospace Toolbox	—
quatmod	Aerospace Toolbox	—
quatmultiply	Aerospace Toolbox	—
quatnorm	Aerospace Toolbox	—
quatnormalize	Aerospace Toolbox	—
rand	MATLAB	—
randi	MATLAB	—
randn	MATLAB	—
randperm	MATLAB	—
range	Fixed-Point Toolbox	—

Function	Product	Remarks/Limitations
rank	MATLAB	—
rcond	MATLAB	—
rcosfir	Communications System Toolbox	<ul style="list-style-type: none"> • Requires a Communications System Toolbox license to generate code.
rdivide	MATLAB	—
rdivide	Fixed-Point Toolbox	—
real	MATLAB	—
real	Fixed-Point Toolbox	—
reallog	MATLAB	—
realmax	MATLAB	—
realmax	Fixed-Point Toolbox	—
realmin	MATLAB	—
realmin	Fixed-Point Toolbox	—
realpow	MATLAB	—
realsqrt	MATLAB	—

Function	Product	Remarks/Limitations
rectwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
reinterpretcast	Fixed-Point Toolbox	—
rem	MATLAB	<ul style="list-style-type: none"> Performs all arithmetic in the output class. Hence, results might not match MATLAB due to different rounding errors.
repmat	MATLAB	—
repmat	Fixed-Point Toolbox	—
resample	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. The upsampling and downsampling factors must be specified as 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
rescale	Fixed-Point Toolbox	—
reshape	MATLAB	—
reshape	Fixed-Point Toolbox	—
rng	MATLAB	<ul style="list-style-type: none"> • For library and executable code generation targets, and for MEX targets when extrinsic calls are disabled, supports only the 'default' input and these generator inputs: <ul style="list-style-type: none"> ▪ 'twister' ▪ 'v4' ▪ '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>.
roots	MATLAB	<ul style="list-style-type: none"> • Output is always variable size • Output is always complex • Roots may not be in the same order as MATLAB • Roots of poorly conditioned polynomials may not match MATLAB
rosser	MATLAB	—
rot90	MATLAB	—
round	MATLAB	—

Function	Product	Remarks/Limitations
round	Fixed-Point Toolbox	—
rsf2csf	MATLAB	—
schur	MATLAB	Might sometimes return a different Schur decomposition in generated code than in MATLAB.
sec	MATLAB	—
secd	MATLAB	—
sech	MATLAB	—
setdiff	MATLAB	<ul style="list-style-type: none"> • When rows is not specified: <ul style="list-style-type: none"> ▪ Inputs must be row vectors. ▪ If a vector is variable-sized, its first dimension must have a fixed length of 1. ▪ The input [] is not supported. Use a 1-by-0 input, for example, zeros(1,0) to represent the empty set. ▪ Empty outputs are always row vectors, 1-by-0, never 0-by-0. • When rows is specified, output i is always a column vector. If i is empty, it is 0-by-1, never 0-by-0, even if the output c is 0-by-0. • Inputs must already be sorted in ascending order. The first output is always sorted in ascending order. • Complex inputs must be single or double.

Function	Product	Remarks/Limitations
setxor	MATLAB	<ul style="list-style-type: none"> • When rows is not specified: <ul style="list-style-type: none"> ▪ Inputs must be row vectors. ▪ If a vector is variable-sized, its first dimension must have a fixed length of 1. ▪ The input [] is not supported. Use a 1-by-0 input, such as zeros(1,0), to represent the empty set. ▪ Empty outputs are always row vectors, 1-by-0, never 0-by-0. • When rows is specified, outputs ia and ib are always 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. • Inputs must already be sorted in ascending order. The first output is always sorted in ascending order. • Complex inputs must be single or double.
sfi	Fixed-Point Toolbox	—
sgolay	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • All inputs 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 codegen, use coder.Constant. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.

Function	Product	Remarks/Limitations
shiftdim	MATLAB	Second argument must be a constant.
sign	MATLAB	—
sign	Fixed-Point Toolbox	—
sin	MATLAB	—
sind	MATLAB	—
single	MATLAB	—
single	Fixed-Point Toolbox	—
sinh	MATLAB	—
size	MATLAB	—
size	Fixed-Point Toolbox	—
sort	MATLAB	—
sort	Fixed-Point Toolbox	—
sortrows	MATLAB	—
sosfilt	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Computation performed at run time. • Requires DSP System Toolbox license to generate code.
sph2cart	MATLAB	—
squeeze	MATLAB	—
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>.

Function	Product	Remarks/Limitations
sqrt	Fixed-Point Toolbox	<ul style="list-style-type: none"> • Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sqrtm	MATLAB	—
std	MATLAB	—
storedInteger	Fixed-Point Toolbox	—
storedIntegerToDouble	Fixed-Point Toolbox	—
str2func	MATLAB	<ul style="list-style-type: none"> • String must be constant/known at compile time
strcmp	MATLAB	<ul style="list-style-type: none"> • Arguments must be computable at compile time.
struct	MATLAB	—
structfun	MATLAB	<ul style="list-style-type: none"> • Does not support the ErrorHandler option. • The number of outputs must be less than or equal to three.
sub	Fixed-Point Toolbox	—
sub2ind	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.
subsasgn	Fixed-Point Toolbox	—
subspace	MATLAB	—
subsref	Fixed-Point Toolbox	—
sum	MATLAB	—

Function	Product	Remarks/Limitations
sum	Fixed-Point Toolbox	<ul style="list-style-type: none"> Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
svd	MATLAB	Uses a different SVD implementation than MATLAB. As 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 swapbytes in a MATLAB Function block is supported only when the class of the input is double. For non-double inputs, the input port data types must be specified, not inherited.
tan	MATLAB	—
tand	MATLAB	—
tanh	MATLAB	—
taylorwin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. Inputs must be constant <p>Specifying constants</p> <p>To specify a constant input for codegen, use coder.Constant. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
times	MATLAB	—

Function	Product	Remarks/Limitations
times	Fixed-Point Toolbox	<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>.
toeplitz	MATLAB	—
trace	MATLAB	—
trapz	MATLAB	—
transpose	MATLAB	—
transpose	Fixed-Point Toolbox	—
triang	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
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.

Function	Product	Remarks/Limitations
tril	Fixed-Point Toolbox	<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 Toolbox	<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.
true	MATLAB	<ul style="list-style-type: none"> Dimensions must be real, nonnegative, integer constants
tukeywin	Signal Processing Toolbox	<ul style="list-style-type: none"> Does not support variable-size inputs. All inputs 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> Requires DSP System Toolbox license to generate code.
typecast	MATLAB	<ul style="list-style-type: none"> Value of string input argument <code>type</code> must be lower case You might receive a size error when you use <code>typecast</code> with inheritance of input port data types in MATLAB Function blocks. To avoid this error, specify the block’s input port data types explicitly.
ufi	Fixed-Point Toolbox	—
uint8, uint16, uint32	MATLAB	—

Function	Product	Remarks/Limitations
uint8, uint16, uint32	Fixed-Point Toolbox	—
uminus	MATLAB	—
uminus	Fixed-Point Toolbox	—
union	MATLAB	<ul style="list-style-type: none"> • When rows is not specified: <ul style="list-style-type: none"> ▪ Inputs must be row vectors. ▪ If a vector is variable-sized, its first dimension must have a fixed length of 1. ▪ The input [] is not supported. Use a 1-by-0 input, such as zeros(1,0) to represent the empty set. ▪ Empty outputs are always row vectors, 1-by-0, never 0-by-0. • When rows is specified, outputs ia and ib are always 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. • Inputs must already be sorted in ascending order. The first output is always sorted in ascending order. • Complex inputs must be single or double.

Function	Product	Remarks/Limitations
unique	MATLAB	<ul style="list-style-type: none"> • When rows is not specified: <ul style="list-style-type: none"> ▪ The first input must be a row vector. ▪ If the vector is variable-sized, its first dimension must have a fixed length of 1. ▪ The input [] is not supported. Use a 1-by-0 input, such as zeros(1,0), to represent the empty set. ▪ Empty outputs are always row vectors, 1-by-0, never 0-by-0. • When rows is specified, outputs m and n are always column vectors. If these outputs are empty, they are 0-by-1, never 0-by-0, even if the output b is 0-by-0. • Complex inputs must be single or double.
unwrap	MATLAB	<ul style="list-style-type: none"> • Row vector input is only supported when the first two inputs are vectors and nonscalar • Performs all 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"> • Does not support variable-size inputs. • 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 codegen, use coder.Constant. For more information, see “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code

Function	Product	Remarks/Limitations
uplus	MATLAB	—
uplus	Fixed-Point Toolbox	—
upperbound	Fixed-Point Toolbox	—
upsample	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • 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, <pre>assert(n<10)</pre>
vander	MATLAB	—
var	MATLAB	—
vertcat	Fixed-Point Toolbox	—
wilkinson	MATLAB	—
xcorr	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • Does not support the case where A is a matrix • Does not support partial (abbreviated) strings of <code>biased</code>, <code>unbiased</code>, <code>coeff</code>, or <code>none</code> • Computation performed at run time. • Requires DSP System Toolbox license to generate code
xor	MATLAB	—

Function	Product	Remarks/Limitations
yulewalk	Signal Processing Toolbox	<ul style="list-style-type: none"> • Does not support variable-size inputs. • 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 “Specifying Constant Inputs at the Command Line”.</p> <ul style="list-style-type: none"> • Requires DSP System Toolbox license to generate code.
zeros	MATLAB	<ul style="list-style-type: none"> • Dimensions must be real, nonnegative, integer constants
zp2tf	MATLAB	—

Functions Supported for Code Generation — Categorical List

In this section...

- “Aerospace Toolbox Functions” on page 2-66
- “Arithmetic Operator Functions” on page 2-66
- “Bit-Wise Operation Functions” on page 2-67
- “Casting Functions” on page 2-67
- “Communications System Toolbox Functions” on page 2-68
- “Complex Number Functions” on page 2-68
- “Computer Vision System Toolbox Functions” on page 2-69
- “Data Type Functions” on page 2-70
- “Derivative and Integral Functions” on page 2-70
- “Discrete Math Functions” on page 2-70
- “Error Handling Functions” on page 2-71
- “Exponential Functions” on page 2-71
- “Filtering and Convolution Functions” on page 2-72
- “Fixed-Point Toolbox Functions” on page 2-72
- “Histogram Functions” on page 2-81
- “Image Processing Toolbox Functions” on page 2-81
- “Input and Output Functions” on page 2-82
- “Interpolation and Computational Geometry” on page 2-82
- “Linear Algebra” on page 2-83
- “Logical Operator Functions” on page 2-83
- “MATLAB Compiler Functions” on page 2-84
- “Matrix and Array Functions” on page 2-84
- “Nonlinear Numerical Methods” on page 2-88
- “Polynomial Functions” on page 2-88

In this section...

“Relational Operator Functions” on page 2-88
“Rounding and Remainder Functions” on page 2-89
“Set Functions” on page 2-89
“Signal Processing Functions in MATLAB” on page 2-90
“Signal Processing Toolbox Functions” on page 2-91
“Special Values” on page 2-95
“Specialized Math” on page 2-96
“Statistical Functions” on page 2-97
“String Functions” on page 2-97
“Structure Functions” on page 2-98
“Trigonometric Functions” on page 2-98

Aerospace Toolbox Functions

Function	Description
quatconj	Calculate conjugate of quaternion
quatdivide	Divide quaternion by another quaternion
quatinv	Calculate inverse of quaternion
quatmod	Calculate modulus of quaternion
quatmultiply	Calculate product of two quaternions
quatnorm	Calculate norm of quaternion
quatnormalize	Normalize quaternion

Arithmetic Operator Functions

See Arithmetic Operators + - * / \ ^ ' in the MATLAB Function Reference documentation for detailed descriptions of the following operator equivalent functions.

Function	Description
ctranspose	Complex conjugate transpose (')
idivide	Integer division with rounding option
isa	Determine if input is object of given class
ldivide	Left array divide
minus	Minus (-)
mldivide	Left matrix divide (\)
mpower	Equivalent of array power operator (.^)
mrdivide	Right matrix divide
mtimes	Matrix multiply (*)
plus	Plus (+)
power	Array power
rdivide	Right array divide
times	Array multiply
transpose	Matrix transpose (')
uminus	Unary minus (-)
uplus	Unary plus (+)

Bit-Wise Operation Functions

Function	Description
swapbytes	Swap byte ordering

Casting Functions

Data Type	Description
cast	Cast variable to different data type
char	Create character array (string)

Data Type	Description
class	Query class of object argument
double	Convert to double-precision floating point
int8, int16, int32	Convert to signed integer data type
logical	Convert to Boolean true or false data type
single	Convert to single-precision floating point
typecast	Convert data types without changing underlying data
uint8, uint16, uint32	Convert to unsigned integer data type

Communications System Toolbox Functions

Function	Remarks/Limitations
bi2de	—
de2bi	—
istrellis	—
poly2trellis	—
rcosfir	—

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components
conj	Return the conjugate of a complex number
imag	Return the imaginary part of a complex number
isnumeric	Return true for numeric arrays
isreal	Return false (0) for a complex number
isscalar	Return true if array is a scalar

Function	Description
real	Return the real part of a complex number
unwrap	Correct phase angles to produce smoother phase plots

Computer Vision System Toolbox Functions

Function	Description
epipolarLine	Compute epipolar lines for stereo images
estimateFundamentalMatrix	Estimate fundamental matrix from corresponding points in stereo image
estimateUncalibratedRectification	Uncalibrated stereo rectification
extractFeatures	Extract interest point descriptors
isEpipoleInImage	Determine whether image contains epipole
lineToBorderPoints	Intersection points of lines in image and image border
matchFeatures	Find matching image features

Data Type Functions

Function	Description
deal	Distribute inputs to outputs
iscell	Determine whether input is cell array
nargchk	Validate number of input arguments
nargoutchk	Validate number of output arguments
str2func	Construct function handle from function name string
structfun	Apply function to each field of scalar structure

Derivative and Integral Functions

Function	Description
cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives
gradient	Numerical gradient
trapz	Trapezoidal numerical integration

Discrete Math Functions

Function	Description
factor	Return a row vector containing the prime factors of n
gcd	Return an array containing the greatest common divisors of the corresponding elements of integer arrays
isprime	Array elements that are prime numbers
lcm	Least common multiple of corresponding elements in arrays
nchoosek	Binomial coefficient or all combinations
primes	Generate list of prime numbers

Error Handling Functions

Function	Description
<code>assert</code>	Generate error when condition is violated
<code>error</code>	Display message and abort function

Exponential Functions

Function	Description
<code>exp</code>	Exponential
<code>expm</code>	Matrix exponential
<code>expm1</code>	Compute $\exp(x) - 1$ accurately for small values of x
<code>factorial</code>	Factorial function
<code>log</code>	Natural logarithm
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
<code>log10</code>	Common (base 10) logarithm
<code>log1p</code>	Compute $\log(1+x)$ accurately for small values of x
<code>nextpow2</code>	Next higher power of 2
<code>nthroot</code>	Real n th root of real numbers
<code>reallog</code>	Natural logarithm for nonnegative real arrays
<code>realpow</code>	Array power for real-only output
<code>realsqrt</code>	Square root for nonnegative real arrays
<code>sqrt</code>	Square root

Filtering and Convolution Functions

Function	Description
<code>conv</code>	Convolution and polynomial multiplication
<code>conv2</code>	2-D convolution
<code>convn</code>	N-D convolution
<code>deconv</code>	Deconvolution and polynomial division
<code>detrend</code>	Remove linear trends
<code>filter</code>	1-D digital filter
<code>filter2</code>	2-D digital filter

Fixed-Point Toolbox Functions

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

- `fipref` and quantizer objects are not supported.
- Dot notation is only supported for getting the values of `fimath` and `numericType` properties. Dot notation is not supported for `fi` objects, and it is not supported for setting properties.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numericType` of a given variable after that variable has been created.
- The boolean and `ScaledDouble` values 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`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- 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.

- All general limitations of C/C++ code generated from MATLAB apply. See “MATLAB Language Features Not Supported for Code Generation” on page 1-11 for more information.

Function	Remarks/Limitations
abs	N/A
add	N/A
all	N/A
any	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	N/A
bitsra	N/A
bitsrl	N/A
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 SumMode property of the governing fimath is set to Specify precision or Keep LSB. • 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 always computed using the SumMode property of the governing fimath. ▪ In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
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.
cordiccxp	Variable-size signals are not supported.
cordiccos	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.
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.
disp	—

Function	Remarks/Limitations
divide	<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>.
double	N/A
end	N/A
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.
eq	Not supported for fixed-point signals with different biases.
fi	<ul style="list-style-type: none"> Use to create a fixed-point constant or variable in the generated code. The default constructor syntax without any input arguments is not supported. The syntax <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>. Works for all input values when complete <code>numericType</code> information of the <code>fi</code> object is provided. Works only for constant input values (value of input must be known at compile time) when complete <code>numericType</code> information of the <code>fi</code> object is not specified.

Function	Remarks/Limitations
	<ul style="list-style-type: none"> • <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>.
<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.
<code>fix</code>	N/A
<code>floor</code>	N/A
<code>ge</code>	Not supported for fixed-point signals with different biases.
<code>get</code>	The syntax <code>structure = get(o)</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</code>	N/A
<code>iscolumn</code>	N/A
<code>isempty</code>	N/A
<code>isequal</code>	N/A
<code>isfi</code>	N/A
<code>isfimath</code>	N/A
<code>isfimathlocal</code>	N/A

Function	Remarks/Limitations
isfinite	N/A
isinf	N/A
isnan	N/A
isnumeric	N/A
isnumerictype	N/A
isreal	N/A
isrow	N/A
isscalar	N/A
issigned	N/A
isvector	N/A
le	Not supported for fixed-point signals with different biases.
length	N/A
logical	N/A
lowerbound	N/A
lsb	<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.
lt	Not supported for fixed-point signals with different biases.
max	N/A
mean	N/A
median	N/A
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.

Function	Remarks/Limitations
mpower	<ul style="list-style-type: none"> • The exponent input, k, must be constant; that is, its value must be known at compile time. • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fi</code>math is set to <code>Specify precision</code> or <code>Keep LSB</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 always computed using the <code>SumMode</code> property of the governing <code>fi</code>math. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fi</code>math when the first input, a, is nonscalar. However, when a is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fi</code>math.
mpy	<p>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 <code>0n</code>.</p>
mrdivide	N/A
mtimes	<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>fi</code>math is set to <code>Specify precision</code> or <code>Keep LSB</code>. • For variable-sized signals, you may see different results between the generated code and MATLAB.

Function	Remarks/Limitations
	<ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is always 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>.
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	numberofelements and numel both work the same as MATLAB numel for <code>fi</code> objects in the generated code.
numerictype	<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.
permute	N/A
plus	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.
pow2	N/A
power	The exponent input, k , must be constant; that is, its value must be known at compile time.
range	N/A

Function	Remarks/Limitations
rdivide	N/A
real	N/A
realmax	N/A
realmin	N/A
reinterpretcast	N/A
repmat	N/A
rescale	N/A
reshape	N/A
round	N/A
sfi	N/A
sign	N/A
single	N/A
size	N/A
sort	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	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.

Function	Remarks/Limitations
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>0n</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	N/A
uint8, uint16, uint32	N/A
uminus	N/A
uplus	N/A
upperbound	N/A
vertcat	N/A

Histogram Functions

Function	Description
hist	Non-graphical histogram
histc	Histogram count

Image Processing Toolbox Functions

You must have the Image Processing Toolbox software installed to generate C/C++ code from MATLAB for these functions.

Function	Remarks/Limitations
<code>fspecial</code>	All inputs must be constants at compilation time. Expressions or variables are allowed if their values do not change.
<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</code>-by-3, <code>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 <code>'noshuffle'</code>.

Input and Output Functions

Function	Description
<code>nargin</code>	Return the number of input arguments a user has supplied
<code>nargout</code>	Return the number of output return values a user has requested

Interpolation and Computational Geometry

Function	Description
<code>cart2pol</code>	Transform Cartesian coordinates to polar or cylindrical
<code>cart2sph</code>	Transform Cartesian coordinates to spherical
<code>interp1</code>	One-dimensional interpolation (table lookup)
<code>meshgrid</code>	Generate <code>X</code> and <code>Y</code> arrays for 3-D plots
<code>pol2cart</code>	Transform polar or cylindrical coordinates to Cartesian
<code>sph2cart</code>	Transform spherical coordinates to Cartesian

Linear Algebra

Function	Description
linsolve	Solve linear system of equations
null	Null space
orth	Range space of matrix
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root

Logical Operator Functions

Function	Description
and	Logical AND (&&)
bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
not	Logical NOT (~)
or	Logical OR ()
xor	Logical exclusive-OR

MATLAB Compiler Functions

Function	Description
isdeployed	Determine whether code is running in deployed or MATLAB mode
ismcc	Test if code is running during compilation process (using mcc)

Matrix and Array Functions

Function	Description
abs	Return absolute value and complex magnitude of an array
all	Test if all elements are nonzero
angle	Phase angle
any	Test for any nonzero elements
blkdiag	Construct block diagonal matrix from input arguments
bsxfun	Applies element-by-element binary operation to two arrays with singleton expansion enabled
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly
compan	Companion matrix
cond	Condition number of a matrix with respect to inversion
cov	Covariance matrix
cross	Vector cross product
cumprod	Cumulative product of array elements
cumsum	Cumulative sum of array elements
det	Matrix determinant
diag	Return a matrix formed around the specified diagonal vector and the specified diagonal (0, 1, 2,...) it occupies
diff	Differences and approximate derivatives
dot	Vector dot product

Function	Description
<code>eig</code>	Eigenvalues and eigenvectors
<code>eye</code>	Identity matrix
<code>false</code>	Return an array of 0s for the specified dimensions
<code>find</code>	Find indices and values of nonzero elements
<code>flipdim</code>	Flip array along specified dimension
<code>flipplr</code>	Flip matrix left to right
<code>flipud</code>	Flip matrix up to down
<code>full</code>	Convert sparse matrix to full matrix
<code>hadamard</code>	Hadamard matrix
<code>hankel</code>	Hankel matrix
<code>hilb</code>	Hilbert matrix
<code>ind2sub</code>	Subscripts from linear index
<code>inv</code>	Inverse of a square matrix
<code>invhilb</code>	Inverse of Hilbert matrix
<code>ipermute</code>	Inverse permute dimensions of array
<code>iscolumn</code>	True if input is a column vector
<code>isempty</code>	Determine whether array is empty
<code>isequal</code>	Test arrays for equality
<code>isequaln</code>	Test arrays for equality, treating NaNs as equal
<code>isfinite</code>	Detect finite elements of an array
<code>isfloat</code>	Determine if input is floating-point array
<code>isinf</code>	Detect infinite elements of an array
<code>isinteger</code>	Determine if input is integer array
<code>islogical</code>	Determine if input is logical array
<code>ismatrix</code>	True if input is a matrix
<code>isnan</code>	Detect NaN elements of an array

Function	Description
isrow	True if input is a row vector
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
kron	Kronecker tensor product
length	Return the length of a matrix
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
lu	Matrix factorization
magic	Magic square
max	Maximum elements of a matrix
min	Minimum elements of a matrix
ndgrid	Generate arrays for N-D functions and interpolation
ndims	Number of dimensions
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
norm	Vector and matrix norms
normest	2-norm estimate
numel	Number of elements in array or subscripted array
ones	Create a matrix of all 1s
pascal	Pascal matrix
permute	Rearrange dimensions of array
pinv	Pseudoinverse of a matrix
planerot	Givens plane rotation
prod	Product of array element
qr	Orthogonal-triangular decomposition

Function	Description
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed random numbers
randperm	Random permutation
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
repmat	Replicate and tile an array
reshape	Reshape one array into the dimensions of another
rng	Control random number generation
rosser	Classic symmetric eigenvalue test problem
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sign	Signum function
size	Return the size of a matrix
sort	Sort elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
sub2ind	Single index from subscripts
subspace	Angle between two subspaces
sum	Sum of matrix elements
toeplitz	Toeplitz matrix
trace	Sum of diagonal elements

Function	Description
tril	Extract lower triangular part
triu	Extract upper triangular part
true	Return an array of logical (Boolean) 1s for the specified dimensions
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix
zeros	Create a matrix of all zeros

Nonlinear Numerical Methods

Function	Description
fzero	Find root of continuous function of one variable
quad2d	Numerically evaluate double integral over planar region
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Polynomial Functions

Function	Description
poly	Polynomial with specified roots
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation
roots	Polynomial roots

Relational Operator Functions

Function	Description
eq	Equal (==)
ge	Greater than or equal to (>=)

Function	Description
gt	Greater than (>)
le	Less than or equal to (<=)
lt	Less than (<)
ne	Not equal (~=)

Rounding and Remainder Functions

Function	Description
ceil	Round toward plus infinity
ceil	Round toward positive infinity
convergent	Round toward nearest integer with ties rounding to nearest even integer
fix	Round toward zero
fix	Round toward zero
floor	Round toward minus infinity
floor	Round toward negative infinity
mod	Modulus (signed remainder after division)
nearest	Round toward nearest integer with ties rounding toward positive infinity
rem	Remainder after division
round	Round toward nearest integer
round	Round fi object toward nearest integer or round input data using quantizer object

Set Functions

Function	Description
intersect	Find set intersection of two vectors
ismember	Array elements that are members of set

Function	Description
issorted	Determine whether set elements are in sorted order
setdiff	Find set difference of two vectors
setxor	Find set exclusive OR of two vectors
union	Find set union of two vectors
unique	Find unique elements of vector

Signal Processing Functions in MATLAB

Function	Description
chol	Cholesky factorization
conv	Convolution and polynomial multiplication
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
filter	Filter a data sequence using a digital filter that works for both real and complex inputs
freqspace	Frequency spacing for frequency response
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse discrete Fourier transform shift

Function	Description
svd	Singular value decomposition
zp2tf	Convert zero-pole-gain filter parameters to transfer function form

Signal Processing Toolbox Functions

All of these functions require a DSP System Toolbox license to generate code. 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” in the Signal Processing Toolbox documentation.

Note Many Signal Processing Toolbox functions require constant inputs in generated code. To specify a constant input for `codegen`, use `coder.Constant`. For more information, see the “MATLAB Coder” documentation.

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.

Function	Remarks/Limitations
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.
buttord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cfirpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ap	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb1ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheb2ord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
chebwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby1	All Inputs must be constants. Expressions or variables are allowed if their values do not change.
cheby2	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
dct	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.
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.
fir1s	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpm	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firpmord	All inputs must be constants. Expressions or variables are allowed if their values do not change.
firrcos	All inputs must be constants. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
flattopwin	All inputs must be constants. Expressions or variables are allowed if their values do not change.
freqz	freqz with no output arguments produces a plot only when the function call terminates in a semicolon. See “freqz With No Output Arguments”.
gaussfir	All inputs must be constant. Expressions or variables are allowed if their values do not change.
gausswin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hamming	All inputs must be constant. Expressions or variables are allowed if their values do not change.
hann	All inputs must be constant. Expressions or variables are allowed if their values do not change.
idct	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.
intfilt	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiser	All inputs must be constant. Expressions or variables are allowed if their values do not change.
kaiserord	—
levinson	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.
rectwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.

Function	Remarks/Limitations
resample	The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.
sgolay	All inputs must be constant. Expressions or variables are allowed if their values do not change.
sosfilt	—
taylorwin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
triang	All inputs must be constant. Expressions or variables are allowed if their values do not change.
tukeywin	All inputs must be constant. Expressions or variables are allowed if their values do not change.
upfirdn	<ul style="list-style-type: none"> • Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change. • Variable-size inputs are not supported.
upsample	<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	If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

Special Values

Symbol	Description
eps	Floating-point relative accuracy
inf	IEEE® arithmetic representation for positive infinity
intmax	Largest possible value of specified integer type

Symbol	Description
intmin	Smallest possible value of specified integer type
NaN or nan	Not a number
pi	Ratio of the circumference to the diameter for a circle
realmax	Largest positive floating-point number
realmin	Smallest positive floating-point number

Specialized Math

Symbol	Description
beta	Beta function
betainc	Incomplete beta function
betaln	Logarithm of beta function
ellipke	Complete elliptic integrals of first and second kind
erf	Error function
erfc	Complementary error function
erfcinv	Inverse of complementary error function
erfcx	Scaled complementary error function
erfinv	Inverse error function
expint	Exponential integral
gamma	Gamma function
gammainc	Incomplete gamma function
gammaln	Logarithm of the gamma function

Statistical Functions

Function	Description
corrcoef	Correlation coefficients
mean	Average or mean value of array
median	Median value of array
mode	Most frequent values in array
std	Standard deviation
var	Variance

String Functions

Function	Description
bin2dec	Convert binary number string to decimal number
bitmax	Maximum double-precision floating-point integer
blanks	Create string of blank characters
char	Create character array (string)
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
hex2dec	Convert hexadecimal number string to decimal number
ischar	True for character array (string)
strcmp	Return a logical result for the comparison of two strings; limited to strings known at compile time

Structure Functions

Function	Description
isfield	Determine whether input is structure array field
struct	Create structure
isstruct	Determine whether input is a structure

Trigonometric Functions

Function	Description
acos	Inverse cosine
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse cosecant and inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atan2	Four quadrant inverse tangent
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent

Function	Description
cos	Cosine
cosd	Cosine; result in degrees
cosh	Hyperbolic cosine
cot	Cotangent; result in radians
cotd	Cotangent; result in degrees
coth	Hyperbolic cotangent
csc	Cosecant; result in radians
cscd	Cosecant; result in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant; result in radians
secd	Secant; result in degrees
sech	Hyperbolic secant
sin	Sine
sind	Sine; result in degrees
sinh	Hyperbolic sine
tan	Tangent
tand	Tangent; result in degrees
tanh	Hyperbolic tangent

System Objects Supported for Code Generation

- “About Code Generation for System Objects” on page 3-2
- “Computer Vision System Toolbox System Objects” on page 3-3
- “Communications System Toolbox System Objects” on page 3-8
- “DSP System Toolbox System Objects” on page 3-14

About Code Generation for System Objects

You can generate C/C++ code for a subset of System objects provided by Communications System Toolbox, DSP System Toolbox, and Computer Vision System Toolbox. To use these System objects, you need to install the requisite toolbox.

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 on MATLAB objects, see *Object-Oriented Programming* in the MATLAB documentation.

Computer Vision System Toolbox System Objects

If you install Computer Vision System Toolbox software, you can generate C/C++ code for the following Computer Vision System Toolbox System objects. For more information on how to use these System objects, see “Use System Objects for Code Generation from MATLAB” in the Computer Vision System Toolbox documentation.

Supported Computer Vision System Toolbox System Objects

Object	Description
Analysis & Enhancement	
<code>vision.BoundaryTracer</code>	Trace object boundaries in binary images
<code>vision.ContrastAdjuster</code>	Adjust image contrast by linear scaling
<code>vision.Deinterlacer</code>	Remove motion artifacts by deinterlacing input video signal
<code>vision.EdgeDetector</code>	Find edges of objects in images
<code>vision.ForegroundDetector</code>	Detect foreground using Gaussian Mixture Models. This object supports tunable properties in code generation.
<code>vision.HistogramEqualizer</code>	Enhance contrast of images using histogram equalization
<code>vision.TemplateMatcher</code>	Perform template matching by shifting template over image
Conversions	
<code>vision.Autothresher</code>	Convert intensity image to binary image
<code>vision.ChromaResampler</code>	Downsample or upsample chrominance components of images
<code>vision.ColorSpaceConverter</code>	Convert color information between color spaces
<code>vision.DemosaicInterpolator</code>	Demosaic Bayer’s format images
<code>vision.GammaCorrector</code>	Apply or remove gamma correction from images or video streams

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
vision.ImageComplementer	Compute complement of pixel values in binary, intensity, or RGB images
vision.ImageDataTypeConverter	Convert and scale input image to specified output data type
Feature Detection, Extraction, and Matching	
vision.CornerDetector	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
vision.Convolver	Compute 2-D discrete convolution of two input matrices
vision.ImageFilter	Perform 2-D FIR filtering of input matrix
vision.MedianFilter	2D median filtering
Geometric Transformations	
vision.GeometricRotator	Rotate image by specified angle
vision.GeometricScaler	Enlarge or shrink image size
vision.GeometricShearer	Shift rows or columns of image by linearly varying offset
vision.GeometricTransformer	Apply projective or affine transformation to an image
vision.GeometricTransformEstimator	Estimate geometric transformation from matching point pairs
vision.GeometricTranslator	Translate image in two-dimensional plane using displacement vector
Morphological Operations	

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.ConnectedComponentLabeler</code>	Label and count the connected regions in a binary image
<code>vision.MorphologicalClose</code>	Perform morphological closing on image
<code>vision.MorphologicalDilate</code>	Perform morphological dilation on an image
<code>vision.MorphologicalErode</code>	Perform morphological erosion on an image
<code>vision.MorphologicalOpen</code>	Perform morphological opening on an image
Object Detection	
<code>vision.HistogramBasedTracker</code>	Track object in video based on histogram. This object supports tunable properties in code generation
Sinks	
<code>vision.DeployableVideoPlayer</code>	Send video data to computer screen
<code>vision.VideoFileWriter</code>	Write video frames and audio samples to multimedia file
Sources	
<code>vision.VideoFileReader</code>	Read video frames and audio samples from compressed multimedia file
Statistics	
<code>vision.Autocorrelator</code>	Compute 2-D autocorrelation of input matrix
<code>vision.BlobAnalysis</code>	Compute statistics for connected regions in a binary image
<code>vision.Crosscorrelator</code>	Compute 2-D cross-correlation of two input matrices
<code>vision.Histogram</code>	Generate histogram of each input matrix

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.LocalMaximaFinder</code>	Find local maxima in matrices
<code>vision.Maximum</code>	Find maximum values in input or sequence of inputs
<code>vision.Mean</code>	Find mean value of input or sequence of inputs
<code>vision.Median</code>	Find median values in an input
<code>vision.Minimum</code>	Find minimum values in input or sequence of inputs
<code>vision.PSNR</code>	Compute peak signal-to-noise ratio (PSNR) between images
<code>vision.StandardDeviation</code>	Find standard deviation of input or sequence of inputs
<code>vision.Variance</code>	Find variance values in an input or sequence of inputs
Text & Graphics	
<code>vision.AlphaBlender</code>	Combine images, overlay images, or highlight selected pixels
<code>vision.MarkerInserter</code>	Draw markers on output image
<code>vision.ShapeInserter</code>	Draw rectangles, lines, polygons, or circles on images
<code>vision.TextInserter</code>	Draw text on image or video stream
Transforms	
<code>vision.DCT</code>	Compute 2-D discrete cosine transform
<code>vision.FFT</code>	Two-dimensional discrete Fourier transform
<code>vision.HoughLines</code>	Find Cartesian coordinates of lines that are described by rho and theta pairs
<code>vision.HoughTransform</code>	Find lines in images via Hough transform
<code>vision.IDCT</code>	Compute 2-D inverse discrete cosine transform

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.IFFT</code>	Two-dimensional inverse discrete Fourier transform
<code>vision.Pyramid</code>	Perform Gaussian pyramid decomposition
Utilities	
<code>vision.ImagePadder</code>	Pad or crop input image along its rows, columns, or both

Communications System Toolbox System Objects

If you install Communications System Toolbox software, you can generate C/C++ code for the following Communications System Toolbox System objects. For information on how to use these System objects, see “Code Generation with System Objects” in the Communications System Toolbox documentation.

Supported Communications System Toolbox System Objects

Object	Description
Source Coding	
comm.DifferentialDecoder	Decode binary signal using differential decoding
comm.DifferentialEncoder	Encode binary signal using differential coding
Channels	
comm.AWGNChannel	Add white Gaussian noise to input signal
comm.BinarySymmetricChannel	Introduce binary errors
Equalizers	
comm.MLSEEqualizer	Equalize using maximum likelihood sequence estimation
Filters	
comm.IntegrateAndDumpFilter	Integrate discrete-time signal with periodic resets
Measurements	
comm.EVM	Measure error vector magnitude
comm.MER	Measure modulation error ratio
Sources	
comm.BarkerCode	Generate Barker code
comm.HadamardCode	Generate Hadamard code
comm.KasamiSequence	Generate a Kasami sequence
comm.OVSFCode	Generate OVSF code
comm.PNSequence	Generate a pseudo-noise (PN) sequence

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.WalshCode	Generate Walsh code from orthogonal set of codes
Error Detection and Correction – Block Coding	
comm.BCHEncoder	Encode data using BCH encoder
comm.RSEncoder	Encode data using Reed-Solomon encoder
Error Detection and Correction – Convolutional Coding	
comm.ConvolutionalEncoder	Convolutionally encode binary data
comm.ViterbiDecoder	Decode convolutionally encoded data using Viterbi algorithm
Error Detection and Correction – Cyclic Redundancy Check Coding	
comm.CRCDetector	Detect errors in input data using cyclic redundancy code
comm.CRCGenerator	Generate cyclic redundancy code bits and append to input data
comm.HDLCRCGenerator	Generate CRC code bits and append to input data, optimized for HDL code generation
comm.TurboDecoder	Decode input signal using parallel concatenated decoding scheme
comm.TurboEncoder	Encode input signal using parallel concatenated encoding scheme
Interleavers – Block	
comm.AlgebraicDeinterleaver	Deinterleave input symbols using algebraically derived permutation vector
comm.AlgebraicInterleaver	Permute input symbols using an algebraically derived permutation vector
comm.BlockDeinterleaver	Deinterleave input symbols using permutation vector
comm.BlockInterleaver	Permute input symbols using a permutation vector

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.MatrixDeinterleaver	Deinterleave input symbols using permutation matrix
comm.MatrixInterleaver	Permute input symbols using permutation matrix
comm.MatrixHelicalScanDeinterleaver	Deinterleave input symbols by filling a matrix along diagonals
comm.MatrixHelicalScanInterleaver	Permute input symbols by selecting matrix elements along diagonals
Interleavers – Convolutional	
comm.ConvolutionalDeinterleaver	Restore ordering of symbols using shift registers
comm.ConvolutionalInterleaver	Permute input symbols using shift registers
comm.HelicalDeinterleaver	Restore ordering of symbols using a helical array
comm.HelicalInterleaver	Permute input symbols using a helical array
comm.MultiplexedDeinterleaver	Restore ordering of symbols using a set of shift registers with specified delays
comm.MultiplexedInterleaver	Permute input symbols using a set of shift registers with specified delays
MIMO	
comm.OSTBCCombiner	Combine inputs using orthogonal space-time block code
comm.OSTBCEncoder	Encode input message using orthogonal space-time block code
Digital Baseband Modulation – Phase	
comm.BPSKDemodulator	Demodulate using binary PSK method
comm.BPSKModulator	Modulate using binary PSK method
comm.DBPSKModulator	Modulate using differential binary PSK method
comm.DPSKDemodulator	Demodulate using M-ary DPSK method
comm.DPSKModulator	Modulate using M-ary DPSK method

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.DQPSKDemodulator	Demodulate using differential quadrature PSK method
comm.DQPSKModulator	Modulate using differential quadrature PSK method
comm.DBPSKDemodulator	Demodulate using M-ary DPSK method
comm.QPSKDemodulator	Demodulate using quadrature PSK method
comm.QPSKModulator	Modulate using quadrature PSK method
comm.PSKDemodulator	Demodulate using M-ary PSK method
comm.PSKModulator	Modulate using M-ary PSK method
comm.OQPSKDemodulator	Demodulate offset quadrature PSK modulated data
comm.OQPSKModulator	Modulate using offset quadrature PSK method
Digital Baseband Modulation – Amplitude	
comm.GeneralQAMDemodulator	Demodulate using arbitrary QAM constellation
comm.GeneralQAMModulator	Modulate using arbitrary QAM constellation
comm.PAMDemodulator	Demodulate using M-ary PAM method
comm.PAMModulator	Modulate using M-ary PAM method
comm.RectangularQAMDemodulator	Demodulate using rectangular QAM method
comm.RectangularQAMModulator	Modulate using rectangular QAM method
Digital Baseband Modulation – Frequency	
comm.FSKDemodulator	Demodulate using M-ary FSK method
comm.FSKModulator	Modulate using M-ary FSK method
Digital Baseband Modulation – Trellis Coded	
comm.GeneralQAMTCMDemodulator	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation
comm.GeneralQAMTCMModulator	Convolutionally encode binary data and map using arbitrary QAM constellation

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.PSKTCMDemodulator	Demodulate convolutionally encoded data mapped to M-ary PSK constellation
comm.PSKTCModulator	Convolutionally encode binary data and map using M-ary PSK constellation
comm.RectangularQAMTCMDemodulator	Demodulate convolutionally encoded data mapped to rectangular QAM constellation
comm.RectangularQAMTCModulator	Convolutionally encode binary data and map using rectangular QAM constellation
Digital Baseband Modulation – Continuous Phase	
comm.CPFSKDemodulator	Demodulate using CPFSK method and Viterbi algorithm
comm.CPFSKModulator	Modulate using CPFSK method
comm.CPMDemodulator	Demodulate using CPM method and Viterbi algorithm
comm.CPMModulator	Modulate using CPM method
comm.GMSKDemodulator	Demodulate using GMSK method and the Viterbi algorithm
comm.GMSKModulator	Modulate using GMSK method
comm.MSKDemodulator	Demodulate using MSK method and the Viterbi algorithm
comm.MSKModulator	Modulate using MSK method
RF Impairments	
comm.PhaseFrequencyOffset	Apply phase and frequency offsets to input signal
comm.ThermalNoise	Add receiver thermal noise
Synchronization – Timing Phase	
comm.EarlyLateGateTimingSynchronizer	Recover symbol timing phase using early-late gate method

Supported Communications System Toolbox System Objects (Continued)

Object	Description
comm.GardnerTimingSynchronizer	Recover symbol timing phase using Gardner's method
comm.GMSKTimingSynchronizer	Recover symbol timing phase using fourth-order nonlinearity method
comm.MSKTimingSynchronizer	Recover symbol timing phase using fourth-order nonlinearity method
comm.MuellerMullerTimingSynchronizer	Recover symbol timing phase using Mueller-Muller method
Synchronization Utilities	
comm.DiscreteTimeVCO	Generate variable frequency sinusoid
Converters	
comm.BitToInteger	Convert vector of bits to vector of integers
comm.IntegerToBit	Convert vector of integers to vector of bits
Sequence Operators	
comm.Descrambler	Descramble input signal
comm.Scrambler	Scramble input signal

DSP System Toolbox System Objects

If you install DSP System Toolbox software, you can generate C/C++ code for the following DSP System Toolbox System objects. For information on how to use these System objects, see “Code Generation with System Objects” in the DSP System Toolbox documentation.

Supported DSP System Toolbox System Objects

Object	Description
Estimation	
<code>dsp.BurgAREstimator</code>	Compute estimate of autoregressive model parameters using Burg method
<code>dsp.BurgSpectrumEstimator</code>	Compute parametric spectral estimate using Burg method
<code>dsp.CepstralToLPC</code>	Convert cepstral coefficients to linear prediction coefficients
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion
<code>dsp.LPCToAutocorrelation</code>	Convert linear prediction coefficients to autocorrelation coefficients
<code>dsp.LPCToCepstral</code>	Convert linear prediction coefficients to cepstral coefficients
<code>dsp.LPCToLSF</code>	Convert linear prediction coefficients to line spectral frequencies
<code>dsp.LPCToLSP</code>	Convert linear prediction coefficients to line spectral pairs
<code>dsp.LPCToRC</code>	Convert linear prediction coefficients to reflection coefficients
<code>dsp.LSFToLPC</code>	Convert line spectral frequencies to linear prediction coefficients
<code>dsp.LSPToLPC</code>	Convert line spectral pairs to linear prediction coefficients

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.RCToAutocorrelation</code>	Convert reflection coefficients to autocorrelation coefficients
<code>dsp.RCToLPC</code>	Convert reflection coefficients to linear prediction coefficients
Filters	
<code>dsp.BiquadFilter</code>	Model biquadratic IIR (SOS) filters
<code>dsp.DigitalFilter</code>	Filter each channel of input over time using discrete-time filter implementations
<code>dsp.FIRDecimator</code>	Filter and downsample input signals
<code>dsp.FIRFilter</code>	Static or time-varying FIR filter
<code>dsp.FIRInterpolator</code>	Upsample and filter input signals
<code>dsp.FIRRateConverter</code>	Upsample, filter and downsample input signals
<code>dsp.LMSFilter</code>	Compute output, error, and weights using LMS adaptive algorithm
Math Operations	
<code>dsp.ArrayVectorAdder</code>	Add vector to array along specified dimension
<code>dsp.ArrayVectorDivider</code>	Divide array by vector along specified dimension
<code>dsp.ArrayVectorMultiplier</code>	Multiply array by vector along specified dimension
<code>dsp.ArrayVectorSubtractor</code>	Subtract vector from array along specified dimension
<code>dsp.CumulativeProduct</code>	Compute cumulative product of channel, column, or row elements
<code>dsp.CumulativeSum</code>	Compute cumulative sum of channel, column, or row elements
<code>dsp.LDLFactor</code>	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.LowerTriangularSolver</code>	Solve $LX = B$ for X when L is lower triangular matrix
<code>dsp.LUFactor</code>	Factor square matrix into lower and upper triangular matrices
<code>dsp.Normalizer</code>	Normalize input
<code>dsp.UpperTriangularSolver</code>	Solve $UX = B$ for X when U is upper triangular matrix
Quantizers	
<code>dsp.ScalarQuantizerDecoder</code>	Convert each index value into quantized output value
<code>dsp.ScalarQuantizerEncoder</code>	Perform scalar quantization encoding
<code>dsp.VectorQuantizerDecoder</code>	Find vector quantizer codeword for given index value
<code>dsp.VectorQuantizerEncoder</code>	Perform vector quantization encoding
Signal Management	
<code>dsp.Counter</code>	Count up or down through specified range of numbers
<code>dsp.DelayLine</code>	Rebuffer sequence of inputs with one-sample shift
Signal Operations	
<code>dsp.Convolver</code>	Compute convolution of two inputs
<code>dsp.Delay</code>	Delay input by specified number of samples or frames
<code>dsp.Interpolator</code>	Interpolate values of real input samples
<code>dsp.NCO</code>	Generate real or complex sinusoidal signals
<code>dsp.PeakFinder</code>	Determine extrema (maxima or minima) in input signal
<code>dsp.PhaseUnwrapper</code>	Unwrap signal phase
<code>dsp.VariableFractionalDelay</code>	Delay input by time-varying fractional number of sample periods
<code>dsp.VariableIntegerDelay</code>	Delay input by time-varying integer number of sample periods
<code>dsp.Window</code>	Generate or apply window function

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.ZeroCrossingDetector	Calculate number of zero crossings of a signal
Sinks	
dsp.AudioPlayer	Write audio data to computer's audio device
dsp.AudioFileWriter	Write audio file
dsp.UDPsender	Send UDP packets to the network
Sources	
dsp.AudioFileReader	Read audio samples from an audio file
dsp.AudioRecorder	Read audio data from computer's audio device
dsp.UDPReceiver	Receive UDP packets from the network
dsp.SineWave	Generate discrete sine wave
Statistics	
dsp.Autocorrelator	Compute autocorrelation of vector inputs
dsp.Crosscorrelator	Compute cross-correlation of two inputs
dsp.Histogram	Output histogram of an input or sequence of inputs
dsp.Maximum	Compute maximum value in input
dsp.Mean	Compute average or mean value in input
dsp.Median	Compute median value in input
dsp.Minimum	Compute minimum value in input
dsp.RMS	Compute root-mean-square of vector elements
dsp.StandardDeviation	Compute standard deviation of vector elements
dsp.Variance	Compute variance of input or sequence of inputs
Transforms	
dsp.AnalyticSignal	Compute analytic signals of discrete-time inputs

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.DCT	Compute discrete cosine transform (DCT) of input
dsp.FFT	Compute fast Fourier transform (FFT) of input
dsp.IDCT	Compute inverse discrete cosine transform (IDCT) of input
dsp.IFFT	Compute inverse fast Fourier transform (IFFT) of input

Defining MATLAB Variables for C/C++ Code Generation

- “Why Define Variables Differently for Code Generation?” on page 4-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 4-3
- “When You Can Reassign Variable Properties for C/C++ Code Generation” on page 4-7
- “Eliminating Redundant Copies of Variables in Generated Code” on page 4-8
- “Defining and Initializing Persistent Variables” on page 4-10
- “Reusing the Same Variable with Different Properties” on page 4-11
- “Avoiding Overflows in for-Loops” on page 4-16
- “Supported Variable Types” on page 4-19

Why Define Variables Differently 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
coder.extrinsic('disp');
if(c>0)
    x = int8(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
coder.extrinsic('disp');
x = zeros(1,3);
if(c>0)
    x = int8(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 4-3.

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

In this section...

“Define Variables By Assignment Before Using Them” on page 4-3

“Use Caution When Reassigning Variables” on page 4-6

“Use Type Cast Operators in Variable Definitions” on page 4-6

“Define Matrices Before Assigning Indexed Variables” on page 4-6

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 all execution paths during C/C++ code generation (see Example: Defining a Variable for Multiple Execution Paths on page 4-4).

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 Example: Defining All Fields in a Structure on page 4-5).

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 “Eliminating Redundant Copies of Variables in Generated Code” on page 4-8.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see “Defining and Initializing Persistent Variables” on page 4-10.

Example: 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
...
```


Example: Defining All Fields in a Structure

Consider the following MATLAB code:

```
...
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    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 Chapter 6, “Code Generation for MATLAB Structures”.

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 “When You Can Reassign Variable Properties for C/C++ Code Generation” on page 4-7.

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); % z 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 any of 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 “Limitations on Matrix Indexing Operations for Code Generation” on page 8-41.

When You Can Reassign 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 “How Working with Variable-Size Data Is Different for Code Generation” on page 8-3.

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 “Reusing the Same Variable with Different Properties” on page 4-11.

Eliminating Redundant Copies of Variables in Generated Code

In this section...

“When Redundant Copies Occur” on page 4-8

“How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 4-8

“Defining Uninitialized Variables” on page 4-9

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 all 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 4-8..

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
```

Defining and Initializing 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
```

For more information, see [Persistent Variables in the MATLAB Programming Fundamentals documentation](#).

Reusing the Same Variable with Different Properties

In this section...

“When You Can Reuse the Same Variable with Different Properties” on page 4-11

“When You Cannot Reuse Variables” on page 4-12

“Limitations of Variable Reuse” on page 4-14

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” in the MATLAB Coder documentation).

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](#) on page 4-12.

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.

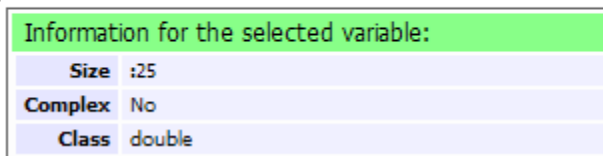
```
% First time t is used to hold a scalar double value.
t = mean(mean(u)) / numel(u);
u = u - t;
```

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 all 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.varsizes`.
- 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.

Avoiding 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>

Loop conditions causing the error	Workaround
<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 end .</pre>
<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 	<p>Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.</p>

Loop conditions causing the error	Workaround
<p>Note The software error checking might be too conservative and report the possibility of an infinite under these circumstances even though an infinite loop would never occur.</p>	

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	Signed integer
logical	Boolean true or false
single	Single-precision floating point
struct	Structure (see Chapter 6, “Code Generation for MATLAB Structures”)
uint8, uint16, uint32	Unsigned integer
Fixed-point	See “Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms” in the Fixed-Point Toolbox User’s Guide documentation.

Defining Data for Code Generation

- “How Working with Data is Different for Code Generation” on page 5-2
- “Code Generation for Complex Data” on page 5-4
- “Code Generation for Characters” on page 5-6

How Working with Data is Different for Code Generation

To generate efficient standalone code, you must use 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
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 always evaluate to a complex result, even if the result is zero <hr/> <p>Note Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic</p> <hr/>	“Code Generation for Complex Data” on page 5-4
Characters	Restricted to 8 bits of precision	“Code Generation for Characters” on page 5-6

Data	What's Different	More Information
Enumerated data	<ul style="list-style-type: none">• Supports integer-based enumerated types only• Restricted use in switch statements and for-loops	Chapter 7, “Code Generation for Enumerated Data”
Function handles	<ul style="list-style-type: none">• Function handles must be scalar values• Same bound variable cannot reference different function handles• Cannot pass function handles to or from primary or extrinsic functions• Cannot view function handles from the debugger	Chapter 10, “Code Generation for Function Handles”

Code Generation for Complex Data

In this section...

“Restrictions When Defining Complex Variables” on page 5-4

“Expressions Containing Complex Operands Yield Complex Results” on page 5-5

Restrictions When Defining Complex Variables

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

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

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

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

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

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

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

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

Expressions Containing Complex Operands Yield Complex Results

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

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

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

There are two exceptions to this behavior:

- Functions that take complex arguments, but produce real results

```
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:

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

Code Generation for 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 generate code from your MATLAB algorithm.

Code Generation for MATLAB Structures

- “How Working with Structures Is Different for Code Generation” on page 6-2
- “Structure Operations Allowed for Code Generation” on page 6-3
- “Defining Scalar Structures for Code Generation” on page 6-4
- “Defining Arrays of Structures for Code Generation” on page 6-7
- “Making Structures Persistent” on page 6-9
- “Indexing Substructures and Fields” on page 6-10
- “Assigning Values to Structures and Fields” on page 6-12
- “Passing Large Structures as Input Parameters” on page 6-13

How Working with Structures Is Different 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 6-3
Observe restrictions on properties and values of scalar structures.	“Defining Scalar Structures for Code Generation” on page 6-4
Make structures uniform in arrays.	“Defining Arrays of Structures for Code Generation” on page 6-7
Reference structure fields individually during indexing.	“Indexing Substructures and Fields” on page 6-10
Avoid type mismatch when assigning values to structures and fields.	“Assigning Values to Structures and Fields” on page 6-12

For an introduction to working with structures in MATLAB, see “Structures” in the MATLAB documentation.

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 subfunctions

Defining Scalar Structures for Code Generation

In this section...

“Restrictions When Using struct” on page 6-4

“Restrictions When Defining Scalar Structures by Assignment” on page 6-4

“Adding Fields in Consistent Order on Each Control Flow Path” on page 6-4

“Restriction on Adding New Fields After First Use” on page 6-5

Restrictions When Using struct

When you use the `struct` function to create scalar structures for code generation, the following restrictions apply:

- Field arguments must be scalar values.
- You cannot create structures of cell arrays.

Restrictions When Defining Scalar Structures by Assignment

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

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

Adding Fields in Consistent Order on Each Control Flow Path

When you create a structure, you must add fields in the same order on each control flow path. For example, the following code generates a compiler error because it adds the fields of structure `x` in a different order in each `if` statement clause:

```

function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.b = 30; % Generates an error (on variable x)
    x.a = 40;
end
y = x.a + x.b;

```

In this example, the assignment to `x.a` comes before `x.b` in the first `if` statement clause, but the assignments appear in reverse order in the `else` clause. Here is the corrected code:

```

function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.a = 40;
    x.b = 30;
end
y = x.a + x.b;

```

Restriction on Adding New Fields After First Use

You cannot add fields to a structure after you perform any of 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.

Defining Arrays of Structures for Code Generation

In this section...

“Ensuring Consistency of Fields” on page 6-7

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 6-7

“Defining an Array of Structures Using Concatenation” on page 6-8

Ensuring Consistency of Fields

When you create an array of MATLAB structures with the intent of generating code, you must be sure that each structure field in the array has the same size, type, and complexity.

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 “Defining Scalar Structures for Code Generation” on page 6-4.
- 2 Call repmat, passing the scalar structure and the dimensions of the array.
- 3 Assign values to each structure using standard array indexing and structure dot notation.

For example, the following code creates X, a 1-by-3 array of scalar structures. Each element of the array is defined by the structure s, which has two fields, a and b:

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,3);
X(1).a = 1;
```

```
X(2).a = 2;  
X(3).a = 3;  
X(1).b = 4;  
X(2).b = 5;  
X(3).b = 6;  
...
```

Defining an Array of Structures Using Concatenation

To create a small array of structures, you can use the concatenation operator, square brackets (`[]`), to join one or more structures into an array (see “Concatenating Matrices” in the MATLAB documentation). For code generation, all the structures that you concatenate must have the same size, class, and complexity.

For example, the following code uses concatenation and a subfunction 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;  
...
```

Making Structures Persistent

To make structures persist, you define them to be persistent variables and initialize them with the `isempty` statement, as described in “Defining and Initializing Persistent Variables” on page 4-10.

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
```

Indexing 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 “Defining Arrays of Structures for Code Generation” on page 6-7 for more information.

Do not reference fields dynamically

You cannot reference fields in a structure by using dynamic names, which express the field as a variable expression that MATLAB evaluates at run time (see “Generate Field Names from Variables” in the MATLAB documentation).

Assigning Values to Structures and Fields

Use these guidelines when assigning values to a structure, substructure, or field for code generation:

Field properties must be consistent across structure-to-structure assignments

If:	Then:
Assigning one structure to another structure.	Define each structure with the same number, type, and size of fields.
Assigning one structure to a substructure of a different structure and vice versa.	Define the structure with the same number, type, and size of fields as the substructure.
Assigning an element of one structure to an element of another structure.	The elements must have the same type and size.

Do not use field values as constants

The values stored in the fields of a structure are not treated as constant values in generated code. Therefore, you cannot use field values to set the size or class of other data. For example, the following code generates a compiler error:

```
...
Y.a = 3;
X = zeros(Y.a); % Generates an error
```

In this example, even though you set field `a` of structure `Y` to the value `3`, `Y.a` is not a constant in generated code and, therefore, it is not a valid argument to pass to the function `zeros`.

Do not assign mxArray to structures

You cannot assign `mxArrays` to structure elements; convert `mxArrays` to known types before code generation (see “Working with `mxArrays`” on page 12-16).

Passing Large Structures as Input Parameters

If you generate a MEX function for a MATLAB function that takes a large structure as an input parameter, for example, a structure containing fields that are matrices, the MEX function might fail to load. This load failure occurs because, when you generate a MEX function from a MATLAB function that has input parameters, the code generation software allocates memory for these input parameters on the stack. To avoid this issue, pass the structure by reference to the MATLAB function. For example, if the original function signature is:

```
y = foo(a, S)
```

where S is the structure input, rewrite the function to:

```
[y, S] = foo(a, S)
```


Code Generation for Enumerated Data

- “How Working with Enumerated Data Is Different for Code Generation” on page 7-2
- “Enumerated Types Supported for Code Generation” on page 7-3
- “When to Use Enumerated Data for Code Generation” on page 7-6
- “Workflows for Using Enumerated Data for Code Generation” on page 7-7
- “How to Define Enumerated Data for Code Generation” on page 7-9
- “How to Instantiate Enumerated Types for Code Generation” on page 7-11
- “How to Generate Code for Enumerated Data” on page 7-12
- “Defining and Using Enumerated Types for Code Generation” on page 7-13
- “Operations on Enumerated Data Allowed for Code Generation” on page 7-15
- “Using Enumerated Data in Control Flow Statements” on page 7-18
- “Customizing Enumerated Data Types” on page 7-24
- “Changing and Reloading Enumerated Data Types” on page 7-32
- “Restrictions on Use of Enumerated Data in for-Loops” on page 7-33
- “Toolbox Functions That Support Enumerated Types for Code Generation” on page 7-34

How Working with Enumerated Data Is Different for Code Generation

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

What's Different	More Information
Supports integer-based enumerated types only	“Enumerated Types Supported for Code Generation” on page 7-3
Name of each enumerated data type must be unique	“Naming Enumerated Types for Code Generation” on page 7-10
Each enumerated data type must be defined in a separate file on the MATLAB path	“How to Define Enumerated Data for Code Generation” on page 7-9 and “How to Generate Code for Enumerated Data” on page 7-12
Restricted set of operations	“Operations on Enumerated Data Allowed for Code Generation” on page 7-15
Restricted use in for-loops	“Restrictions on Use of Enumerated Data in for-Loops” on page 7-33

See Also

- “How to Define Enumerated Data for Code Generation” on page 7-9
- “Defining and Organizing Classes” in the MATLAB Object-Oriented Programming documentation for more information about defining MATLAB classes
- “Enumerations and Modeling” for more information about enumerated types based on `Simulink.IntEnumType`

Enumerated Types Supported for Code Generation

In this section...

“Enumerated Type Based on int32” on page 7-3

“Enumerated Type Based on Simulink.IntEnumType” on page 7-4

Enumerated Type Based on int32

This enumerated data type is based on the built-in type int32. Use this enumerated type when generating code from MATLAB algorithms.

Syntax

```
classdef(Enumeration) type_name < int32
```

Example

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

In this example, the statement `classdef(Enumeration) PrimaryColors < int32` means that the enumerated type `PrimaryColors` is based on the built-in type `int32`. As such, `PrimaryColors` inherits the characteristics of the `int32` type, as well as defining 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

How to Use

Define enumerated data in MATLAB code and compile the source file. For example, to generate C/C++ code from your MATLAB source, you can use `codegen`, as described in “Workflow for Generating Code for Enumerated Data from MATLAB Algorithms” on page 7-7.

Note `codegen` requires a MATLAB Coder license.

Enumerated Type Based on `Simulink.IntEnumType`

This enumerated data type is based on the built-in type `Simulink.IntEnumType`, which is available with a Simulink license. Use this enumerated type when exchanging enumerated data with Simulink blocks and Stateflow charts.

Syntax

```
classdef(Enumeration) type_name < Simulink.IntEnumType
```

Example

```
classdef(Enumeration) myMode < Simulink.IntEnumType
    enumeration
        OFF(0)
        ON(1)
    end
end
```

How to Use

Here are the basic guidelines for using enumerated data based on `Simulink.IntEnumType`:

Application	What to Do
When exchanging enumerated data with Simulink blocks	Define enumerated data in MATLAB Function blocks in Simulink models. Requires Simulink software.
When exchanging enumerated data with Stateflow charts	Define enumerated data in MATLAB functions in Stateflow charts. Requires Simulink and Stateflow software.

For more information, see:

- “Using Enumerated Data in MATLAB Function Blocks” in the Simulink documentation
- “Using Enumerated Data in Stateflow Charts” in the Stateflow documentation

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 need to restrict data to a finite set of values and refer to these values by name. Even though you can sometimes achieve these goals by using integers or strings, enumerated types offer the following advantages:

- Provide more readable code than integers
- Allow 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.

- Produce more efficient code than strings

For example, comparisons of enumerated values execute faster than comparisons of strings.

Workflows for Using Enumerated Data for Code Generation

In this section...

“Workflow for Generating Code for Enumerated Data from MATLAB Algorithms” on page 7-7

“Workflow for Generating Code for Enumerated Data from MATLAB Function Blocks” on page 7-8

Workflow for Generating Code for Enumerated Data from MATLAB Algorithms

Step	Action	How?
1	Define an enumerated data type that inherits from <code>int32</code> .	See “How to Define Enumerated Data for Code Generation” on page 7-9.
2	Instantiate the enumerated type in your MATLAB algorithm.	See “How to Instantiate Enumerated Types for Code Generation” on page 7-11.
3	Compile the function with <code>codegen</code> .	See “How to Generate Code for Enumerated Data” on page 7-12.

This workflow requires a MATLAB Coder license.

Workflow for Generating Code for Enumerated Data from MATLAB Function Blocks

Step	Action	How?
1	Define an enumerated data type that inherits from <code>Simulink.IntEnumType</code> .	See “How to Define Enumerated Data Types for MATLAB Function Blocks” in the Simulink documentation.
2	Add the enumerated data to your MATLAB Function block.	See “How to Add Enumerated Data to MATLAB Function Blocks” in the Simulink documentation.
3	Instantiate the enumerated type in your MATLAB Function block.	See “How to Instantiate Enumerated Data in MATLAB Function Blocks” in the Simulink documentation.
4	Simulate and/or generate code.	See “Enumerations” in the Simulink Coder documentation.

This workflow requires the following licenses:

- Simulink (for simulation)
- MATLAB Coder and Simulink Coder (for code generation)

How to Define Enumerated Data for Code Generation

Follow these to define enumerated data for code generation from MATLAB algorithms:

- 1 Create a class definition file.

In the MATLAB Command Window, select **File > New > Class**.

- 2 Enter the class definition as follows:

```
classdef(Enumeration) EnumTypeName < int32
```

For example, the following code defines an enumerated type called `sysMode`:

```
classdef(Enumeration) sysMode < int32  
    ...  
end
```

EnumTypeName is a case-sensitive string that must be unique among data type names and workspace variable names. It must inherit from the built-in type `int32`.

- 3 Define enumerated values in an enumeration section as follows:

```
classdef(Enumeration) EnumTypeName < int32  
    enumeration  
        EnumName(N)  
        ...  
    end  
end
```

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

```
classdef(Enumeration) sysMode < int32  
    enumeration  
        OFF(0)  
        ON(1)  
    end  
  
end
```

An enumerated type can define any number of values. Each enumerated value consists of a string *EnumName* and an underlying integer *N*. Each *EnumName* must be unique within its type, but can also appear in other enumerated types. The underlying integers need not be either consecutive or ordered, nor must they be unique within the type 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.

To add a folder to the MATLAB search path, type `addpath pathname` at the MATLAB command prompt. For more information, see “Using the MATLAB Search Path”, `addpath`, and `savepath` in the MATLAB documentation.

For examples of enumerated data type definitions, see “Class Definition: `sysMode`” on page 7-13 and “Class Definition: `LEDcolor`” on page 7-14.

Naming Enumerated Types for Code Generation

You must use a unique name for each enumerated data type. The name of an enumerated data type cannot match the name of a toolbox function supported for code generation, or another data type or a variable in the MATLAB base workspace. Otherwise, a name conflict occurs.

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 Chapter 2, “Functions Supported for Code Generation”.

How to Instantiate Enumerated Types for Code Generation

To instantiate an enumerated type for code generation from MATLAB algorithms, use dot notation to specify *ClassName.EnumName*. For example, the following function `displayState` instantiates the enumerated types `sysMode` and `LEDcolor` from “Defining and Using Enumerated Types for Code Generation” on page 7-13. The dot notation appears highlighted in the code.

```
function led = displayState(state)
    %#codegen

    if state == sysMode.ON
        led = LEDcolor.GREEN;
    else
        led = LEDcolor.RED;
    end
```

How to Generate Code for Enumerated Data

Use the command `codegen` to generate MEX, C, or C++ code from the MATLAB algorithm that contains the enumerated data (requires a MATLAB Coder license). Each enumerated data type must be defined on the MATLAB path in a separate file as a class derived from the built-in type `int32`. See “How to Define Enumerated Data for Code Generation” on page 7-9.

If your function has inputs, you must specify the properties of these inputs to `codegen`. For an enumerated data input, use the `-args` option to pass one of its allowable values as a sample value. For example, the following `codegen` command specifies that the function `displayState` takes one input of enumerated data type `sysMode`.

```
codegen displayState -args {sysMode.ON}
```

After executing this command, `codegen` generates a platform-specific MEX function that you can test in MATLAB. For example, to test `displayState`, type the following command:

```
displayState(sysMode.OFF)
```

You should get the following result:

```
ans =
```

```
    RED
```

See Also

- MATLAB Coder documentation to learn more about `codegen`
- “Defining and Using Enumerated Types for Code Generation” on page 7-13 for a description of the example function `displayState` and its enumerated type definitions

Defining and Using Enumerated Types for Code Generation

In this section...

“About the Example” on page 7-13

“Class Definition: sysMode” on page 7-13

“Class Definition: LEDcolor” on page 7-14

“Function: displayState” on page 7-14

About the Example

The following example appears throughout this section to illustrate how to define and use enumerated types for code generation. The function, `displayState` uses enumerated data to represent the modes of a device that controls the colors of an LED display.

Before using enumerated data, you must define your enumerated data types as MATLAB classes that inherit from the built-in type `int32`. Each class definition must reside in a separate file on the MATLAB path. This example uses two enumerated types: `sysMode` to represent the set of allowable modes and `LEDcolor` to represent the set of allowable colors.

See Also

- “Workflows for Using Enumerated Data for Code Generation” on page 7-7
- “How to Define Enumerated Data for Code Generation” on page 7-9
- “How to Instantiate Enumerated Types for Code Generation” on page 7-11
- “How to Generate Code for Enumerated Data” on page 7-12

Class Definition: `sysMode`

Here is the class definition of the `sysMode` enumerated data type:

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0)
        ON(1)
```

```
    end  
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `sysMode.m`.

Class Definition: LEDcolor

Here is the class definition of the LEDcolor enumerated data type:

```
classdef(Enumeration) LEDcolor < int32  
    enumeration  
        GREEN(1),  
        RED(2),  
    end  
end
```

This definition must reside on the MATLAB path in a file called `LEDcolor.m`.

Function: displayState

The following function `displayState` 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
```

This function models a simple control.

The compiler directive `%#codegen` indicates that you intend to generate code from the MATLAB algorithm. See “Adding the Compilation Directive `%#codegen`” on page 12-8.

Operations on Enumerated Data Allowed for Code Generation

To generate efficient standalone code for enumerated data, you are restricted to the following operations. The examples are based on the definitions of the enumeration type `LEDcolor` described in “Class Definition: `LEDcolor`” on page 7-14.

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
<code>double(LEDcolor.RED)</code>	ans = 2
<code>z = 2</code> <code>y = LEDcolor(z)</code>	z = 2 y = RED

Indexing Operation

Example	Result
<code>m = [1 2]</code> <code>n = LEDcolor(m)</code> <code>p = n(LEDcolor.GREEN)</code>	m = 1 2 n = GREEN RED p = GREEN

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>	<p>“Using the if Statement on Enumerated Data Types” on page 7-18</p>
switch	<pre> 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 </pre>	<p>“Using the switch Statement on Enumerated Data Types” on page 7-19</p>
while	<pre> while state ~= State.Ready switch state case State.Standby initialize(); state = State.Boot; case State.Boot boot(); state = State.Ready; end end </pre>	<p>“Using the while Statement on Enumerated Data Types” on page 7-22</p>

Using Enumerated Data in Control Flow Statements

The following control statements work with enumerated operands in generated code. However, there are restrictions (see “Restrictions on Use of Enumerated Data in for-Loops” on page 7-33).

Using the `if` Statement on Enumerated Data Types

This example is based on the definition of the enumeration types `LEDcolor` and `sysMode`. The function `displayState` uses these enumerated data types to activate an LED display.

Class Definition: `sysMode`

```
classdef(Enumeration) sysMode < int32
    enumeration
        OFF(0)
        ON(1)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `sysMode.m`.

Class Definition: `LEDcolor`

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
end
```

This definition must reside on the MATLAB path in a file called `LEDcolor.m`.

MATLAB Function: `displayState`

This function uses enumerated data to activate an LED display, based on the state of a device. It lights a green LED display to indicate the ON state and lights a red LED display to indicate the OFF state.

```
function led = displayState(state)
%#codegen

if state == sysMode.ON
    led = LEDcolor.GREEN;
else
    led = LEDcolor.RED;
end
```

Build and Test a MEX Function for displayState

- 1 Generate a MEX function for displayState. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen displayState -args {sysMode.ON}
```

- 2 Test the function. For example,

```
displayState(sysMode.OFF)
```

You should get the following result:

```
ans =

    RED
```

Using the switch Statement on Enumerated Data Types

This example is based on the definition of the enumeration types `VCRState` and `VCRButton`. The function `VCR` uses these enumerated data types to set the state of the VCR.

Class Definition: VCRState

```
classdef(Enumeration) VCRState < int32
    enumeration
        Stop(0),
        Pause(1),
```

```
        Play(2),  
        Forward(3),  
        Rewind(4)  
    end  
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRState.m`.

Class Definition: VCRButton

```
classdef(Enumeration) VCRButton < int32  
    enumeration  
        Stop(1),  
        PlayOrPause(2),  
        Next(3),  
        Previous(4)  
    end  
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `VCRButton.m`.

MATLAB Function: VCR

This function uses enumerated data to set the state of a VCR, based on the initial state of the VCR and the state of the VCR button.

```
function s = VCR(button)  
%#codegen  
  
persistent state  
  
if isempty(state)  
    state = VCRState.Stop;  
end  
  
switch state  
    case {VCRState.Stop, VCRState.Forward, VCRState.Rewind}  
        state = handleDefault(button);  
end
```



```

    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)
```

You should get the following result:

```
s =

    Stop
```

Using the while Statement on Enumerated Data Types

This example is based on the definition of the enumeration type `State`. The function `Setup` uses this enumerated data type to set the state of a device.

Class Definition: `State`

```
classdef(Enumeration) State < int32
    enumeration
        Standby(0),
        Boot(1),
        Ready(2)
    end
end
```

This definition must reside on the MATLAB path in a file with the same name as the class, `State.m`.

MATLAB Function: `Setup`

The following function `Setup` uses enumerated data to set the state of a device.

```
function s = Setup(initState)
%#codegen

state = initState;

if isempty(state)
    state = State.Standby;
end

while state ~= State.Ready
    switch state
        case State.Standby
            initialize();
            state = State.Boot;
        case State.Boot
            boot();
            state = State.Ready;
    end
end
```

```
s = state;

function initialize()
% Perform initialization.

function boot()
% Boot the device.
```

Build and Test a MEX Executable for Setup

- 1 Generate a MEX executable for Setup. Use the `-args` option to pass one of the allowable values for the enumerated data input as a sample value.

```
codegen Setup -args {State.Standby}
```

- 2 Test the function. For example,

```
s = Setup(State.Standby)
```

You should get the following result:

```
s =

    Ready
```

Customizing Enumerated Data Types

In this section...
“Customizing Enumerated Types Based on int32” on page 7-24
“Customizing Enumerated Types Based on Simulink.IntEnumType” on page 7-31

Customizing Enumerated Types Based on int32

About Customizing Enumerated Types

You can customize an enumerated type by using the same techniques that work with MATLAB classes, as described in [Modifying Superclass Methods and Properties](#). A primary source of customization are the methods associated with an enumerated type.

Enumerated class definitions can include an optional methods section. 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.	true — prefix is used	If you do not want the class name to become a prefix in the generated code, override this method to set the return value to false. See “Controlling the Names of Enumerated Type Values in Generated Code” on page 7-26.
getDefaultValue	Returns the default enumerated value.	''	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 “Specifying a Default Enumerated Value” on page 7-28.

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 “Specifying a Header File” on page 7-29

Controlling the Names of Enumerated Type Values in Generated Code

This example shows how to control the name of enumerated type values in code generated by MATLAB Coder. (Requires a MATLAB Coder license.)

- 1 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`.

- 2 Click the *View Report* link.
- 3 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.

```
typedef enum LEDcolor
{
    LEDcolor_GREEN = 1,
    LEDcolor_RED
} LEDcolor;
```

The enumerated value names include the class name prefix LEDcolor_.

- 4** Modify the definition of LEDcolor to override the addClassNameToEnumNames method. Set the return value to false instead of true so that the enumerated value names in the generated code do not contain the class prefix.

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end

    methods(Static)
        function y=addClassNameToEnumNames()
            y=false;
        end
    end
end
```

- 5** Clear existing class instances:

```
clear classes
```

- 6** Generate code again.

```
codegen -config:lib -report displayState -args {sysMode.ON}
```

- 7** Open the code generation report and look at the enumerated type definition in displayState_types.h.

```
typedef enum LEDcolor
{
```

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

This time the enumerated value names do not include the class name prefix.

See Also.

- MATLAB Coder documentation to learn more about `codegen`
- “Defining and Using Enumerated Types for Code Generation” on page 7-13 for a description of the example function `displayState` and its enumerated type definitions

Specifying a Default Enumerated Value

The code generation software and related generated code use an enumerated data type’s default value when you provide no other initial 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 desired default. `ThisClass` must be the name of the class within which the method exists. `EnumName` must be the name of an enumerated value defined in that class. For example:

```
classdef(Enumeration) LEDcolor < int32  
    enumeration  
        GREEN(1),  
        RED(2),  
    end
```



```

        methods (Static)
        function y = getDefaultvalue()
            y = LEDcolor.RED;
        end
    end
end

```

This example defines the default as `LEDcolor.RED`. If this method does not appear, the default value would be `LEDcolor.GREEN`, because that is the first value listed in the enumerated class definition.

Specifying 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's 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 any legal filename for `filename`. Be sure to provide a filename suffix, typically `.h`. Providing the method replaces the declaration that would otherwise have appeared in the generated code with a `#include` statement like:

```
#include "imported_enum_type.h"
```

The `getHeaderFile` method does not create the declaration file itself. You must provide a file of the specified name that declares the enumerated data type. The file can also contain definitions of enumerated types that you do not use in your MATLAB code.

For example, to use the definition of `LEDcolor` in `my_LEDcolor.h`:

- 1 Modify the definition of `LEDcolor` to override the `getHeaderFile` method to return the name of the external header file:

```
classdef(Enumeration) LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end

    methods(Static)
        function y=getHeaderFile()
            y='my_LEDcolor.h';
        end
    end
end
```

- 2** In the current folder, provide a header file, `my_LEDcolor.h`, that contains the definition:

```
typedef enum LEDcolor
{
    GREEN = 1,
    RED
} 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.

Customizing Enumerated Types Based on Simulink.IntEnumType

You can customize a Simulink enumerated type by using the same techniques that work with MATLAB classes, as described in *Modifying Superclass Methods and Properties*. For more information, see “Customizing a Simulink Enumeration”.

Changing and Reloading 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 any 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...
Do one of the following: <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, you can cast the enumerated data to `int32` in the loop counter.

For example, suppose you define an enumerated type `ColorCodes` as follows:

```
classdef(Enumeration) ColorCodes < int32
    enumeration
        Red(1),
        Blue(2),
        Green(3)
        Yellow(4)
        Purple(5)
    end
end
```

Because the enumerated values are consecutive, you can use `ColorCodes` data in a for-loop like this:

```
...
for i = int32(ColorCodes.Red):int32(ColorCodes.Purple)
    c = ColorCodes(i);
    ...
end
```

Toolbox Functions That Support Enumerated Types for Code Generation

The following MATLAB toolbox functions support enumerated types for code generation:

- `cast`
- `cat`
- `circshift`
- `flipdim`
- `fliplr`
- `flipud`
- `histc`
- `ipermute`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `shiftdim`
- `sort`
- `sortrows`

- squeeze

Code Generation for Variable-Size Data

- “What Is Variable-Size Data?” on page 8-2
- “How Working with Variable-Size Data Is Different for Code Generation” on page 8-3
- “Bounded Versus Unbounded Variable-Size Data” on page 8-4
- “How to Control Memory Allocation of Variable-Size Data” on page 8-5
- “How to Generate Code for MATLAB Functions with Variable-Size Data” on page 8-6
- “Generating Code for a MATLAB Function That Expands a Vector in a Loop” on page 8-8
- “Using Variable-Size Data Without Dynamic Memory Allocation” on page 8-16
- “Variable-Size Data in Code Generation Reports” on page 8-20
- “Defining Variable-Size Data for Code Generation” on page 8-22
- “C Code Interface for Arrays” on page 8-29
- “Troubleshooting Issues with Variable-Size Data” on page 8-33
- “Limitations with Variable-Size Support for Code Generation” on page 8-37
- “Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation” on page 8-43

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
```

How Working with Variable-Size Data Is Different for Code Generation

In the MATLAB language, all 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. You can then generate standalone code for bounded and unbounded variable-size data.

See Also

- “Bounded Versus Unbounded Variable-Size Data” on page 8-4
- “Related Products That Support Code Generation from MATLAB” on page 1-12

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 “How to Control Memory Allocation of Variable-Size Data” on page 8-5.

How to Control Memory Allocation of Variable-Size Data

All data whose size exceeds the dynamic memory allocation threshold is allocated on the heap. The default dynamic memory allocation threshold is 64 kilobytes. All data whose size is less than this threshold is allocated on the stack.

Dynamic memory allocation is an expensive operation; the performance cost may 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 “Controlling Dynamic Memory Allocation”.

You can control memory allocation globally for your application by modifying the dynamic memory allocation threshold. See “Generating Code for a MATLAB Function That Expands a Vector in a Loop” on page 8-8. You can control memory allocation for individual variables by specifying upper bounds. See “Specifying Upper Bounds for Variable-Size Data” on page 8-16.

How to Generate 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. Code generation requires a MATLAB Coder license.

To work through these steps with a simple example, see “Generating Code for a MATLAB Function That Expands a Vector in a Loop” on page 8-8.

- 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 (requires a MATLAB Coder license). 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 myFcn -args {coder.typeof(0,[2 4],1)}
```

This command uses `coder.typeof` to specify one variable-size input for function `myFcn`. 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 matrix is variable-size. 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.	Define the data to be variable sized.	See “Diagnosing and Fixing Size Mismatch Errors” on page 8-33

5 Generate C/C++ code using the `codegen` function (requires a MATLAB Coder license).

Generating Code for a MATLAB Function That Expands a Vector in a Loop

In this section...

“About the MATLAB Function `uniquetol`” on page 8-8

“Step 1: Add Compilation Directive for Code Generation” on page 8-8

“Step 2: Address Issues Detected by the Code Analyzer” on page 8-9

“Step 3: Generate MEX Code” on page 8-9

“Step 4: Fix the Size Mismatch Error” on page 8-11

“Step 5: Generate C Code” on page 8-13

“Step 6: Change the Dynamic Memory Allocation Threshold” on page 8-14

About the MATLAB Function `uniquetol`

This example uses the function `uniquetol`. This function returns in vector `B` a version of input vector `A`, where the elements are unique to within tolerance `tol` of each other. In vector `B`, $\text{abs}(B(i) - B(j)) > \text{tol}$ for all `i` and `j`. Initially, assume input vector `A` can store up to 100 elements.

```
function B = uniquetol(A, tol)
A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

Step 1: Add Compilation Directive for Code Generation

Add the `codegen` compilation directive at the top of the function:

```
function B = uniquetol(A, tol) %codegen
```



```

A = sort(A);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end

```

Step 2: Address Issues Detected by the Code Analyzer

The Code Analyzer detects that variable B might change size in the for-loop. It issues this warning:

The variable 'B' appears to change size on every loop iteration. Consider preallocating for speed.

In this function, vector B should expand in size as it adds values from vector A. Therefore, you can ignore this warning.

Step 3: Generate MEX Code

To generate MEX code, use the `codegen` function.

1 Generate a MEX function for `uniquetol`:

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

What do these command-line options mean?

The `-args` option specifies the class, complexity, and size of each input to function `uniquetol`:

- The first argument, `coder.typeof`, defines a variable-size input. The expression `coder.typeof(0,[1 100],1)` defines input A as a real double vector with a fixed upper bound. Its first dimension is fixed at 1 and its second dimension can vary in size up to 100 elements.

For more information, see “Specifying Variable-Size Inputs at the Command Line” in the MATLAB Coder documentation.

- 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, even if no errors or warnings occur.

For more information, see the `codegen` reference page.

Executing this command generates a compiler error:

```
??? Size mismatch (size [1 x 1] ~= size [1 x 2]).  
The size to the left is the size  
of the left-hand side of the assignment.
```

- 2 Open the error report and select the **Variables** tab.

```

1 function B = uniquetol(A, tol) %#codegen
2 A = sort(A);
3 B = A(1);
4 k = 1;
5 for i = 2:length(A)
6     if abs(A(k) - A(i)) > tol
7         B = [B A(i)];
8         k = i;
9     end
10 end

```

Order	Variable	Type	Size	Complex	Class
1	B	Output	1 x 1	No	double
2	A > 1	Input	1 x :100	No	double
3	A > 2	Local	1 x :?	No	double
4	tol	Input	1 x 1	No	double
5	k	Local	1 x 1	No	double
6	i	Local	1 x 1	No	double

The error indicates a size mismatch between the left-hand side and right-hand side of the assignment statement `B = [B A(i)]`. The assignment `B = A(1)` establishes the size of `B` as a fixed-size scalar (1 x 1). Therefore, the concatenation of `[B A(i)]` creates a 1 x 2 vector.

Step 4: Fix the Size Mismatch Error

To fix this error, declare `B` to be a variable-size vector.

1 Add this statement to the `uniquetol` function:

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

It should appear before `B` is used (read). For example:

```
function B = uniquetol(A, tol) %#codegen
```

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

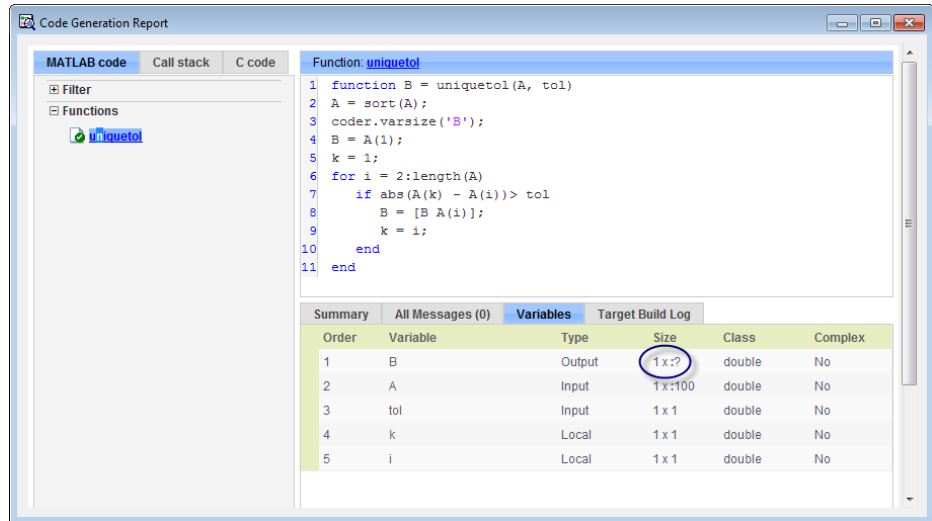
The function `coder.varsize` declares every instance of `B` in `uniquetol` to be variable sized.

- 2 Generate code again using the same command:

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

In the current folder, `codegen` generates a MEX function for `uniquetol` and provides a link to the code generation report.

- 3 Click the *View report* link.
- 4 In the code generation report, select the **Variables** tab.



The size of variable B is 1x?, indicating that it is variable size with no upper bounds.

Step 5: Generate C Code

Generate C code for variable-size inputs. By default, codegen allocates memory statically for any data whose size is less than the dynamic memory allocation threshold of 64 kilobytes. If the size of the data exceeds the threshold or is unbounded, codegen allocates memory dynamically on the heap.

- 1 Create a configuration option for C library generation:

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

- 2 Issue this command:

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

codegen generates a static library in the default location, codegen\lib\uniquetol and provides a link to the code generation report.

- 3** Click the *View report* link.
- 4** In the code generation report, click the **C code** tab.
- 5** On the **C code** tab, click the link to `uniquetol.h`.

The function declaration is:

```
extern void uniquetol(const real_T A_data[100], const int32_T A_size[2],...  
    real_T 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 64 kilobytes, allocates this memory statically. The generated code contains two pieces of information about A:

- `real_T A_data[100]`: the maximum size of input A (where 100 is the maximum size specified using `coder.typeof`).
- `int32_T A_sizes[2]`: the actual size of the input.

Because B is variable size with unknown upper bounds, in the generated code, codegen represents B as `emxArray_real_T`. MATLAB provides utility functions for creating and interacting with `emxArrays` in your generated code. For more information, see “C Code Interface for Arrays” on page 8-29.

Step 6: Change the Dynamic Memory Allocation Threshold

In this step, you reduce the dynamic memory allocation threshold and generate code for an input that exceeds this threshold.

- 1** Set the dynamic memory allocation threshold to 4 kilobytes and generate code where the size of input A exceeds this threshold.

```
cfg.DynamicMemoryAllocationThreshold=4096;  
codegen -config cfg -report uniquetol -args {coder.typeof(0,[1 10000],1),coder.typeof(0)}
```

- 2 View the generated code in the report. Because the maximum size of input A now exceeds the dynamic memory allocation threshold, codegen allocates A dynamically on the heap and represents A as `emxArray_real_T`.

```
extern void uniquetol(const emxArray_real_T *A, ...
    real_T tol, emxArray_real_T *B);
```

Using Variable-Size Data Without Dynamic Memory Allocation

In this section...

“Fixing Upper Bounds Errors” on page 8-16

“Specifying Upper Bounds for Variable-Size Data” on page 8-16

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 8-16 and “Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 8-35

Specifying Upper Bounds for Variable-Size Data

- “When to Specify Upper Bounds for Variable-Size Data” on page 8-16
- “Specifying Upper Bounds on the Command Line for Variable-Size Inputs” on page 8-16
- “Specifying Unknown Upper Bounds for Variable-Size Inputs” on page 8-17
- “Specifying Upper Bounds for Local Variable-Size Data” on page 8-17
- “Using a Matrix Constructor with Nonconstant Dimensions” on page 8-18

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` in the MATLAB Coder documentation.

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 all 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” on page 8-17
- “Specifying the Upper Bounds for All Instances of a Local Variable” on page 8-18

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

```
function Y = example_bounds1(u) %#codegen
Y = [1 2 3 4 5];
coder. varsize('Y', [1 10]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

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

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

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

Using a Matrix Constructor with Nonconstant Dimensions

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

```
function y = var_by_assign(u) %#codegen
if (u > 0)
```

```
    y = ones(3,u);  
else  
    y = zeros(3,1);  
end
```

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

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

Variable-Size Data in Code Generation Reports

In this section...
“What Reports Tell You About Size” on page 8-20
“How Size Appears in Code Generation Reports” on page 8-21
“How to Generate a Code Generation Report” on page 8-21

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 (requires a MATLAB Coder license).

Defining Variable-Size Data for Code Generation

In this section...
“When to Define Variable-Size Data Explicitly” on page 8-22
“Using a Matrix Constructor with Nonconstant Dimensions” on page 8-23
“Inferring Variable Size from Multiple Assignments” on page 8-23
“Defining Variable-Size Data Explicitly Using <code>coder.versize</code> ” on page 8-24

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 8-23
Assign multiple, constant sizes to the same variable before using (reading) the variable.	“Inferring Variable Size from Multiple Assignments” on page 8-23
Define all instances of a variable to be variable sized	“Defining Variable-Size Data Explicitly Using <code>coder.versize</code> ” on page 8-24

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
```

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 all variable-size data is unbounded.

Example: 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 8-25). 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 8-26).

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 any dimension for which $\text{size}(A, \text{dim}) = 1$. 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.versize` to specify explicitly that singleton dimensions vary. For example:

```
function Y = dim_singleton_vary(u) %#codegen
Y = [1 2];
coder.versize('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.versize` is a vector of ones, indicating that each dimension of `Y` varies in size. For more information, see the `coder.versize` 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.versize('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.ysize('data(:).values')` defines the field values inside each element of matrix `data` to be variable sized.

Here are other examples:

- `coder.ysize('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.ysize('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 8-29

“C Code Interface for Dynamically Allocated Arrays” on page 8-30

“Utility Functions for Creating emxArray Data Structures” on page 8-31

C Code Interface for Statically Allocated Arrays

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

For example, consider the MATLAB function `uniquetol`:

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

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

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

In the generated code, the function declaration is:

```
extern void uniquetol(const real_T A_data[100], const int32_T A_size[2],...
    real_T tol, emxArray_real_T *B);
```

There are two pieces of information about A:

- `real_T A_data[100]`: the maximum size of input A (where 100 is the maximum size specified using `coder.typeof`).
- `int32_T A_sizes[2]`: the actual size of the input.

C Code Interface for Dynamically Allocated Arrays

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

`emxArray` Structure Definition

```
typedef struct emxArray_<baseTypeName>
{
    <baseTypeName> *data;
    int32_T *size;
    int32_T allocated;
    int32_T numDimensions;
    boolean_T canFreeData;
} emxArray_<baseTypeName>;
```

For example, here's the definition for an `emxArray` of base type `real_T` with unknown upper bounds:

```
typedef struct emxArray_real_T
{
    real_T *data;
    int32_T *size;
    int32_T allocated;
    int32_T numDimensions;
    boolean_T canFreeData;
} emxArray_real_T;
```

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

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

See Also.

- “Generating Code for a MATLAB Function That Expands a Vector in a Loop” on page 8-8

C Code Interface for Structure Fields

Field	Description
*data	Pointer to data of type <i><baseTypeName></i>
*size	Pointer to first element of size vector. Length of the vector equals the number of dimensions.
allocatedSize	Number of elements currently allocated for the array. If the size changes, MATLAB reallocates memory based on the new size.
numDimensions	Number of dimensions of the size vector, that is, the number of dimensions you can access without crossing into unallocated or unused memory
canFreeData	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”.

Function	Arguments	Description
<code>emxArray_<baseTypeName></code> <code>*emxCreateWrapper_<baseTypeName></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 never inadvertently free user memory, such as the stack.
<code>emxArray_<baseTypeName></code> <code>*emxCreateWrapperND_<baseTypeName></code> <code>(...)</code>	<code>*data</code> <code>numDimensions</code> <code>*size</code>	Same as <code>emxCreateWrapper</code> , except it creates a new N-dimensional <code>emxArray</code> .
<code>emxArray_<baseTypeName></code> <code>*emxCreate_<baseTypeName> (...)</code>	<code>num_rows</code> <code>num_cols</code>	Creates a new two-dimensional <code>emxArray</code> on the heap, initialized to zero. All data elements have the data type specified by <i>baseTypeName</i> .
<code>emxArray_<baseTypeName></code> <code>*emxCreateND_<baseTypeName> (...)</code>	<code>numDimensions</code> <code>*size</code>	Same as <code>emxCreate</code> , except it creates a new N-dimensional <code>emxArray</code> on the heap.
<code>emxArray_<baseTypeName></code> <code>*emxDestroyArray_<baseTypeName></code> <code>(...)</code>	<code>*emxArray</code>	Frees dynamic memory allocated by <code>*emxCreate</code> and <code>*emxCreateND</code> functions.

Troubleshooting Issues with Variable-Size Data

In this section...

“Diagnosing and Fixing Size Mismatch Errors” on page 8-33

“Diagnosing and Fixing Errors in Detecting Upper Bounds” on page 8-35

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

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

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

Compiling this function produces this error:

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

There are several ways to fix this error:

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

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

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

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

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

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

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

Empty Matrix Reshaped to Match Variable-Size Specification

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

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

```

if u < 0
    Y = [Y u];
end

```

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

Performing Binary Operations on Fixed and Variable-Size Operands

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

```

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

```

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

```

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

```

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

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

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

However, compiling this function generates an error because you did not specify an upper bound for `u`. There are several ways to fix the problem:

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

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

Limitations with Variable-Size Support for Code Generation

In this section...

“Limitation on Scalar Expansion” on page 8-37

“Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays” on page 8-38

“Incompatibility with MATLAB in Determining Size of Empty Arrays” on page 8-39

“Limitation on Vector-Vector Indexing” on page 8-40

“Limitations on Matrix Indexing Operations for Code Generation” on page 8-41

“Dynamic Memory Allocation Not Supported for MATLAB Function Blocks” on page 8-42

Limitation on Scalar Expansion

Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. During code generation, the standard MATLAB scalar expansion rules apply except when adding two variable-size expressions. In this case, both operands must be the same size. MATLAB does not perform scalar expansion even if one of the variable-size expressions is scalar. Instead, it generates a size mismatch error at run time.

For example, the following code applies the standard MATLAB scalar expansion rules:

```
function y = scalar_exp_test()%#codegen
A = zeros(2,2);
coder.varsize('A');
B = 1;
y = A + B;
```

It determines that B is scalar and adds it to the variable-size matrix A to produce this result:

```
ans =
```

```
1    1
1    1
```

However, suppose **B** is also variable sized:

```
function y = scalar_exp_test_err1()%#codegen
A = zeros(2,2);
coder.varsize('A','B');
B = 1;
y = A + B;
```

In this case, the `coder.varsize` statement obscures the fact that **B** is scalar. The function compiles, but generates a run-time error:

```
??? Sizes mismatch: [2][2] ~= [1][1]
```

Workaround

To fix the problem, use indexing to force **B** to be a scalar value:

```
function y = scalar_exp_test_fix()%#codegen
A = zeros(2,2);
coder.varsize('A','B');
B = 1;
y = A + B(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)` always 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)` always 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
```

Limitation on 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 $? \times ?$) 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 in simulation.

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.

Limitations on Matrix Indexing Operations for Code Generation

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

- Initialization of the following style:

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

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

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

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

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

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

During code generation, memory is never dynamically allocated for the size of the expressions that change as the program executes. To implement this behavior, use `for`-loops as shown in the following example:

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

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

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.

Restrictions on Variable Sizing in Toolbox Functions Supported for Code Generation

In this section...

“Common Restrictions” on page 8-43

“Toolbox Functions with Variable Sizing Restrictions” on page 8-44

Common Restrictions

The following common restrictions apply to multiple toolbox functions supported 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 8-44.

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 all scalars as fixed size.

Toolbox Functions with Variable Sizing Restrictions

Function	Restrictions with Variable-Size Data
all	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 8-43. • 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 8-43. • 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.

Function	Restrictions with Variable-Size Data
conv	<ul style="list-style-type: none"> • See “Variable-length vector restriction” on page 8-43. • Input vectors must have the same orientation, either both row vectors or both column vectors.
cov	<ul style="list-style-type: none"> • For cov(X), see “Array-to-vector restriction” on page 8-44.
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 8-43.
detrend	<ul style="list-style-type: none"> • For first argument for row vectors only, see “Array-to-vector restriction” on page 8-44 .
diag	<ul style="list-style-type: none"> • See “Array-to-vector restriction” on page 8-44 .
diff	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 8-43. • 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, diff(x,2,1) works but diff(x,5,1) generates a run-time error.
fft	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 8-43.

Function	Restrictions with Variable-Size Data
filter	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 8-43. • See “Automatic dimension restriction” on page 8-43.
hist	<ul style="list-style-type: none"> • For second argument, see “Variable-length vector restriction” on page 8-43. • For second input argument, see “Array-to-scalar restriction” on page 8-44.
histc	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 8-43.
ifft	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 8-43.
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 8-44. • 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 8-43.

Function	Restrictions with Variable-Size Data
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 8-43.
mean	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 8-43.• An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
median	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 8-43.• 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 8-43.
mode	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 8-43.• 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
mtimes	<ul style="list-style-type: none"> • When an input is variable sized, 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 happens to be a scalar at run time.
nchoosek	<ul style="list-style-type: none"> • Inputs must be fixed sized. • Second input must be a constant for static allocation. If you enable dynamic allocation, 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 8-43.
polyfit	<ul style="list-style-type: none"> • For first and second arguments, see “Variable-length vector restriction” on page 8-43.

Function	Restrictions with Variable-Size Data
prod	<ul style="list-style-type: none"> • See “Automatic dimension restriction” on page 8-43. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
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.
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"> • 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 8-43.

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 always 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 always 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 8-43. • 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 8-43. • 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 8-43. • 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
typecast	<ul style="list-style-type: none">• See “Variable-length vector restriction” on page 8-43 on first argument.
var	<ul style="list-style-type: none">• See “Automatic dimension restriction” on page 8-43.• An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

Code Generation for MATLAB Classes

- “About Code Generation for MATLAB Classes” on page 9-2
- “How Working with MATLAB Classes is Different for Code Generation” on page 9-3
- “Memory Allocation Requirements” on page 9-9
- “Generate Code for MATLAB Value Classes” on page 9-10
- “Generate Code for MATLAB Handle Classes and System Objects” on page 9-16
- “MATLAB Classes in Code Generation Reports” on page 9-19
- “Troubleshooting” on page 9-22

About Code Generation for MATLAB Classes

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 9-10
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 9-16

See Also

- “MATLAB Classes Overview”
- “How Working with MATLAB Classes is Different for Code Generation” on page 9-3

How Working with MATLAB Classes is Different for Code Generation

To generate efficient standalone code for MATLAB classes, you must use classes differently than you normally would when running your code in the MATLAB environment.

What's Different	More Information
Class must be in a single file. Because of this limitation, there is no code generation support for a class definition that uses an @-folder.	“Creating a Single, Self-Contained Class Definition File”
Restricted set of language features.	“Language Limitations” on page 9-3
Restricted set of code generation features.	“Code Generation Features Not Compatible with Classes” on page 9-5
Definition of class properties.	“Defining Class Properties for Code Generation” on page 9-6
Use of handle classes.	“Generate Code for MATLAB Handle Classes and System Objects” on page 9-16
Calls to base class constructor.	“Calls to Base Class Constructor” on page 9-7

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, all 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`
- `findprop`
- `ge`
- `gt`
- `invalid`
- `le`
- `lt`
- `ne`
- `notify`

- Diamond inheritance. If classes `B` and `C` both inherit from the same class and class `D` inherits from both class `B` and `C`, you cannot generate code for class `D`.

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}
```

- You cannot generate code for a value class that has a `set.prop` method. For example, you cannot generate code for the following `Square` class because of the `set.side` method.

```
classdef Square < Shape %#codegen
    properties
        side;
    end
    methods
        function obj = Square(side)
            obj = obj@Shape(side^2);
            obj.side = side;
        end
        function set.side(obj,value)
            obj.side = value;
            obj.area = value^2;
        end
    end
end
```

To generate code for this class, modify the class definition to remove the `set.side` method.

- 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.

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:

- If a class has a property of handle type, set the property in the class constructor. For System objects, you can also use the `setupImpl` method.
- 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 any 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 any properties, you must explicitly define the class, size, and complexity of all properties. For more information, see Chapter 4, “Defining MATLAB Variables for C/C++ Code Generation”.

- Initial values:
 - If the property has no 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 has no initial value and the code generation software cannot determine that the property is assigned on all paths 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 any 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` is always `''`.

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 be before any `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
```

Memory Allocation Requirements

When you create a handle object, you must assign the object to a persistent variable or to a property of another MATLAB object that must also be a persistent variable. The assignment must be in an `if - isempty` clause. After assignment, you can copy the object to a local variable, pass it to or return it from another function. For more information, see “Generate Code for MATLAB Handle Classes and System Objects” on page 9-16.

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
    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
methods(Abstract = true)
    getarea(obj);
end
methods(Static)
    function d = distanceBetweenShapes(shape1,shape2)
        xDist = abs(shape1.centerX - shape2.centerX);
        yDist = abs(shape1.centerY - shape2.centerY);
        d = sqrt(xDist^2 + yDist^2);
    end
end
end
```

- 2 In the same folder, create a class, `Square`, that is a subclass of `Shape`. Save the code as `Square.m`.

```

classdef Square < Shape
% Create a Square at coordinates center X and center Y
% with sides of length of side
    properties
        side;
    end
    methods
        function obj = Square(side,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.side = side;
        end
        function Area = getarea(obj)
            Area = obj.side^2;
        end
    end
end

```

- 3** In the same folder, create a class, Rhombus, that is a subclass of Shape. Save the code as Rhombus.m.

```

classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)
            Area = 0.5*obj.diag1*obj.diag2;
        end
    end
end

```

- 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 all the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties, as shown for `obj>1`. To view the complete list of properties, expand the list as shown for `obj>2`.

The screenshot shows the 'Code Generation Report' window. On the left, there is a 'Filter' section with 'Filter functions and methods' checked, and a 'Classes' tree showing 'Rhombus' selected. The main area displays the MATLAB code for the 'Rhombus' class, including properties, methods, and a constructor. Below the code is a 'Summary' table with columns for Order, Variable, Type, Size, Class, and Complex.

```

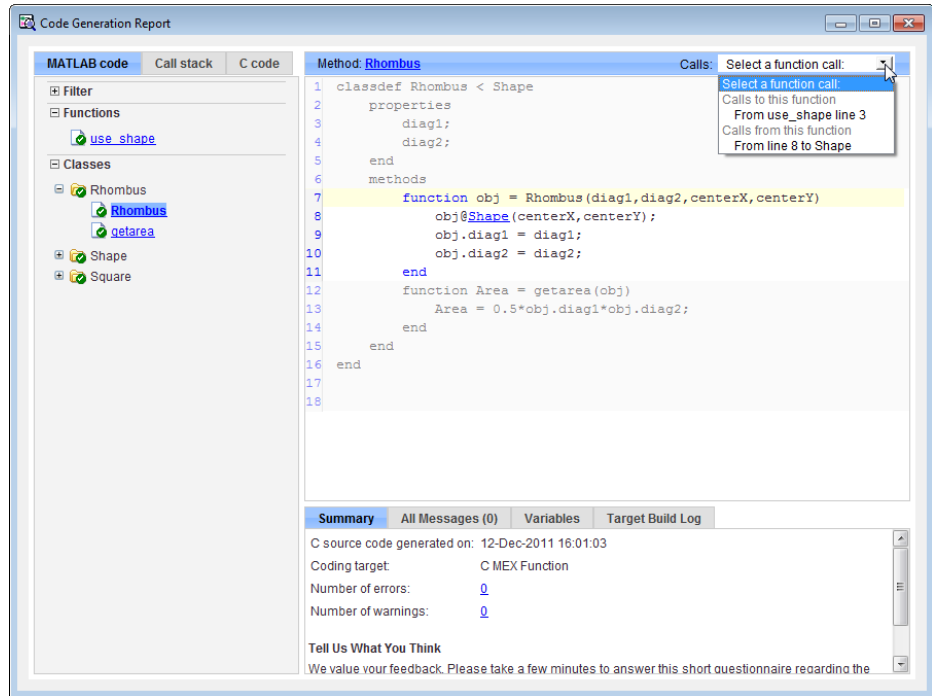
1 classdef Rhombus < Shape
2     properties
3         diag1;
4         diag2;
5     end
6     methods
7         function obj = Rhombus(diag1,diag2,centerX,centerY)
8             obj@Shape(centerX,centerY);
9             obj.diag1 = diag1;
10            obj.diag2 = diag2;
11        end
12        function Area = getarea(obj)
13            Area = 0.5*obj.diag1*obj.diag2;
14        end
15    end
16 end
17
18

```

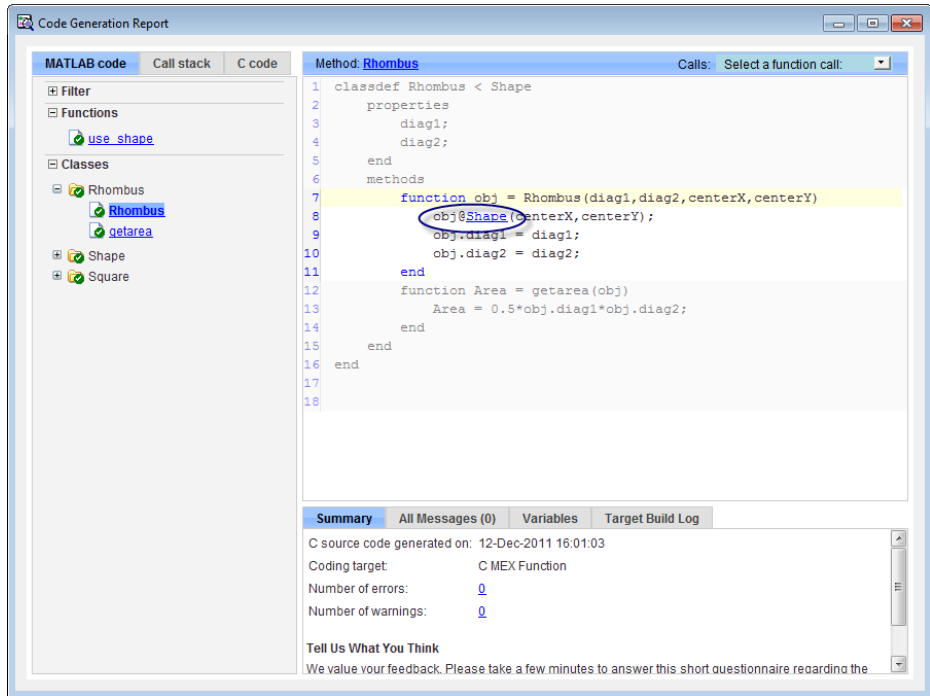
Order	Variable	Type	Size	Class	Complex
1	obj > 1	Output	1 x 1	Rhombus	-
2	obj > 2	Local	1 x 1	Rhombus	-
2.1	centerX	Property	1 x 1	double	No
2.2	centerY	Property	1 x 1	double	No
2.3	diag1	Property	1 x 1	double	No
2.4	diag2	Property	1 x 1	double	No
3	diag1	Input	1 x 1	double	No
4	diag2	Input	1 x 1	double	No
5	centerX	Input	1 x 1	double	No
6	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.



- 9 The constructor for the Rhombus class calls the Shape method of the base Shape class: `obj@Shape`. In the report, click the Shape link in this call.



The link takes you to the Shape method in the Shape class definition.

Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report. When you create a System or handle object, you must assign the object to a persistent variable or to a property of another MATLAB object that must also be a persistent variable. The assignment must be in an `if isempty` clause. After assignment, you can copy the object to a local variable, pass it to or return it from another function.

- 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
    persistent p;
    if isempty(p)
        p = AddOne();
    end
    y = p.step(x);
end
```

For code generation, you must immediately assign a System object to a persistent variable in an `if isempty` clause as in this example.

- 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.

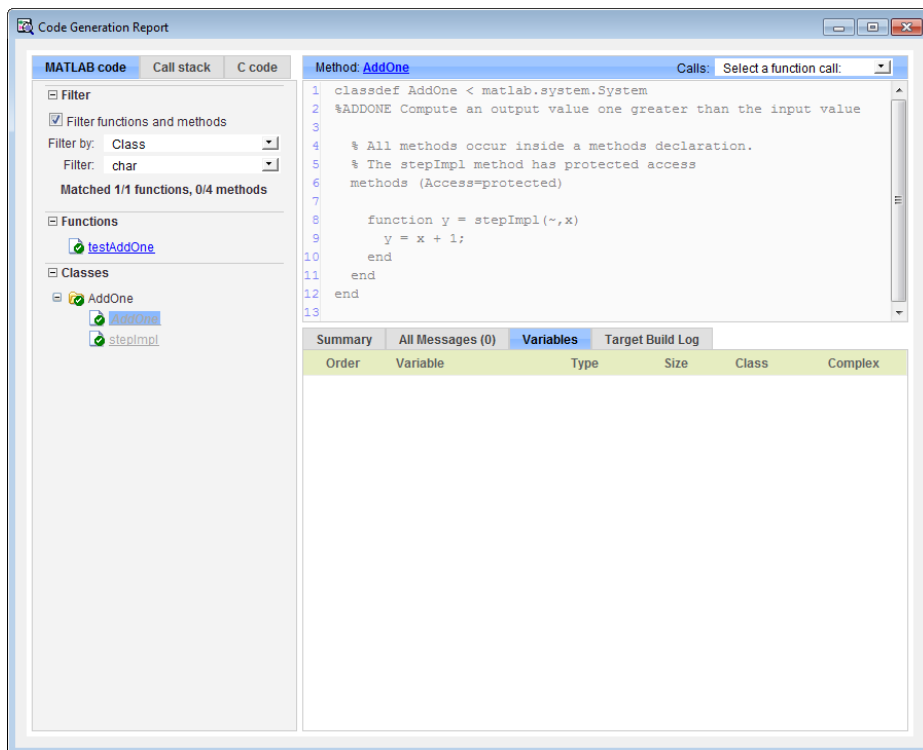
```
>> codegen -report testAddOne -args {0}
Code generation successful: View report
```

- 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.

The screenshot shows the Code Generation Report window. The left sidebar shows the project structure with 'testAddOne' selected under 'Functions'. The main area displays the MATLAB code for the function `testAddOne`. A tooltip is visible over the variable `p` in the code, showing its properties: Size 1 x 1 and Class AddOne. Below the code, the 'Variables' tab is active, displaying a table of variables.

Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	No
2	x	Input	1 x 1	double	No
3	p	Persistent	1 x 1	AddOne	-
3.1	isInitialized	Property	1 x 1	logical	-
3.2	isReleased	Property	1 x 1	logical	-
3.3	TunablePropsChanged	Property	1 x 1	logical	-
3.4	inputDataType1	Property	1 x 6	char	-
3.5	inputSize1	Property	1 x 2	double	No
3.6	isInputComplex1	Property	1 x 1	logical	-
3.7	inputDirectFeedthrough1	Property	1 x 1	logical	-
3.8	inputDirectFeedthrough2	Property	1 x 1	logical	-
3.9	inputDirectFeedthrough3	Property	1 x 1	logical	-
3.10	inputDirectFeedthrough4	Property	1 x 1	logical	-
3.11	inputDirectFeedthrough5	Property	1 x 1	logical	-
3.12	inputDirectFeedthrough6	Property	1 x 1	logical	-
3.13	inputDirectFeedthrough7	Property	1 x 1	logical	-
3.14	inputDirectFeedthrough8	Property	1 x 1	logical	-
3.15	inputDirectFeedthrough9	Property	1 x 1	logical	-

- 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:

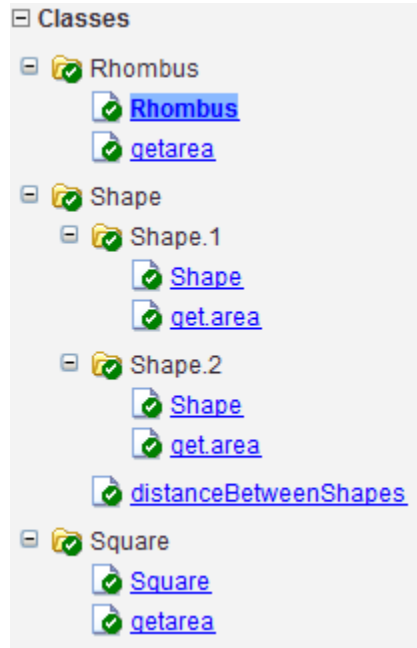
- Expand the class information to view the class methods.
- View a class method by clicking its name. The report displays the methods in the context of the full class definition.
- Filter the methods by size, complexity, and class by using the **Filter functions and methods** option.

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 “Create a Namespace with Packages”.

In the Variables Tab

The report displays all the objects in the selected function or class. By default, for classes that have properties, the list of properties is collapsed. Click the + symbol next to the object name to open the list.

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 subfunctions 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

Class *class* 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

persistent h
if isempty(h)
    h = MyClass;
end
```

```
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

persistent h
if isempty(h)
    h = MyClass;
end

b=h.mymethod();
b.aa=12;
```


Code Generation for Function Handles

- “How Working with Function Handles is Different for Code Generation” on page 10-2
- “Example: Defining and Passing Function Handles for Code Generation” on page 10-3
- “Limitations with Function Handles for Code Generation” on page 10-6

How Working with Function Handles is Different for Code Generation

You can use function handles to invoke functions indirectly and parameterize operations that you repeat frequently (see “Function Handles” in the MATLAB Programming Fundamentals documentation). 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 Chapter 2, “Functions Supported for Code Generation”)

Note You cannot define handles that reference extrinsic MATLAB functions (see “Calling MATLAB Functions” on page 12-11).

- Define function handles as scalar values
- Pass function handles as arguments to other functions (excluding extrinsic functions)

To generate efficient standalone code for enumerated data, you are restricted to using a subset of the operations you can perform with function handles in MATLAB, as described in “Limitations with Function Handles for Code Generation” on page 10-6

Example: Defining and Passing 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]));

% Define MATLAB function disp to be extrinsic
coder.extrinsic('disp');

disp(m);

% 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 “Generating MEX Functions from MATLAB Code at the Command Line” in the MATLAB Coder documentation.

Limitations with Function Handles for Code Generation

Function handles must be scalar values.

You cannot store function handles in matrices or structures.

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 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 12-11

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'));

coder.extrinsic('plot');

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, indicating 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.

You cannot view function handles from the debugger

You cannot display or watch function handles from the debugger. They appear as empty matrices.

Defining Functions for Code Generation

- “Specifying Variable Numbers of Arguments” on page 11-2
- “Supported Index Expressions” on page 11-3
- “Using varargin and varargout in for-Loops” on page 11-4
- “Implementing Wrapper Functions with varargin and varargout” on page 11-7
- “Passing Property/Value Pairs with varargin” on page 11-8
- “Rules for Using Variable Length Argument Lists for Code Generation” on page 11-10

Specifying Variable Numbers of Arguments

You can use `varargin` and `varargout` for passing and returning variable numbers of parameters to MATLAB functions called from a top-level function.

Common applications of `varargin` and `varargout` for code generation include:

- Using for-loops to apply operations to a variable number of arguments
- Implementing wrapper functions that accept any number of inputs and pass them to another function
- Passing variable numbers of property/value pairs as arguments to a function

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 not all (see “Rules for Using Variable Length Argument Lists for Code Generation” on page 11-10). The following sections explain how to code effectively using these constructs.

For more information about using `varargin` and `varargout` in MATLAB functions, see [Passing Variable Numbers of Arguments in the MATLAB Programming Fundamentals documentation](#).

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.

Using `varargin` and `varargout` in for-Loops

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

```



```
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;

```

For more information, see `coder.unroll` in the Code Generation from MATLAB reference documentation.

Example: 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 subfunction `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 “Rules for Using Variable Length Argument Lists for Code Generation” on page 11-10.

Implementing Wrapper Functions with varargin and varargout

You can use varargin and varargout to write wrapper functions that accept any number of inputs and pass them directly to another function.

Example: 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
...

```

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 “Rules for Using Variable Length Argument Lists for Code Generation” on page 11-10.

Passing Property/Value Pairs with varargin

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
...

```

Rules for Using Variable Length Argument Lists for Code Generation

Do not use varargin or varargout in top-level functions

You **cannot** use varargin or varargout as arguments to top-level functions. A *top-level function* is:

- The function called by Simulink in MATLAB Function block or by Stateflow in a MATLAB function.
- The function that you provide on the command line to codegen

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 and then call `inch_2_cm` as an external function or subfunction, 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 11-3.

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 “Using `varargin` and `varargout` in for-Loops” on page 11-4.

Do not write to `varargin`

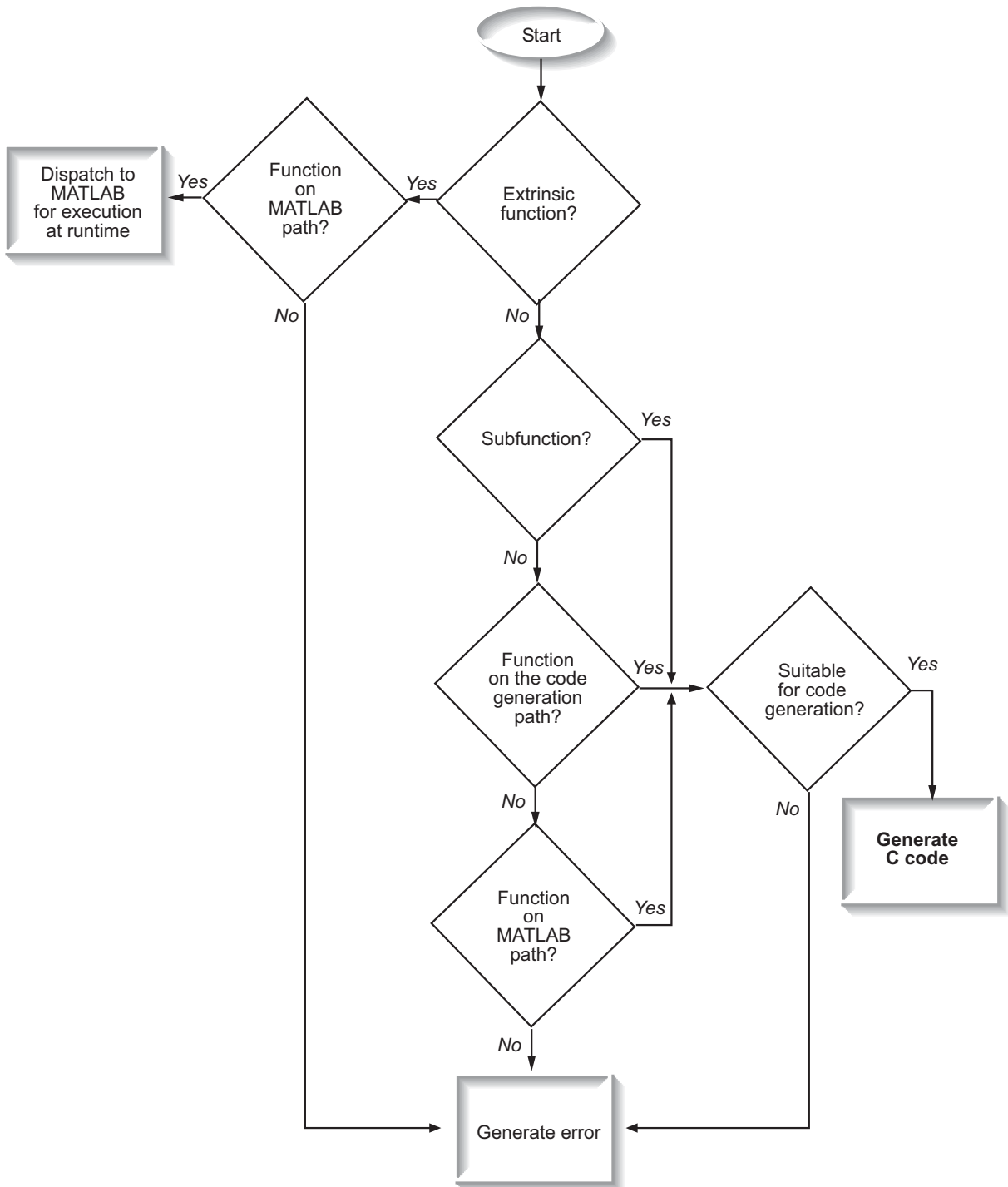
Generated code treats `varargin` as a read-only variable. If you want to write to any of the input arguments, copy the values into a local variable.

Calling Functions for Code Generation

- “How MATLAB Resolves Function Calls in Generated Code” on page 12-2
- “How MATLAB Resolves File Types on the Path for Code Generation” on page 12-6
- “Adding the Compilation Directive `%#codegen`” on page 12-8
- “Calling Subfunctions” on page 12-9
- “Calling Supported Toolbox Functions” on page 12-10
- “Calling MATLAB Functions” on page 12-11

How MATLAB Resolves Function Calls in Generated Code

From a MATLAB function, you can call subfunctions, 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 12-4.
- Attempts to compile all functions unless you explicitly declare them to be extrinsic

An extrinsic function is a function on the MATLAB path that the compiler dispatches to MATLAB software for execution because the target language does not support the function. MATLAB does not generate code for extrinsic functions. You declare functions to be extrinsic by using the construct `coder.extrinsic`, as described in “Declaring MATLAB Functions as Extrinsic Functions” on page 12-11.

- Resolves file type based on precedence rules described in “How MATLAB Resolves File Types on the Path for Code Generation” on page 12-6

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.

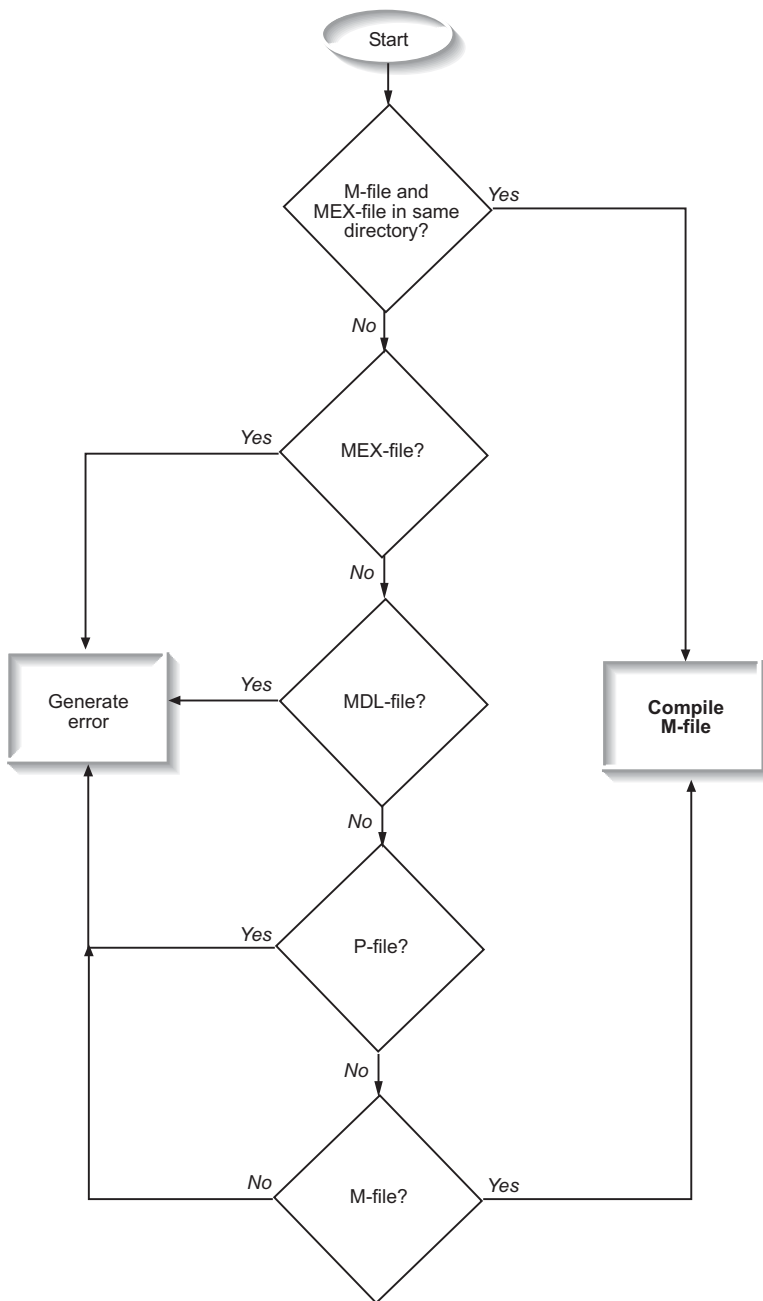
MATLAB applies the same dispatcher rules when searching each path (see “Function Precedence Order” in the MATLAB Programming Fundamentals documentation).

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 always shadows a file of the same name on the MATLAB path.

How MATLAB Resolves File Types on the Path for Code Generation

MATLAB uses the following precedence rules for code generation:



Adding the Compilation Directive `%#codegen`

Add the `%#codegen` directive (or pragma) to your function 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.

Calling Subfunctions

Subfunctions 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 subfunction mean:

```
function [mean, stdev] = stats(vals)
%#codegen

% Calculates a statistical mean and a standard
% deviation for the values in vals.

coder.extrinsic('plot');
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;
```

See “Subfunctions” in the MATLAB Programming Fundamentals documentation for more information.

Calling Supported Toolbox Functions

You can call toolbox functions directly if they are supported for code generation. For a list of supported functions, see “Functions Supported for Code Generation — Alphabetical List” on page 2-3.

Calling MATLAB Functions

MATLAB attempts to generate code for all functions unless you explicitly declare them to be extrinsic (see “How MATLAB Resolves Function Calls in Generated Code” on page 12-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 12-15).

There are two ways to declare a function to be extrinsic:

- Use the `coder.extrinsic` construct in main functions or subfunctions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 12-11).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using `feval`” on page 12-15).

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 subfunction:

```
coder.extrinsic('function_name_1', ... , 'function_name_n');
```

Example: Declaring Extrinsic Functions

The following code declares the MATLAB `patch` and `axis` functions extrinsic in the subfunction `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', 'axis');
```

```
x = [0;a;a];  
y = [0;0;b];  
patch(x, y, color);  
axis('equal');
```

By declaring these functions extrinsic, you instruct the compiler not to generate code for `patch` and `axis`, but instead dispatch them to MATLAB for execution.

To test the function, follow these steps:

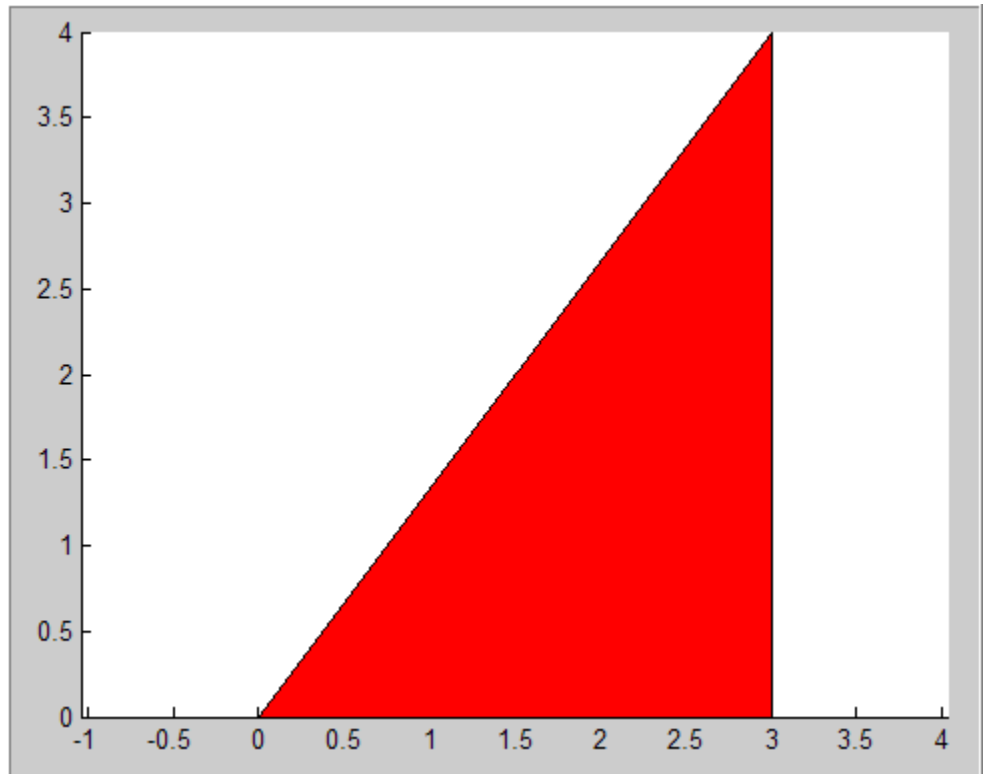
- 1 Convert `pythagoras` to a MEX function by executing this command at the MATLAB prompt:

```
codegen -args {1, 1, [.3 .3 .3]} pythagoras
```

- 2 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 produce no output — such as `plot` and `patch` — for visualizing results during simulation, without generating unnecessary code (see “How MATLAB Resolves Extrinsic Functions During Simulation” on page 12-15).
- 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 12-16).

- 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 12-14).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 12-14). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using `feval`” on page 12-15).

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 subfunction, 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 subfunction `mymin`.

- Call the MATLAB function using `feval`, as described in “Calling MATLAB Functions Using `feval`” on page 12-15.

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.

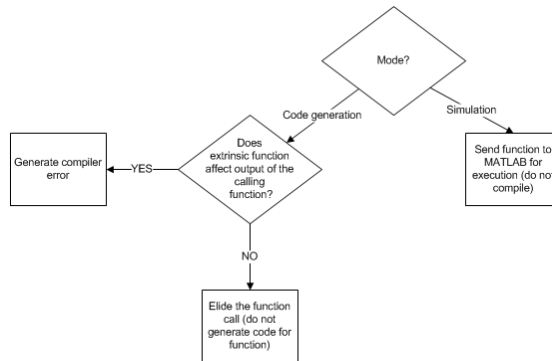
Consider the following example:

```
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 effect 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 12-16). Provided that there is no change to the output, 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 12-17.

Converting mxArray 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 write to the caller's workspace do not work during code generation. Such functions include:
 - `dbstack`
 - `evalin`
 - `assignin`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code may produce unpredictable results if your extrinsic function performs any of 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.

Generating Efficient and Reusable Code

- “Generating Efficient Code” on page 13-2
- “Generating Reusable Code” on page 13-4

Generating Efficient Code

In this section...

“Unrolling for-Loops” on page 13-2

“Inlining Functions” on page 13-2

“Eliminating Redundant Copies of Function Inputs” on page 13-2

Unrolling 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. By unrolling short loops with known bounds at compile time, MATLAB generates highly optimized code with no branches.

You can also force loop unrolling for individual functions by wrapping the loop header in an `coder.unroll` function. For more information, see `coder.unroll` in the Code Generation from MATLAB Function Reference.

Inlining Functions

MATLAB uses internal heuristics to determine whether or not to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. See `coder.inline` in the Code Generation from MATLAB Function Reference.

Eliminating Redundant Copies of Function Inputs

You can reduce the number of copies in your generated code by writing functions that use the same variable as both an input and an output. For example:

```
function A = foo( A, B ) %#codegen
A = A * B;
end
```

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, MATLAB passes the variable by

reference in the generated code instead of redundantly copying the input to a temporary variable. For example, input A above is passed by reference in the generated code because it also acts as an output for function `foo`:

```
...
/* Function Definitions */
void foo(real_T *A, real_T B)
{
    *A *= B;
}
...
```

The reference parameter optimization reduces memory usage and improves run-time performance, especially when the variable passed by reference is a large data structure. To achieve these benefits at the call site, call the function with the same variable as both input and output.

By contrast, suppose you rewrite function `foo` without using this optimization:

```
function y = foo2( A, B ) %#codegen
y = A * B;
end
```

In this case, MATLAB generates code that passes the inputs by value and returns the value of the output:

```
...
/* Function Definitions */
real_T foo2(real_T A, real_T B)
{
    return A * B;
}
...
```

Generating 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 “How MATLAB Resolves Function Calls in Generated Code” on page 12-2.

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

Examples

Use this list to find examples in the documentation.

Data Management

Example: Defining a Variable for Multiple Execution Paths on page 4-4

Example: Defining All Fields in a Structure on page 4-5

“Defining Uninitialized Variables” on page 4-9

Variable Reuse in an if Statement on page 4-12

Code Generation for Structures

“Adding Fields in Consistent Order on Each Control Flow Path” on page 6-4

“Using repmat to Define an Array of Structures with Consistent Field Properties” on page 6-7

“Defining an Array of Structures Using Concatenation” on page 6-8

“Making Structures Persistent” on page 6-9

Code Generation for Enumerated Data

“Defining and Using Enumerated Types for Code Generation” on page 7-13

“Using the `if` Statement on Enumerated Data Types” on page 7-18

“Using the `switch` Statement on Enumerated Data Types” on page 7-19

“Using the `while` Statement on Enumerated Data Types” on page 7-22

Generating Code for Variable-Size Data

“Generating Code for a MATLAB Function That Expands a Vector in a Loop” on page 8-8

Code Generation for Variable-Size Data

“Constraining the Value of a Variable That Specifies Dimensions of Variable-Size Data” on page 8-17

“Specifying the Upper Bounds for All Instances of a Local Variable” on page 8-18

“Example: Inferring Upper Bounds from Multiple Definitions with Different Shapes” on page 8-23

Code Generation for Function Handles

“Example: Defining and Passing Function Handles for Code Generation”
on page 10-3

Using Variable-Length Argument Lists

“Example: Using Variable Numbers of Arguments in a for-Loop” on page 11-5

“Example: Passing Variable Numbers of Arguments from One Function to Another” on page 11-7

Optimizing Generated Code

“Eliminating Redundant Copies of Function Inputs” on page 13-2

A

- arguments
 - limit on number for code generation from MATLAB 12-18

C

- C/C++ code generation for supported functions 2-1
- code generation from MATLAB
 - best practices for working with variables 4-3
 - calling MATLAB functions 12-11
 - calling MATLAB functions using feval 12-15
 - calling subfunctions 12-9
- characters 5-6
- communications system toolbox System objects 3-8
- compilation directive `%#codegen` 12-8
- computer vision system toolbox System objects 3-3
- converting mxArray to known types 12-17
- declaring MATLAB functions as extrinsic functions 12-11
- defining persistent variables 4-10
- defining variables 4-2
- defining variables by assignment 4-3
- dsp system toolbox System objects 3-14
- eliminating redundant copies of function inputs 13-2
- eliminating redundant copies of uninitialized variables 4-8
- function handles 10-1
- generating efficient code 13-2
- how it resolves function calls 12-2
- initializing persistent variables 4-10
- inlining functions 13-2
- limit on number of function arguments 12-18
- pragma 12-8
- resolving extrinsic function calls during simulation 12-15

- resolving extrinsic function calls in generated code 12-16
- rules for defining uninitialized variables 4-8
- setting properties of indexed variables 4-6
- supported toolbox functions 12-10
- unrolling for-loops 13-2
- using type cast operators in variable definitions 4-6
- variables, complex 5-4
- working with mxArray 12-16

- `coder.extrinsic` 12-11
- `coder.nullcopy`
 - uninitialized variables 4-8
- communications system toolbox System objects supported for code generation from MATLAB 3-8
- computer vision system toolbox System objects supported for code generation from MATLAB 3-3

D

- defining uninitialized variables
 - rules 4-8
- defining variables
 - for C/C++ code generation 4-3
- dsp system toolbox System objects supported for code generation from MATLAB 3-14

E

- eliminating redundant copies of function inputs 13-2
- extrinsic functions 12-11

F

- function handles
 - for code generation from MATLAB 10-1
- functions

- limit on number of arguments for code generation 12-18
- Functions supported for C/C++ code
 - generation 2-1
 - alphabetical list 2-3
 - arithmetic operator functions 2-66
 - bit-wise operation functions 2-67
 - casting functions 2-67
 - Communications System Toolbox functions 2-68
 - complex number functions 2-68
 - Computer Vision System Toolbox functions 2-69
 - data type functions 2-70
 - derivative and integral functions 2-70
 - discrete math functions 2-70
 - error handling functions 2-71
 - exponential functions 2-71
 - filtering and convolution functions 2-72
 - Fixed-Point Toolbox functions 2-72
 - histogram functions 2-81
 - Image Processing Toolbox functions 2-81
 - input and output functions 2-82
 - interpolation and computational geometry functions 2-82
 - linear algebra functions 2-83
 - logical operator functions 2-83
 - MATLAB Compiler functions 2-84
 - matrix/array functions 2-84
 - nonlinear numerical methods 2-88
 - polynomial functions 2-88
 - relational operator functions 2-88
 - rounding and remainder functions 2-89
 - set functions 2-89
 - signal processing functions 2-90
 - Signal Processing Toolbox functions 2-91
 - special value functions 2-95
 - specialized math functions 2-96
 - statistical functions 2-97

- string functions 2-97
- structure functions 2-98
- trigonometric functions 2-98
- Functions supported for MEX and C/C++ code
 - generation
 - categorized list 2-65

I

- indexed variables
 - setting properties for code generation from MATLAB 4-6
- initialization
 - persistent variables 4-10

M

- MATLAB
 - features not supported for code generation 1-11
- MATLAB for code generation
 - variable types 4-19
- MATLAB functions
 - and generating code for mxArray 12-16
- mxArrays
 - converting to known types 12-17
 - for code generation from MATLAB 12-16

P

- persistent variables
 - defining for code generation from MATLAB 4-10
 - initializing for code generation from MATLAB 4-10

S

- signal processing functions
 - for C/C++ code generation 2-91

T

type cast operators
using in variable definitions 4-6

U

uninitialized variables
eliminating redundant copies in generated
code 4-8

V

variable types supported for code generation
from MATLAB 4-19

variables

eliminating redundant copies in C/C++ code
generated from MATLAB 4-8

Variables

defining by assignment for code generation
from MATLAB 4-3

defining for code generation from
MATLAB 4-2