MATLAB® Coder™

User's Guide

MATLAB®

R2017a

MathWorks®

How to Contact MathWorks

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

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

MATLAB® Coder™ User's Guide

© COPYRIGHT 2011–2017 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	Online only	New for Version 2 (R2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.4 (Release 2013a)
September 2013	Online only	Revised for Version 2.5 (Release 2013b)
March 2014	Online only	Revised for Version 2.6 (Release 2014a)
October 2014	Online only	Revised for Version 2.7 (Release 2014b)
March 2015	Online only	Revised for Version 2.8 (Release 2015a)
September 2015	Online only	Revised for Version 3.0 (Release 2015b)
October 2015	Online only	Rereleased for Version 2.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.1 (Release 2016a)
September 2016	Online only	Revised for Version 3.2 (Release 2016b)
March 2017	Online only	Revised for Version 3.3 (Release 2017a)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

1 About MATLAB Coder

2 Design Considerations for C/C++ Code Generation

Functions, Classes, and System Objects Supported for Code Generation

3

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

5 Defining Data for Code Generation

6 Code Generation for Variable-Size Data

7 Code Generation for MATLAB Structures

8 Code Generation for Cell Arrays

9 Code Generation for Enumerated Data

10 Code Generation for MATLAB Classes

11 Code Generation for Function Handles

12 Defining Functions for Code Generation

13 Calling Functions for Code Generation

14 Fixed-Point Conversion

Automated Fixed-Point Conversion Using Programmatic Workflow

15

16 Single-Precision Conversion

17 Setting Up a MATLAB Coder Project

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

19 Testing MEX Functions in MATLAB

20 Generating C/C++ Code from MATLAB Code

21 Verify Generated C/C++ Code

22 Code Replacement for MATLAB Code

23 Custom Toolchain Registration

24 Deploying Generated Code

25 Accelerating MATLAB Algorithms

26 Calling C/C++ Functions from Generated Code

27 External Code Integration

28 Generate Efficient and Reusable Code

29 Generating Reentrant C Code from MATLAB Code

30 Troubleshooting Code Generation Problems

About MATLAB Coder

MATLAB Coder Product Description

Generate C and C++ code from MATLAB code

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

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

Key Features

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

Product Overview

When to Use MATLAB Coder

Use MATLAB Coder to:

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

Code Generation for Embedded Software Applications

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

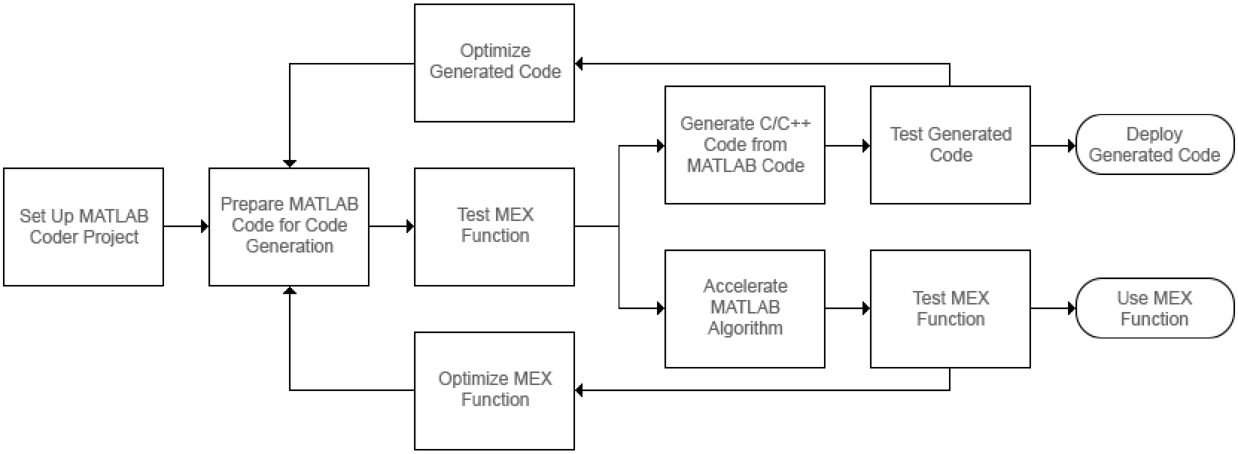
- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize the generated code for a specific target environment.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

Code Generation for Fixed-Point Algorithms

Using the Fixed-Point Designer™ product, you can generate:

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

Code Generation Workflow



See Also

- “Set Up a MATLAB Coder Project” on page 17-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 18-2
- “Workflow for Testing MEX Functions in MATLAB” on page 19-3
- “Code Generation Workflow” on page 20-3
- “Workflow for Accelerating MATLAB Algorithms” on page 25-2

Design Considerations for C/C++ Code Generation

When to Generate Code from MATLAB Algorithms

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

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

When Not to Generate Code from MATLAB Algorithms

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

To:	Use:
Deploy an application that uses handle graphics	MATLAB Compiler™
Use Java®	MATLAB Compiler SDK™
Use toolbox functions that do not support code generation	Toolbox functions that you rewrite for desktop and embedded applications
Deploy MATLAB based GUI applications on a supported MATLAB host	MATLAB Compiler
Deploy web-based or Windows® applications	MATLAB Compiler SDK

To:	Use:
Interface C code with MATLAB	MATLAB `mex` function

Which Code Generation Feature to Use

To...	Use...	Required Product	To Explore Further...
Generate MEX functions for verifying generated code	`codegen` function	MATLAB Coder	Try this in “MEX Function Generation at the Command Line”.
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder app	MATLAB Coder	Try this in “C Code Generation Using the MATLAB Coder App”.
	`codegen` function	MATLAB Coder	Try this in “C Code Generation at the Command Line”.
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder app	MATLAB Coder	See “Accelerate MATLAB Algorithms” on page 25-13.
	`codegen` function	MATLAB Coder	
Integrate MATLAB code into Simulink®	MATLAB Function block	Simulink	Try this in “Track Object Using MATLAB Code” (Simulink).
Speed up fixed-point MATLAB code	`fiaccel` function	Fixed-Point Designer	Learn more in “Code Acceleration and Code Generation from MATLAB” (Fixed-Point Designer).
Integrate custom C code into MATLAB and generate efficient, readable code	`codegen` function	MATLAB Coder	Learn more in “Specify External File Locations” on page 24-14.
Integrate custom C code into code generated from MATLAB	`coder.ceval` function	MATLAB Coder	Learn more in `coder.ceval`.
Generate HDL from MATLAB code	MATLAB Function block	Simulink and HDL Coder™	Learn more at `www.mathworks.com/products/slhdlcoder`.

Prerequisites for C/C++ Code Generation from MATLAB

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

- MATLAB Coder product
- C/C++ compiler

MATLAB Code Design Considerations for Code Generation

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

- Data types

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

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

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

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

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

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

  To improve the speed of the generated code:

  - Choose a suitable C/C++ compiler. Do not use the default compiler that MathWorks supplies with MATLAB for Windows 64-bit platforms.
  - Consider disabling run-time checks.

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

See Also

- “Data Definition Basics”
- “Code Generation for Variable-Size Arrays” on page 6-2
- “Control Run-Time Checks” on page 25-15

Differences Between Generated Code and MATLAB Code

To convert MATLAB code to efficient C/C++ code, the code generator introduces optimizations that intentionally cause the generated code to behave differently, and sometimes produce different results, than the original source code.

Here are some of the differences:

- “Character Size” on page 2-8
- “Order of Evaluation in Expressions” on page 2-8
- “Termination Behavior” on page 2-10
- “Size of Variable-Size N-D Arrays” on page 2-10
- “Size of Empty Arrays” on page 2-10
- “Size of Empty Array That Results from Deleting Elements of an Array” on page 2-10
- “Floating-Point Numerical Results” on page 2-11
- “NaN and Infinity Patterns” on page 2-12
- “Code Generation Target” on page 2-12
- “MATLAB Class Initial Values” on page 2-12
- “Variable-Size Data” on page 2-12
- “Complex Numbers” on page 2-12

When you run your program, run-time error checks can detect some of these differences. By default, run-time error checks are enabled for MEX code and disabled for standalone C/C++ code. To help you identify and address differences before you deploy code, the code generator reports a subset of the differences as potential 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 Character Arrays” on page 5-9.

Order of Evaluation in Expressions

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

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

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

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

- Rewrite

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

  as

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

  so that the generated code calls `f1` before `f2`.
- Assign the outputs of a multi-output function call to variables that do not depend on one another. For example, rewrite

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

  as

  ```
  [y, a, b] = foo;
  y.f = a;
  y.g = b;
  ```
- When you access the contents of multiple cells of a cell array, assign the results to variables that do not depend on one another. For example, rewrite

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

  as

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

```
  y.g = b;
```

Termination Behavior

Generated code does not match the termination behavior of MATLAB source code. For example, if infinite loops do not have side effects, optimizations remove them from generated code. As a result, the generated code can possibly 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 6-26.

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 6-27.

Size of Empty Array That Results from Deleting Elements of an Array

Deleting all elements of an array results in an empty array. The size of this empty array in generated code might differ from its size in MATLAB source code.

Case	Example Code	Size of Empty Array in MATLAB	Size of Empty Array in Generated Code

Delete all elements of an m-by-n array by using the `colon` operator (`:`).	`coder.varsize('X',[4,4],[1,1]);`
	`X = zeros(2);`
	`X(:) = [];`
		`0-by-0`	`1-by-0`

Delete all elements of a row vector by	`coder.varsize('X',[1,4],[0,1]);`
	`X = zeros(1,4);`
	`X(:) = [];`
		`0-by-0`	`1-by-0`

Case	Example Code	Size of Empty Array in MATLAB	Size of Empty Array in Generated Code
using the `colon` operator (`:`).			

Delete all elements of a column vector by using the `colon` operator (`:`).	`coder.varsize('X',[4,1],[1,0]);`
	`X = zeros(4,1);`
	`X(:) = [];`
		`0-by-0`	`0-by-1`

Delete all elements of a column vector by deleting one element at a time.	`coder.varsize('X',[4,1],[1,0]);`
	`X = zeros(4,1);`
	`for i = 1:4`
	`    X(1)= [];`
	`end`
		`1-by-0`	`0-by-1`

Floating-Point Numerical Results

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

When computer hardware uses extended precision registers

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

For certain advanced library functions

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

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

For implementation of BLAS library functions

For implementations of BLAS library functions, generated C/C++ code uses reference implementations of BLAS functions. These reference implementations might produce different results from platform-specific BLAS implementations in MATLAB.

NaN and Infinity Patterns

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

Code Generation Target

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

MATLAB Class Initial Values

Before code generation, at class loading time, MATLAB computes class initial values. The code generator uses the value that MATLAB computes. It does not recompute the initial value. If the initialization uses a function call to compute the initial value, the code generator does not execute this function. If the function modifies a global state, for example, a persistent variable, code generator might provide a different initial value than MATLAB. For more information, see "Defining Class Properties for Code Generation" on page 10-4.

Variable-Size Data

See "Incompatibilities with MATLAB in Variable-Size Support for Code Generation" on page 6-24.

Complex Numbers

See "Code Generation for Complex Data" on page 5-4.

More About

- "Potential Differences Reporting" on page 2-13
- "Potential Differences Messages" on page 2-15

Potential Differences Reporting

Generation of efficient C/C++ code from MATLAB code sometimes results in behavior differences between the generated code and the MATLAB code. When you run your program, run-time error checks can detect some of these differences. By default, run-time error checks are enabled for MEX code and disabled for standalone C/C++ code. To help you identify and address differences before you deploy code, the code generator reports a subset of the differences as potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

Addressing Potential Differences Messages

If the code generator detects a potential difference, it displays a message for the difference on the **Potential Differences** tab of the report or the MATLAB Coder app. To highlight the MATLAB code that corresponds to the message, click the message.

The presence of a potential difference message does not necessarily mean that the difference will occur when you run the generated code. To determine whether the potential difference affects your application:

- Analyze the behavior of your MATLAB code for the range of data for your application.
- Test a MEX function generated from your MATLAB code. Use the range of data that your application uses. If the difference occurs, the MEX function reports an error.

If your analysis or testing confirms the reported difference, consider modifying your code. Some potential differences messages provide a workaround. For additional information about some of the potential differences messages, see “Potential Differences Messages” on page 2-15. Even if you modify your code to prevent a difference from occurring at run time, the code generator might still report the potential difference.

The set of potential differences that the code generator detects is a subset of the differences that MEX functions report as errors. It is a best practice to test a MEX function over the full range of application data.

Disabling and Enabling Potential Differences Reporting

By default, potential differences reporting is enabled for:

- Code generation with the `codegen` command

- The **Check for Run-Time Issues** step in the MATLAB Coder app

To disable potential differences reporting:

- In a code configuration object, set `ReportPotentialDifferences` to `false`.
- In the MATLAB Coder app, in the **Debugging** settings, clear the **Report differences from MATLAB** check box.

By default, potential differences reporting is disabled for the **Generate code** step and the code generation report in the MATLAB Coder app. To enable potential differences reporting, in the **Debugging** settings, select the **Report differences from MATLAB** check box.

More About

- “Potential Differences Messages” on page 2-15
- “Differences Between Generated Code and MATLAB Code” on page 2-8
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 6-24
- “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 21-28

Potential Differences Messages

When you enable potential differences reporting, the code generator reports potential differences between the behavior of the generated code and the behavior of the MATLAB code. Reviewing and addressing potential differences before you generate standalone code helps you to avoid errors and incorrect answers in generated code.

Here are some of the potential differences messages:

- "Automatic Dimension Incompatibility" on page 2-15
- "mtimes No Dynamic Scalar Expansion" on page 2-16
- "Matrix-Matrix Indexing" on page 2-16
- "Vector-Vector Indexing" on page 2-17
- "Size Mismatch" on page 2-17

Automatic Dimension Incompatibility

```
In the generated code, the dimension to operate along is selected automatically,
and might be different from MATLAB. Consider specifying the working dimension
explicitly as a constant value.
```

This restriction applies to functions that take the working dimension (the dimension along which to operate) as input. In MATLAB and in code generation, if you do not supply the working dimension, the function selects it. In MATLAB, the function selects the first dimension whose size does not equal 1. For code generation, the function selects the first dimension that has a variable size or that has a fixed size that does not equal 1. If the working dimension has a variable size and it becomes 1 at run time, then the working dimension is different from the working dimension in MATLAB. Therefore, when run-time error checks are enabled, an error can occur.

For example, suppose that `X` is a variable-size matrix with dimensions `1x:3x:5`. In the generated code, `sum(X)` behaves like `sum(X,2)`. In MATLAB, `sum(X)` behaves like `sum(X,2)` unless `size(X,2)` is 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`.

To avoid this issue, specify the intended working dimension explicitly as a constant value. For example, `sum(X,2)`.

mtimes No Dynamic Scalar Expansion

The generated code performs a general matrix multiplication. If a variable-size matrix operand becomes a scalar at run time, dimensions must still agree. There will not be an automatic switch to scalar multiplication.

Consider the multiplication `A*B`. If the code generator is aware that `A` is scalar and `B` is a matrix, the code generator produces code for scalar-matrix multiplication. However, if the code generator is aware that `A` and `B` are variable-size matrices, it produces code for a general matrix multiplication. At run time, if `A` turns out to be scalar, the generated code does not change its behavior. Therefore, when run-time error checks are enabled, a size mismatch error can occur.

Matrix-Matrix Indexing

For indexing a matrix with a matrix, matrix1(matrix2), the code generator assumed that the result would have the same size as matrix2. If matrix1 and matrix2 are vectors at run time, their orientations must match.

In matrix-matrix indexing, you use one matrix to index into another matrix. In MATLAB, the general rule for matrix-matrix indexing is that the size and orientation of the result match the size and orientation of the index matrix. For example, if `A` and `B` are matrices, `size(A(B))` equals `size(B)`. When `A` and `B` are vectors, MATLAB applies a special rule. The special vector-vector indexing rule is that the orientation of the result is the orientation of the data matrix. For example, i`A` is 1-by-5 and `B` is 3-by-1, then `A(B)` is 1-by-3.

The code generator applies the same matrix-matrix indexing rules as MATLAB. If `A` and `B` are variable-size matrices, to apply the matrix-matrix indexing rules, the code generator assumes that the `size(A(B))` equals `size(B)`. If, at run time, `A` and `B` become vectors and have different orientations, then the assumption is incorrect. Therefore, when run-time error checks are enabled, an error can occur.

To avoid this issue, force your data to be a vector by using the colon operator for indexing. For example, suppose that 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. Therefore, the code generator applies the indexing rule for indexing one vector with another vector. The orientation of the result is the orientation of the data vector, C.

Vector-Vector Indexing

```
For indexing a vector with a vector, vector1(vector2), the code generator
assumed that the result would have the same orientation as vector1. If vector1
is a scalar at run time, the orientation of vector2 must match vector1.
```

In MATLAB, the special rule for vector-vector indexing is that the orientation of the result is the orientation of the data vector. For example, if A is 1-by-5 and B is 3-by-1, then A(B) is 1-by-3. If, however, the data vector A is a scalar, then the orientation of A(B) is the orientation of the index vector B.

The code generator applies the same vector-vector indexing rules as MATLAB. If A and B are variable-size vectors, to apply the indexing rules, the code generator assumes that the orientation of B matches the orientation of A. At run time, if A is scalar and the orientation of A and B do not match, then the assumption is incorrect. Therefore, when run-time error checks are enabled, a run-time error can occur.

To avoid this issue, make the orientations of the vectors match. Alternatively, index single elements by specifying the row and column. For example, A(row, column).

Size Mismatch

```
The generated code assumes that the sizes on the left and right sides match.
```

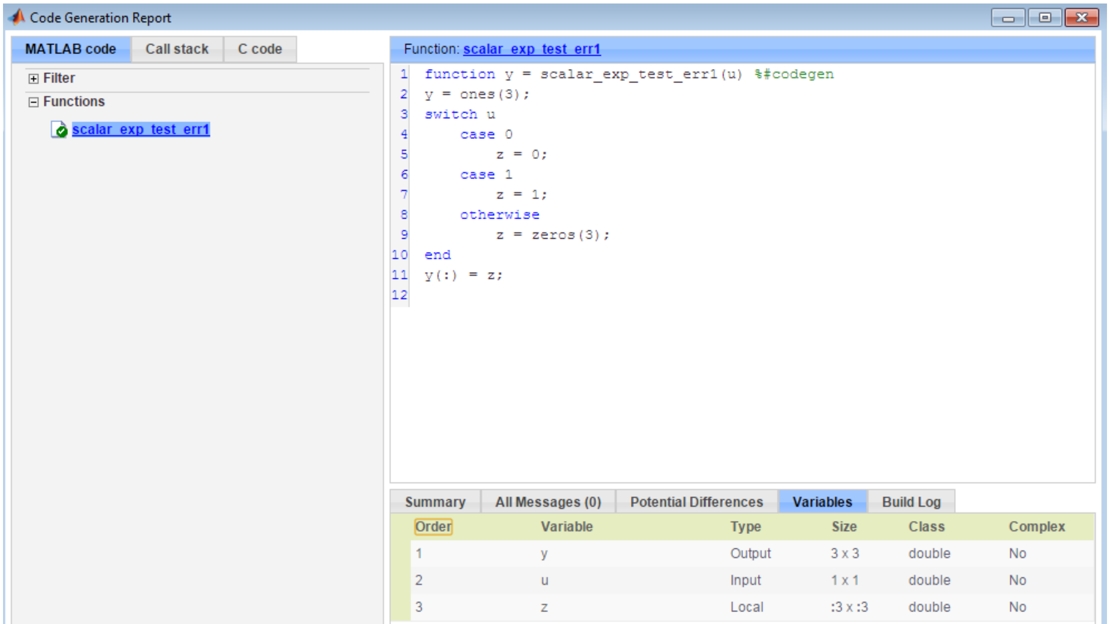
Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. If one operand is a scalar and the other is not, scalar expansion applies the scalar to every element of the other operand.

During code generation, scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Therefore, when run-time error checks are enabled, a run-time error can occur.

Consider this function:

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

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



Function: scalar_exp_test_err1

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

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

If you run the MEX function with `u` equal to 0 or 1, the generated code does not perform scalar expansion, even though `z` is scalar at run time. Therefore, when run-time error checks are enabled, a run-time error can occur.

```
scalar_exp_test_err1_mex(0)
```

```
Sizes mismatch: 9 ~= 1.

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

To avoid this issue, use indexing to force `z` to be a scalar value.

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

More About

- "Potential Differences Reporting" on page 2-13
- "Differences Between Generated Code and MATLAB Code" on page 2-8
- "Incompatibilities with MATLAB in Variable-Size Support for Code Generation" on page 6-24
- "Run-Time Error Detection and Reporting in Standalone C/C++ Code" on page 21-28

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

MATLAB Features That Code Generation Supports

Code generation from MATLAB code supports the following language features:

- n-dimensional arrays (see “Array Size Restrictions for Code Generation” on page 5-10)
- matrix operations, including deletion of rows and columns
- variable-sized data (see “Code Generation for Variable-Size Arrays” on page 6-2)
- subscripting (see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 6-30)
- complex numbers (see “Code Generation for Complex Data” on page 5-4)
- numeric classes (see “Supported Variable Types” on page 4-16)
- double-precision, single-precision, and integer math
- fixed-point arithmetic
- program control statements `if`, `switch`, `for`, `while`, and `break`
- arithmetic, relational, and logical operators
- local functions
- persistent variables
- global variables (see “Specify Global Variable Type and Initial Value Using the App” on page 17-30)
- structures (see “Structure Definition for Code Generation” on page 7-2)
- cell arrays (see “Cell Arrays”)
- characters (see “Code Generation for Character Arrays” on page 5-9)
- function handles (see “Function Handle Limitations for Code Generation” on page 11-2)
- anonymous functions (see “Code Generation for Anonymous Functions” on page 12-6)
- recursive functions (see “Code Generation for Recursive Functions” on page 13-18)
- nested functions (see “Code Generation for Nested Functions” on page 12-7)

- variable length input and output argument lists
- subset of MATLAB toolbox functions (see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 3-2)
- subset of functions and System objects in several toolboxes (see “Functions and Objects Supported for C/C++ Code Generation — Category List” on page 3-67)
- MATLAB classes (see “MATLAB Classes Definition for Code Generation” on page 10-2)
- function calls (see “Resolution of Function Calls for Code Generation” on page 13-2)

MATLAB Language Features That Code Generation Does Not Support

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

- implicit expansion

  Code generation does not support implicit expansion of arrays with compatible sizes during execution of element-wise operations or functions. If your MATLAB code relies on implicit expansion, code generation results in a size-mismatch error. For fixed-size arrays, the error occurs at compile time. For variable-size arrays, the error occurs at run time. For more information about implicit expansion, see “Compatible Array Sizes for Basic Operations” (MATLAB).
- string arrays
- categorical arrays
- date and time arrays
- Java
- Map containers
- sparse matrices
- tables
- time series objects
- `try/catch` statements

This list is not exhaustive. To see if a feature is supported for code generation, see “MATLAB Features That Code Generation Supports” on page 2-20.

3

Functions, Classes, and System Objects Supported for Code Generation

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

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

To find supported functions, classes, and System objects by MATLAB category or toolbox, see "Functions and Objects Supported for C/C++ Code Generation — Category List" on page 3-67.

Note: For more information on code generation for fixed-point algorithms, refer to "Code Acceleration and Code Generation from MATLAB" (Fixed-Point Designer).

In the following table, an asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Name	Product
abs	MATLAB
abs	Fixed-Point Designer
accumneg	Fixed-Point Designer
accumpos	Fixed-Point Designer
acos*	MATLAB
acosd	MATLAB
acosh*	MATLAB
acot	MATLAB
acotd	MATLAB
acoth	MATLAB
acsc	MATLAB
acscd	MATLAB
acsch	MATLAB
adaptthresh*	Image Processing Toolbox

Name	Product
add*	Fixed-Point Designer
affine2d*	Image Processing Toolbox
aictest*	Phased Array System Toolbox™
airy*	MATLAB
albersheim*	Phased Array System Toolbox
alignsignals	Signal Processing Toolbox
all*	MATLAB
all	Fixed-Point Designer
ambgfun*	Phased Array System Toolbox
and	MATLAB
angdiff	Robotics System Toolbox™
angle	MATLAB
any*	MATLAB
any	Fixed-Point Designer
aperture2gain*	Phased Array System Toolbox
appcoef*	Wavelet Toolbox™
appcoef2*	Wavelet Toolbox
asec	MATLAB
asecd	MATLAB
asech	MATLAB
asin*	MATLAB
asind	MATLAB
asinh	MATLAB
assert*	MATLAB
assignDetectionsToTracks	Computer Vision System Toolbox
atan	MATLAB
atan2	MATLAB

Name	Product
atan2	Fixed-Point Designer
atan2d	MATLAB
atand	MATLAB
atanh*	MATLAB
audioDeviceReader*	Audio System Toolbox™
audioDeviceWriter*	Audio System Toolbox
audioDeviceWriter*	DSP System Toolbox
audioOscillator*	Audio System Toolbox
audioPluginInterface	Audio System Toolbox
audioPluginParameter	Audio System Toolbox
audioPlugin	Audio System Toolbox
audioPluginSource	Audio System Toolbox
axang2quat	Robotics System Toolbox
axang2rotm	Robotics System Toolbox
axang2tform	Robotics System Toolbox
az2broadside*	Phased Array System Toolbox
azel2phitheta*	Phased Array System Toolbox
azel2phithetapat*	Phased Array System Toolbox
azel2uv*	Phased Array System Toolbox
azel2uvpat*	Phased Array System Toolbox
azelaxes*	Phased Array System Toolbox
bandwidth	MATLAB
barthannwin*	Signal Processing Toolbox
bartlett*	Signal Processing Toolbox
bboxOverlapRatio*	Computer Vision System Toolbox
bbox2points	Computer Vision System Toolbox
bchgenpoly*	Communications System Toolbox
beat2range*	Phased Array System Toolbox

Name	Product
besselap*	Signal Processing Toolbox
besseli*	MATLAB
besselj*	MATLAB
beta	MATLAB
betacdf	Statistics and Machine Learning Toolbox™
betafit	Statistics and Machine Learning Toolbox
betainc*	MATLAB
betaincinv*	MATLAB
betainv	Statistics and Machine Learning Toolbox
betalike	Statistics and Machine Learning Toolbox
betaln	MATLAB
betapdf	Statistics and Machine Learning Toolbox
betarnd*	Statistics and Machine Learning Toolbox
betastat	Statistics and Machine Learning Toolbox
bi2de	Communications System Toolbox
billingsleyicm*	Phased Array System Toolbox
bin2dec*	MATLAB
bin2gray	Communications System Toolbox
binocdf	Statistics and Machine Learning Toolbox
binoinv	Statistics and Machine Learning Toolbox

Name	Product
binopdf	Statistics and Machine Learning Toolbox
binornd*	Statistics and Machine Learning Toolbox
binostat	Statistics and Machine Learning Toolbox
bitand	MATLAB
bitand*	Fixed-Point Designer
bitandreduce	Fixed-Point Designer
bitcmp	MATLAB
bitcmp	Fixed-Point Designer
bitconcat	Fixed-Point Designer
bitget	MATLAB
bitget	Fixed-Point Designer
bitor	MATLAB
bitor*	Fixed-Point Designer
bitorreduce	Fixed-Point Designer
bitreplicate	Fixed-Point Designer
bitrevorder	Signal Processing Toolbox
bitrol	Fixed-Point Designer
bitror	Fixed-Point Designer
bitset	MATLAB
bitset	Fixed-Point Designer
bitshift	MATLAB
bitshift	Fixed-Point Designer
bitsliceget	Fixed-Point Designer
bitsll*	Fixed-Point Designer
bitsra*	Fixed-Point Designer

Name	Product
bitsrl*	Fixed-Point Designer
bitxor*	MATLAB
bitxor	Fixed-Point Designer
bitxorreduce	Fixed-Point Designer
blackman*	Signal Processing Toolbox
blackmanharris*	Signal Processing Toolbox
blanks	MATLAB
blkdiag	MATLAB
bohmanwin*	Signal Processing Toolbox
boundarymask*	Image Processing Toolbox
break	MATLAB
BRISKPoints*	Computer Vision System Toolbox
broadside2az*	Phased Array System Toolbox
bsxfun*	MATLAB
buttap*	Signal Processing Toolbox
butter*	Signal Processing Toolbox
buttord*	Signal Processing Toolbox
bw2range*	Phased Array System Toolbox
bwareaopen*	Image Processing Toolbox
bwboundaries*	Image Processing Toolbox
bwconncomp*	Image Processing Toolbox
bwdist*	Image Processing Toolbox
bweuler*	Image Processing Toolbox
bwlabel*	Image Processing Toolbox
bwlookup*	Image Processing Toolbox
bwmorph*	Image Processing Toolbox
bwpack*	Image Processing Toolbox

Name	Product
bwperim*	Image Processing Toolbox
bwselect*	Image Processing Toolbox
bwtraceboundary*	Image Processing Toolbox
bwunpack*	Image Processing Toolbox
cameas	Automated Driving System Toolbox™
cameasjac	Automated Driving System Toolbox
cameraMatrix*	Computer Vision System Toolbox
cameraParameters*	Computer Vision System Toolbox
cameraPose*	Computer Vision System Toolbox
cameraPoseToExtrinsics	Computer Vision System Toolbox
cart2hom	Robotics System Toolbox
cart2pol	MATLAB
cart2sph	MATLAB
cart2sphvec*	Phased Array System Toolbox
cast	MATLAB
cat*	MATLAB
cbfweights*	Phased Array System Toolbox
cconv	Signal Processing Toolbox
cdf	Statistics and Machine Learning Toolbox
ceil	MATLAB
ceil	Fixed-Point Designer
cell*	MATLAB
cfirpm*	Signal Processing Toolbox
char*	MATLAB
cheb1ap*	Signal Processing Toolbox
cheb1ord*	Signal Processing Toolbox

Name	Product
cheb2ap*	Signal Processing Toolbox
cheb2ord*	Signal Processing Toolbox
chebwin*	Signal Processing Toolbox
cheby1*	Signal Processing Toolbox
cheby2*	Signal Processing Toolbox
chi2cdf	Statistics and Machine Learning Toolbox
chi2inv	Statistics and Machine Learning Toolbox
chi2pdf	Statistics and Machine Learning Toolbox
chi2rnd*	Statistics and Machine Learning Toolbox
chi2stat	Statistics and Machine Learning Toolbox
chol	MATLAB
cholupdate	MATLAB
circpol2pol*	Phased Array System Toolbox
circshift	MATLAB
cl2tf*	DSP System Toolbox
class	MATLAB
ClassificationECOC* or CompactClassificationECOC*	Statistics and Machine Learning Toolbox
ClassificationEnsemble*, ClassificationBaggedEnsemble*, or CompactClassificationEnsemble*	Statistics and Machine Learning Toolbox
ClassificationLinear*	Statistics and Machine Learning Toolbox
ClassificationSVM* or CompactClassificationSVM*	Statistics and Machine Learning Toolbox

Name	Product
ClassificationTree* or CompactClassificationTree*	Statistics and Machine Learning Toolbox
colon*	MATLAB
comm.ACPR*	Communications System Toolbox
comm.AGC*	Communications System Toolbox
comm.AlgebraicDeinterleaver*	Communications System Toolbox
comm.AlgebraicInterleaver*	Communications System Toolbox
comm.APPDecoder*	Communications System Toolbox
comm.AWGNChannel*	Communications System Toolbox
comm.BarkerCode*	Communications System Toolbox
comm.BasebandFileReader*	Communications System Toolbox
comm.BasebandFileWriter*	Communications System Toolbox
comm.BCHDecoder*	Communications System Toolbox
comm.BCHEncoder*	Communications System Toolbox
comm.BinarySymmetricChannel*	Communications System Toolbox
comm.BlockDeinterleaver*	Communications System Toolbox
comm.BlockInterleaver*	Communications System Toolbox
comm.BPSKDemodulator*	Communications System Toolbox
comm.BPSKModulator*	Communications System Toolbox
comm.CarrierSynchronizer*	Communications System Toolbox
comm.CCDF*	Communications System Toolbox
comm.CoarseFrequencyCompensator*	Communications System Toolbox
comm.ConstellationDiagram*	Communications System Toolbox
comm.ConvolutionalDeinterleaver*	Communications System Toolbox
comm.ConvolutionalEncoder*	Communications System Toolbox
comm.ConvolutionalInterleaver*	Communications System Toolbox
comm.CPFSKDemodulator*	Communications System Toolbox
comm.CPFSKModulator*	Communications System Toolbox

Name	Product
comm.CPMCarrierPhaseSynchronizer*	Communications System Toolbox
comm.CPMDemodulator*	Communications System Toolbox
comm.CPMModulator*	Communications System Toolbox
comm.CRCDetector*	Communications System Toolbox
comm.CRCGenerator*	Communications System Toolbox
comm.DBPSKDemodulator*	Communications System Toolbox
comm.DBPSKModulator*	Communications System Toolbox
comm.Descrambler*	Communications System Toolbox
comm.DifferentialDecoder*	Communications System Toolbox
comm.DifferentialEncoder*	Communications System Toolbox
comm.DiscreteTimeVCO*	Communications System Toolbox
comm.DPSKDemodulator*	Communications System Toolbox
comm.DPSKModulator*	Communications System Toolbox
comm.DQPSKDemodulator*	Communications System Toolbox
comm.DQPSKModulator*	Communications System Toolbox
comm.ErrorRate*	Communications System Toolbox
comm.EVM*	Communications System Toolbox
comm.EyeDiagram*	Communications System Toolbox
comm.FMBroadcastDemodulator*	Communications System Toolbox
comm.FMBroadcastModulator*	Communications System Toolbox
comm.FMDemodulator*	Communications System Toolbox
comm.FMModulator*	Communications System Toolbox
comm.FSKDemodulator*	Communications System Toolbox
comm.FSKModulator*	Communications System Toolbox
comm.GeneralQAMDemodulator*	Communications System Toolbox
comm.GeneralQAMModulator*	Communications System Toolbox
comm.GeneralQAMTCMDemodulator*	Communications System Toolbox
comm.GeneralQAMTCMModulator*	Communications System Toolbox

Name	Product
comm.GMSKDemodulator*	Communications System Toolbox
comm.GMSKModulator*	Communications System Toolbox
comm.GMSKTimingSynchronizer*	Communications System Toolbox
comm.GoldSequence*	Communications System Toolbox
comm.HadamardCode*	Communications System Toolbox
comm.HDLCRCDetector*	Communications System Toolbox
comm.HDLCRCGenerator*	Communications System Toolbox
comm.HDLRSDecoder*	Communications System Toolbox
comm.HDLRSEncoder*	Communications System Toolbox
comm.HelicalDeinterleaver*	Communications System Toolbox
comm.HelicalInterleaver*	Communications System Toolbox
comm.IntegrateAndDumpFilter*	Communications System Toolbox
comm.IQImbalanceCompensator*	Communications System Toolbox
comm.KasamiSequence*	Communications System Toolbox
comm.LDPCDecoder*	Communications System Toolbox
comm.LDPCEncoder*	Communications System Toolbox
comm.LTEMIMOChannel*	Communications System Toolbox
comm.MatrixDeinterleaver*	Communications System Toolbox
comm.MatrixHelicalScanDeinterleaver*	Communications System Toolbox
comm.MatrixHelicalScanInterLeaver*	Communications System Toolbox
comm.MatrixInterleaver*	Communications System Toolbox
comm.MemorylessNonlinearity*	Communications System Toolbox
comm.MER*	Communications System Toolbox
comm.MIMOChannel*	Communications System Toolbox
comm.MLSEEqualizer*	Communications System Toolbox
comm.MSKDemodulator*	Communications System Toolbox
comm.MSKModulator*	Communications System Toolbox
comm.MSKTimingSynchronizer*	Communications System Toolbox

Name	Product
comm.MultiplexedDeinterleaver*	Communications System Toolbox
comm.MultiplexedInterleaver*	Communications System Toolbox
comm.OFDMDemodulator*	Communications System Toolbox
comm.OFDMModulator*	Communications System Toolbox
comm.OSTBCCombiner*	Communications System Toolbox
comm.OSTBCEncoder*	Communications System Toolbox
comm.OQPSKDemodulator*	Communications System Toolbox
comm.OQPSKModulator*	Communications System Toolbox
comm.PAMDemodulator*	Communications System Toolbox
comm.PAMModulator*	Communications System Toolbox
comm.PhaseFrequencyOffset*	Communications System Toolbox
comm.PhaseNoise*	Communications System Toolbox
comm.PNSequence*	Communications System Toolbox
comm.PreambleDetector*	Communications System Toolbox
comm.PSKCoarseFrequencyEstimator*	Communications System Toolbox
comm.PSKDemodulator*	Communications System Toolbox
comm.PSKModulator*	Communications System Toolbox
comm.PSKTCMDemodulator*	Communications System Toolbox
comm.PSKTCMModulator*	Communications System Toolbox
comm.QAMCoarseFrequencyEstimator*	Communications System Toolbox
comm.QPSKDemodulator*	Communications System Toolbox
comm.QPSKModulator*	Communications System Toolbox
comm.RaisedCosineReceiveFilter*	Communications System Toolbox
comm.RaisedCosineTransmitFilter*	Communications System Toolbox
comm.RayleighChannel*	Communications System Toolbox
comm.RBDSWaveformGenerator*	Communications System Toolbox
comm.RectangularQAMDemodulator*	Communications System Toolbox
comm.RectangularModulator*	Communications System Toolbox

Name	Product
comm.RectangularQAMTCMDemodulator*	Communications System Toolbox
comm.RectangularQAMTCMModulator*	Communications System Toolbox
comm.RicianChannel*	Communications System Toolbox
comm.RSDecoder*	Communications System Toolbox
comm.RSEncoder*	Communications System Toolbox
comm.Scrambler*	Communications System Toolbox
comm.SphereDecoder*	Communications System Toolbox
comm.SymbolSynchronizer*	Communications System Toolbox
comm.ThermalNoise*	Communications System Toolbox
comm.TurboDecoder*	Communications System Toolbox
comm.TurboEncoder*	Communications System Toolbox
comm.ViterbiDecoder*	Communications System Toolbox
comm.WalshCode*	Communications System Toolbox
compan	MATLAB
complex	MATLAB
complex	Fixed-Point Designer
compressor*	Audio System Toolbox
computer*	MATLAB
cond	MATLAB
conj	MATLAB
conj	Fixed-Point Designer
conndef*	Image Processing Toolbox
constacc	Automated Driving System Toolbox
constaccjac	Automated Driving System Toolbox
constturn	Automated Driving System Toolbox
constturnjac	Automated Driving System Toolbox
constvel	Automated Driving System Toolbox
constveljac	Automated Driving System Toolbox

Name	Product
continue	MATLAB
conv*	MATLAB
conv*	Fixed-Point Designer
conv2	MATLAB
convenc	Communications System Toolbox
convergent	Fixed-Point Designer
convmtx	Signal Processing Toolbox
convn	MATLAB
cordicabs*	Fixed-Point Designer
cordicangle*	Fixed-Point Designer
cordicatan2*	Fixed-Point Designer
cordiccart2pol*	Fixed-Point Designer
cordiccexp*	Fixed-Point Designer
cordiccos*	Fixed-Point Designer
cordicpol2cart*	Fixed-Point Designer
cordicrotate*	Fixed-Point Designer
cordicsin*	Fixed-Point Designer
cordicsincos*	Fixed-Point Designer
cordicsqrt*	Fixed-Point Designer
cornerPoints*	Computer Vision System Toolbox
corrcoef*	MATLAB
corrmtx	Signal Processing Toolbox
cos	MATLAB
cos	Fixed-Point Designer
cosd	MATLAB
cosh	MATLAB
cot	MATLAB

Name	Product
cotd*	MATLAB
coth	MATLAB
cov*	MATLAB
cplxpair	MATLAB
cross*	MATLAB
crossoverFilter*	Audio System Toolbox
csc	MATLAB
cscd*	MATLAB
csch	MATLAB
ctmeas	Automated Driving System Toolbox
ctmeasjac	Automated Driving System Toolbox
ctranspose	MATLAB
ctranspose	Fixed-Point Designer
cummin	MATLAB
cummax	MATLAB
cumprod*	MATLAB
cumsum*	MATLAB
cumtrapz	MATLAB
cvmeas	Automated Driving System Toolbox
cvmeasjac	Automated Driving System Toolbox
db2pow	Signal Processing Toolbox
dct*	Signal Processing Toolbox
ddencmp*	Wavelet Toolbox
de2bi	Communications System Toolbox
deal	MATLAB
deblank*	MATLAB
dec2bin*	MATLAB

Name	Product
dec2hex*	MATLAB
dechirp*	Phased Array System Toolbox
deconv*	MATLAB
deg2rad	MATLAB
del2	MATLAB
delayseq*	Phased Array System Toolbox
demosaic*	Image Processing Toolbox
depressionang*	Phased Array System Toolbox
designMultirateFIR*	DSP System Toolbox
designParamEQ	Audio System Toolbox
designShelvingEQ	Audio System Toolbox
designVarSlopeFilter	Audio System Toolbox
det	MATLAB
detcoef*	Wavelet Toolbox
detcoef2*	Wavelet Toolbox
detectBRISKFeatures*	Computer Vision System Toolbox
detectCheckerboardPoints*	Computer Vision System Toolbox
detectFASTFeatures*	Computer Vision System Toolbox
detectHarrisFeatures*	Computer Vision System Toolbox
detectMinEigenFeatures*	Computer Vision System Toolbox
detectMSERFeatures*	Computer Vision System Toolbox
detectSURFFeatures*	Computer Vision System Toolbox
detrend*	MATLAB
diag*	MATLAB
diag*	Fixed-Point Designer
diagbfweights*	Phased Array System Toolbox
diff*	MATLAB

Name	Product
disparity*	Computer Vision System Toolbox
divide*	Fixed-Point Designer
dop2speed*	Phased Array System Toolbox
dopsteeringvec*	Phased Array System Toolbox
doppler*	Communications System Toolbox
dot	MATLAB
double	MATLAB
double*	Fixed-Point Designer
downsample	Signal Processing Toolbox
dpskdemod	Communications System Toolbox
dpskmod	Communications System Toolbox
dpss*	Signal Processing Toolbox
dsp.AdaptiveLatticeFilter*	DSP System Toolbox
dsp.AffineProjectionFilter*	DSP System Toolbox
dsp.AllpassFilter*	DSP System Toolbox
dsp.AllpoleFilter*	DSP System Toolbox
dsp.AnalyticSignal*	DSP System Toolbox
dsp.ArrayPlot*	DSP System Toolbox
dsp.ArrayVectorAdder*	DSP System Toolbox
dsp.ArrayVectorDivider*	DSP System Toolbox
dsp.ArrayVectorMultiplier*	DSP System Toolbox
dsp.ArrayVectorSubtractor*	DSP System Toolbox
dsp.AsyncBuffer*	
dsp.AudioFileReader*	DSP System Toolbox
dsp.AudioFileWriter*	DSP System Toolbox
dsp.Autocorrelator*	DSP System Toolbox
dsp.BinaryFileReader*	DSP System Toolbox
dsp.BinaryFileWriter*	DSP System Toolbox

Name	Product
dsp.BiquadFilter*	DSP System Toolbox
dsp.BurgAREstimator*	DSP System Toolbox
dsp.BurgSpectrumEstimator*	DSP System Toolbox
dsp.CepstralToLPC*	DSP System Toolbox
dsp.Channelizer*	DSP System Toolbox
dsp.ChannelSynthesizer*	DSP System Toolbox
dsp.CICCompensationDecimator*	DSP System Toolbox
dsp.CICCompensationInterpolator*	DSP System Toolbox
dsp.CICDecimator*	DSP System Toolbox
dsp.CICInterpolator*	DSP System Toolbox
dsp.Convolver*	DSP System Toolbox
dsp.Counter*	DSP System Toolbox
dsp.Crosscorrelator*	DSP System Toolbox
dsp.CrossSpectrumEstimator*	DSP System Toolbox
dsp.CumulativeProduct *	DSP System Toolbox
dsp.CumulativeSum*	DSP System Toolbox
dsp.DCBlocker*	DSP System Toolbox
dsp.DCT*	DSP System Toolbox
dsp.Delay*	DSP System Toolbox
dsp.DelayLine*	DSP System Toolbox
dsp.Differentiator*	DSP System Toolbox
dsp.DigitalDownConverter*	DSP System Toolbox
dsp.DigitalUpConverter*	DSP System Toolbox
dsp.DigitalFilter*	DSP System Toolbox
dsp.FarrowRateConverter*	DSP System Toolbox
dsp.FastTransversalFilter*	DSP System Toolbox
dsp.FFT*	DSP System Toolbox
dsp.FilterCascade*	DSP System Toolbox

Name	Product
dsp.FilteredXLMSFilter*	DSP System Toolbox
dsp.FIRDecimator*	DSP System Toolbox
dsp.FIRFilter*	DSP System Toolbox
dsp.FIRHalfbandDecimator*	DSP System Toolbox
dsp.FIRHalfbandInterpolator*	DSP System Toolbox
dsp.FIRInterpolator*	DSP System Toolbox
dsp.FIRRateConverter*	DSP System Toolbox
dsp.FrequencyDomainAdaptiveFilter*	DSP System Toolbox
dsp.HampelFilter*	
dsp.HighpassFilter*	DSP System Toolbox
dsp.Histogram*	DSP System Toolbox
dsp.IDCT*	DSP System Toolbox
dsp.IFFT*	DSP System Toolbox
dsp.IIRFilter*	DSP System Toolbox
dsp.IIRHalfbandDecimator*	DSP System Toolbox
dsp.IIRHalfbandInterpolator*	DSP System Toolbox
dsp.Interpolator*	DSP System Toolbox
dsp.KalmanFilter*	DSP System Toolbox
dsp.LDLFactor*	DSP System Toolbox
dsp.LevinsonSolver*	DSP System Toolbox
dsp.LMSFilter*	DSP System Toolbox
dsp.LowerTriangularSolver*	DSP System Toolbox
dsp.LowpassFilter*	DSP System Toolbox
dsp.LPCToAutocorrelation*	DSP System Toolbox
dsp.LPCToCepstral*	DSP System Toolbox
dsp.LPCToLSF*	DSP System Toolbox
dsp.LPCToLSP*	DSP System Toolbox
dsp.LPCToRC*	DSP System Toolbox

Name	Product
dsp.LSFToLPC*	DSP System Toolbox
dsp.LSPToLPC*	DSP System Toolbox
dsp.LUFactor*	DSP System Toolbox
dsp.Maximum*	DSP System Toolbox
dsp.Mean*	DSP System Toolbox
dsp.Median*	DSP System Toolbox
dsp.MedianFilter*	DSP System Toolbox
dsp.MovingAverage*	DSP System Toolbox
dsp.MovingMaximum*	DSP System Toolbox
dsp.MovingMinimum*	DSP System Toolbox
dsp.MovingRMS*	DSP System Toolbox
dsp.MovingStandardDeviation*	DSP System Toolbox
dsp.MovingVariance*	DSP System Toolbox
dsp.Minimum*	DSP System Toolbox
dsp.NCO*	DSP System Toolbox
dsp.Normalizer*	DSP System Toolbox
dsp.PeakFinder*	DSP System Toolbox
dsp.PeakToPeak*	DSP System Toolbox
dsp.PeakToRMS*	DSP System Toolbox
dsp.PhaseExtractor*	DSP System Toolbox
dsp.PhaseUnwrapper*	DSP System Toolbox
dsp.RCToAutocorrelation*	DSP System Toolbox
dsp.RCToLPC*	DSP System Toolbox
dsp.RMS*	DSP System Toolbox
dsp.RLSFilter*	DSP System Toolbox
dsp.SampleRateConverter*	DSP System Toolbox
dsp.ScalarQuantizerDecoder*	DSP System Toolbox
dsp.ScalarQuantizerEncoder*	DSP System Toolbox

Name	Product
dsp.SignalSource*	DSP System Toolbox
dsp.SineWave*	DSP System Toolbox
dsp.SpectrumAnalyzer*	DSP System Toolbox
dsp.SpectrumEstimator*	DSP System Toolbox
dsp.StandardDeviation*	DSP System Toolbox
dsp.StateLevels*	DSP System Toolbox
dsp.SubbandAnalysisFilter*	DSP System Toolbox
dsp.SubbandSynthesisFilter*	DSP System Toolbox
dsp.TimeScope*	DSP System Toolbox
dsp.TransferFunctionEstimator*	DSP System Toolbox
dsp.UDPReceiver*	DSP System Toolbox
dsp.UDPSender*	DSP System Toolbox
dsp.UpperTriangularSolver*	DSP System Toolbox
dsp.VariableBandwidthFIRFilter*	DSP System Toolbox
dsp.VariableBandwidthIIRFilter*	DSP System Toolbox
dsp.VariableFractionDelay*	DSP System Toolbox
dsp.VariableIntegerDelay*	DSP System Toolbox
dsp.Variance*	DSP System Toolbox
dsp.VectorQuantizerDecoder*	DSP System Toolbox
dsp.VectorQuantizerEncoder*	DSP System Toolbox
dsp.Window*	DSP System Toolbox
dsp.ZeroCrossingDetector*	DSP System Toolbox
dvbs2ldpc*	Communications System Toolbox
dwt*	Wavelet Toolbox
dwt2*	Wavelet Toolbox
dyadup*	Wavelet Toolbox
edge*	Image Processing Toolbox
effearthradius*	Phased Array System Toolbox

Name	Product
eig*	MATLAB
ellip*	Signal Processing Toolbox
ellipap*	Signal Processing Toolbox
ellipke	MATLAB
ellipord*	Signal Processing Toolbox
end	MATLAB
end	Fixed-Point Designer
envelope*	Signal Processing Toolbox
epipolarLine	Computer Vision System Toolbox
eps	MATLAB
eps*	Fixed-Point Designer
eq*	MATLAB
eq*	Fixed-Point Designer
erf	MATLAB
erfc	MATLAB
erfcinv	MATLAB
erfcx	MATLAB
erfinv	MATLAB
error*	MATLAB
espritdoa*	Phased Array System Toolbox
estimateEssentialMatrix*	Computer Vision System Toolbox
estimateFundamentalMatrix*	Computer Vision System Toolbox
estimateGeometricTransform*	Computer Vision System Toolbox
estimateUncalibratedRectification	Computer Vision System Toolbox
estimateWorldCameraPose*	Computer Vision System Toolbox
eul2quat	Robotics System Toolbox
eul2rotm	Robotics System Toolbox

Name	Product
eul2tform	Robotics System Toolbox
evcdf	Statistics and Machine Learning Toolbox
evinv	Statistics and Machine Learning Toolbox
evpdf	Statistics and Machine Learning Toolbox
evrnd*	Statistics and Machine Learning Toolbox
evstat	Statistics and Machine Learning Toolbox
exp	MATLAB
expander*	Audio System Toolbox
expcdf	Statistics and Machine Learning Toolbox
expint	MATLAB
expinv	Statistics and Machine Learning Toolbox
expm	MATLAB
expm1	MATLAB
exppdf	Statistics and Machine Learning Toolbox
exprnd	Statistics and Machine Learning Toolbox
expstat	Statistics and Machine Learning Toolbox
extendedKalmanFilter*	Control System Toolbox™
extendedKalmanFilter*	System Identification Toolbox™
extractFeatures*	Computer Vision System Toolbox
extractHOGFeatures*	Computer Vision System Toolbox

Name	Product
extractLBPFeatures*	Computer Vision System Toolbox
extrinsics*	Computer Vision System Toolbox
extrinsicsToCameraPose	Computer Vision System Toolbox
eye*	MATLAB
factor*	MATLAB
factorial	MATLAB
false*	MATLAB
fcdf	Statistics and Machine Learning Toolbox
fclose	MATLAB
feof	MATLAB
fft*	MATLAB
fft2	MATLAB
fftn*	MATLAB
fftshift	MATLAB
fi*	Fixed-Point Designer
fieldnames*	MATLAB
filter*	MATLAB
filter*	Fixed-Point Designer
filter2	MATLAB
filtfilt*	Signal Processing Toolbox
fimath*	Fixed-Point Designer
find*	MATLAB
finddelay	Signal Processing Toolbox
findpeaks	Signal Processing Toolbox
finv	Statistics and Machine Learning Toolbox
fir1*	Signal Processing Toolbox

Name	Product
fir2*	Signal Processing Toolbox
firceqrip*	DSP System Toolbox
fircls*	Signal Processing Toolbox
fircls1*	Signal Processing Toolbox
fireqint*	DSP System Toolbox
firgr*	DSP System Toolbox
firhalfband*	DSP System Toolbox
firlpnorm*	DSP System Toolbox
firls*	Signal Processing Toolbox
firminphase*	DSP System Toolbox
firnyquist*	DSP System Toolbox
firpr2chfb*	DSP System Toolbox
firpm*	Signal Processing Toolbox
firpmord*	Signal Processing Toolbox
fitgeotrans*	Image Processing Toolbox
fix	MATLAB
fix	Fixed-Point Designer
fixed.Quantizer	Fixed-Point Designer
flattopwin*	Signal Processing Toolbox
flintmax	MATLAB
flip*	MATLAB
flip*	Fixed-Point Designer
flipdim*	MATLAB
fliplr*	MATLAB
fliplr	Fixed-Point Designer
flipud*	MATLAB
flipud	Fixed-Point Designer

Name	Product
floor	MATLAB
floor	Fixed-Point Designer
fminbnd*	MATLAB
fminbnd*	Optimization Toolbox™
fminsearch*	MATLAB
fminsearch*	Optimization Toolbox
fogpl*	Phased Array System Toolbox
fopen*	MATLAB
for	MATLAB
for	Fixed-Point Designer
fpdf	Statistics and Machine Learning Toolbox
fprintf*	MATLAB
fread*	MATLAB
freqspace	MATLAB
freqz*	Signal Processing Toolbox
frewind	MATLAB
frnd*	Statistics and Machine Learning Toolbox
fseek*	MATLAB
fspecial*	Image Processing Toolbox
fspl*	Phased Array System Toolbox
fstat	Statistics and Machine Learning Toolbox
ftell*	MATLAB
full	MATLAB
fwrite*	MATLAB
fzero*	MATLAB

Name	Product
fzero*	Optimization Toolbox
gain2aperture*	Phased Array System Toolbox
gamcdf	Statistics and Machine Learning Toolbox
gaminv	Statistics and Machine Learning Toolbox
gamma	MATLAB
gammainc*	MATLAB
gammaincinv*	MATLAB
gammaln	MATLAB
gampdf	Statistics and Machine Learning Toolbox
gamrnd*	Statistics and Machine Learning Toolbox
gamstat	Statistics and Machine Learning Toolbox
gaspl*	Phased Array System Toolbox
gausswin*	Signal Processing Toolbox
gccphat*	Phased Array System Toolbox
gcd	MATLAB
ge	MATLAB
ge*	Fixed-Point Designer
GeneralizedLinearModel* or CompactGeneralizedLinearModel*	Statistics and Machine Learning Toolbox
generateCheckerboardPoints*	Computer Vision System Toolbox
genqamdemod	Communications System Toolbox
geocdf	Statistics and Machine Learning Toolbox
geoinv	Statistics and Machine Learning Toolbox

Name	Product
geomean	Statistics and Machine Learning Toolbox
geopdf	Statistics and Machine Learning Toolbox
geornd*	Statistics and Machine Learning Toolbox
geostat	Statistics and Machine Learning Toolbox
get*	Fixed-Point Designer
getlsb	Fixed-Point Designer
getmsb	Fixed-Point Designer
getNumInputs*	MATLAB
getNumOutputs*	MATLAB
getrangefromclass*	Image Processing Toolbox
getTrackPositions	Automated Driving System Toolbox
getTrackVelocities	Automated Driving System Toolbox
gevcdf	Statistics and Machine Learning Toolbox
gevinv	Statistics and Machine Learning Toolbox
gevpdf	Statistics and Machine Learning Toolbox
gevrnd*	Statistics and Machine Learning Toolbox
gevstat	Statistics and Machine Learning Toolbox
glmval*	Statistics and Machine Learning Toolbox
global2localcoord*	Phased Array System Toolbox
gpcdf	Statistics and Machine Learning Toolbox

Name	Product
gpinv	Statistics and Machine Learning Toolbox
gppdf	Statistics and Machine Learning Toolbox
gprnd*	Statistics and Machine Learning Toolbox
gpstat	Statistics and Machine Learning Toolbox
gradient	MATLAB
gray2bin	Communications System Toolbox
grayconnected*	Image Processing Toolbox
grazingang*	Phased Array System Toolbox
gt	MATLAB
gt*	Fixed-Point Designer
hadamard*	MATLAB
hamming*	Signal Processing Toolbox
hankel	MATLAB
hann*	Signal Processing Toolbox
harmmean	Statistics and Machine Learning Toolbox
hex2dec*	MATLAB
hex2num*	MATLAB
hilb	MATLAB
hilbert	Signal Processing Toolbox
hist*	MATLAB
histc*	MATLAB
histcounts*	MATLAB
histeq*	Image Processing Toolbox
hom2cart	Robotics System Toolbox

Name	Product
horizonrange*	Phased Array System Toolbox
horzcat	Fixed-Point Designer
hough*	Image Processing Toolbox
houghlines*	Image Processing Toolbox
houghpeaks*	Image Processing Toolbox
hygecdf	Statistics and Machine Learning Toolbox
hygeinv	Statistics and Machine Learning Toolbox
hygepdf	Statistics and Machine Learning Toolbox
hygernd*	Statistics and Machine Learning Toolbox
hygestat	Statistics and Machine Learning Toolbox
hypot	MATLAB
icdf	Statistics and Machine Learning Toolbox
idct*	Signal Processing Toolbox
if, elseif, else	MATLAB
idivide*	MATLAB
idwt*	Wavelet Toolbox
idwt2*	Wavelet Toolbox
ifft*	MATLAB
ifft2*	MATLAB
ifftn*	MATLAB
ifftshift	MATLAB
ifir*	DSP System Toolbox
iircomb*	DSP System Toolbox

Name	Product
iirgrpdelay*	DSP System Toolbox
iirlpnorm*	DSP System Toolbox
iirlpnormc*	DSP System Toolbox
iirnotch*	DSP System Toolbox
iirpeak*	DSP System Toolbox
im2double	MATLAB
im2int16*	Image Processing Toolbox
im2single*	Image Processing Toolbox
im2uint8*	Image Processing Toolbox
im2uint16*	Image Processing Toolbox
imabsdiff*	Image Processing Toolbox
imadjust*	Image Processing Toolbox
imag	MATLAB
imag	Fixed-Point Designer
imaq.VideoDevice*	Image Acquisition Toolbox™
imbinarize*	Image Processing Toolbox
imbothat*	Image Processing Toolbox
imboxfilt*	Image Processing Toolbox
imclearborder*	Image Processing Toolbox
imclose*	Image Processing Toolbox
imcomplement*	Image Processing Toolbox
imcrop*	Image Processing Toolbox
imdilate*	Image Processing Toolbox
imerode*	Image Processing Toolbox
imextendedmax*	Image Processing Toolbox
imextendedmin*	Image Processing Toolbox
imfill*	Image Processing Toolbox

Name	Product
imfilter*	Image Processing Toolbox
imfindcircles*	Image Processing Toolbox
imgaborfilt*	Image Processing Toolbox
imgaussfilt*	Image Processing Toolbox
imgradient3*	Image Processing Toolbox
imgradientxyz*	Image Processing Toolbox
imhist*	Image Processing Toolbox
imhmax*	Image Processing Toolbox
imhmin*	Image Processing Toolbox
imlincomb*	Image Processing Toolbox
immse*	Image Processing Toolbox
imodwpt*	Wavelet Toolbox
imodwt*	Wavelet Toolbox
imopen*	Image Processing Toolbox
imoverlay*	Image Processing Toolbox
impyramid*	Image Processing Toolbox
imquantize*	Image Processing Toolbox
imread*	Image Processing Toolbox
imreconstruct*	Image Processing Toolbox
imref2d*	Image Processing Toolbox
imref3d*	Image Processing Toolbox
imregionalmax*	Image Processing Toolbox
imregionalmin*	Image Processing Toolbox
imresize*	Image Processing Toolbox
imrotate*	Image Processing Toolbox
imtophat*	Image Processing Toolbox
imtranslate*	Image Processing Toolbox
imwarp*	Image Processing Toolbox

Name	Product
`ind2sub*`	MATLAB
`inf*`	MATLAB
`initcaekf`	Automated Driving System Toolbox
`initcakf`	Automated Driving System Toolbox
`initcaukf`	Automated Driving System Toolbox
`initctekf`	Automated Driving System Toolbox
`initctukf`	Automated Driving System Toolbox
`initcvekf`	Automated Driving System Toolbox
`initcvkf`	Automated Driving System Toolbox
`initcvukf`	Automated Driving System Toolbox
`inpolygon*`	MATLAB
`insertMarker*`	Computer Vision System Toolbox
`insertObjectAnnotation*`	Computer Vision System Toolbox
`insertShape*`	Computer Vision System Toolbox
`insertText*`	Computer Vision System Toolbox
`int8`, `int16`, `int32`, `int64`	MATLAB
`int8`, `int16`, `int32`, `int64`	Fixed-Point Designer
`integralBoxFilter*`	Image Processing Toolbox
`integralImage`	Computer Vision System Toolbox
`integratedLoudness`	Audio System Toolbox
`interp1*`	MATLAB
`interp1q*`	MATLAB
`interp2*`	MATLAB
`interp3*`	MATLAB
`interpn*`	MATLAB
`intersect*`	MATLAB
`intfilt*`	Signal Processing Toolbox

Name	Product
intlut*	Image Processing Toolbox
intmax	MATLAB
intmin	MATLAB
inv*	MATLAB
invhilb	MATLAB
ipermute*	MATLAB
ipermute	Fixed-Point Designer
iptcheckconn*	Image Processing Toolbox
iptcheckmap*	Image Processing Toolbox
iqcoef2imbal	Communications System Toolbox
iqimbal	Communications System Toolbox
iqimbal2coef	Communications System Toolbox
iqr	Statistics and Machine Learning Toolbox
isa	MATLAB
isbanded	MATLAB
iscell	MATLAB
iscellstr	MATLAB
ischar	MATLAB
iscolumn	MATLAB
iscolumn	Fixed-Point Designer
isdeployed*	MATLAB Compiler
isdiag	MATLAB
isempty	MATLAB
isempty	Fixed-Point Designer
isenum	MATLAB
isEpipoleInImage	Computer Vision System Toolbox
isequal	MATLAB

Name	Product
isequal	Fixed-Point Designer
isequaln	MATLAB
isfi*	Fixed-Point Designer
isfield*	MATLAB
isfimath	Fixed-Point Designer
isfimathlocal	Fixed-Point Designer
isfinite	MATLAB
isfinite	Fixed-Point Designer
isfloat	MATLAB
ishermitian	MATLAB
isinf	MATLAB
isinf	Fixed-Point Designer
isinteger	MATLAB
isletter*	MATLAB
isLocked*	MATLAB
islogical	MATLAB
ismac*	MATLAB
ismatrix	MATLAB
ismcc*	MATLAB Compiler
ismember*	MATLAB
ismethod	MATLAB
isnan	MATLAB
isnan	Fixed-Point Designer
isnumeric	MATLAB
isnumeric	Fixed-Point Designer
isnumerictype	Fixed-Point Designer
isobject	MATLAB

Name	Product
ispc*	MATLAB
isprime*	MATLAB
isreal	MATLAB
isreal	Fixed-Point Designer
isrow	MATLAB
isrow	Fixed-Point Designer
isscalar	MATLAB
isscalar	Fixed-Point Designer
issigned	Fixed-Point Designer
issorted*	MATLAB
isspace*	MATLAB
issparse	MATLAB
isstrprop*	MATLAB
isstruct	MATLAB
issymmetric	MATLAB
istrellis	Communications System Toolbox
istril	MATLAB
istriu	MATLAB
isunix*	MATLAB
isvector	MATLAB
isvector	Fixed-Point Designer
kaiser	Signal Processing Toolbox
kaiserord	Signal Processing Toolbox
kron	MATLAB
kmeans*	Statistics and Machine Learning Toolbox
kurtosis	Statistics and Machine Learning Toolbox

Name	Product
lab2rgb*	Image Processing Toolbox
label2idx*	Image Processing Toolbox
label2rgb*	Image Processing Toolbox
lcm	MATLAB
lcmvweights*	Phased Array System Toolbox
ldivide	MATLAB
le	MATLAB
le*	Fixed-Point Designer
length	MATLAB
length	Fixed-Point Designer
levinson*	Signal Processing Toolbox
limiter*	Audio System Toolbox
LinearModel* or CompactLinearModel*	Statistics and Machine Learning Toolbox
lineToBorderPoints	Computer Vision System Toolbox
linsolve*	MATLAB
linspace	MATLAB
load*	MATLAB
loadCompactModel*	Statistics and Machine Learning Toolbox
local2globalcoord*	Phased Array System Toolbox
log*	MATLAB
log2	MATLAB
log10	MATLAB
log1p	MATLAB
logical	MATLAB
logical	Fixed-Point Designer

Name	Product
logncdf	Statistics and Machine Learning Toolbox
logninv	Statistics and Machine Learning Toolbox
lognpdf	Statistics and Machine Learning Toolbox
lognrnd*	Statistics and Machine Learning Toolbox
lognstat	Statistics and Machine Learning Toolbox
logspace	MATLAB
loudnessMeter*	Audio System Toolbox
lower*	MATLAB
lowerbound	Fixed-Point Designer
lsb*	Fixed-Point Designer
lsqnonneg*	MATLAB
lsqnonneg*	Optimization Toolbox
lt	MATLAB
lt*	Fixed-Point Designer
lteZadoffChuSeq	Communications System Toolbox
lu	MATLAB
mad*	Statistics and Machine Learning Toolbox
magic*	MATLAB
matchFeatures*	Computer Vision System Toolbox
max*	MATLAB
max	Fixed-Point Designer
maxflat*	Signal Processing Toolbox
mdltest*	Phased Array System Toolbox

Name	Product
`mean*`	MATLAB
`mean`	Fixed-Point Designer
`mean2*`	Image Processing Toolbox
`medfilt2*`	Image Processing Toolbox
`median*`	MATLAB
`median`	Fixed-Point Designer
`meshgrid`	MATLAB
`mfilename`	MATLAB
`min*`	MATLAB
`min`	Fixed-Point Designer
`minus`	MATLAB
`minus*`	Fixed-Point Designer
`mkpp*`	MATLAB
`mldivide`	MATLAB
`mnpdf`	Statistics and Machine Learning Toolbox
`mod*`	MATLAB
`mode*`	MATLAB
`modwpt*`	Wavelet Toolbox
`modwptdetails*`	Wavelet Toolbox
`modwt*`	Wavelet Toolbox
`modwtmra*`	Wavelet Toolbox
`moment*`	Statistics and Machine Learning Toolbox
`mpower*`	MATLAB
`mpower*`	Fixed-Point Designer
`mpy*`	Fixed-Point Designer
`mrdivide`	MATLAB

Name	Product
mrdivide	Fixed-Point Designer
MSERRegions*	Computer Vision System Toolbox
mtimes*	MATLAB
mtimes*	Fixed-Point Designer
multibandParametricEQ*	Audio System Toolbox
multiObjectTracker*	Automated Driving System Toolbox
multithresh*	Image Processing Toolbox
musicdoa*	Phased Array System Toolbox
mvdrweights*	Phased Array System Toolbox
NaN or nan*	MATLAB
nancov*	Statistics and Machine Learning Toolbox
nanmax	Statistics and Machine Learning Toolbox
nanmean	Statistics and Machine Learning Toolbox
nanmedian	Statistics and Machine Learning Toolbox
nanmin	Statistics and Machine Learning Toolbox
nanstd	Statistics and Machine Learning Toolbox
nansum	Statistics and Machine Learning Toolbox
nanvar	Statistics and Machine Learning Toolbox
nargin*	MATLAB
narginchk	MATLAB
nargout*	MATLAB
nargoutchk	MATLAB

Name	Product
nbincdf	Statistics and Machine Learning Toolbox
nbininv	Statistics and Machine Learning Toolbox
nbinpdf	Statistics and Machine Learning Toolbox
nbinrnd*	Statistics and Machine Learning Toolbox
nbinstat	Statistics and Machine Learning Toolbox
ncfcdf	Statistics and Machine Learning Toolbox
ncfinv	Statistics and Machine Learning Toolbox
ncfpdf	Statistics and Machine Learning Toolbox
ncfrnd*	Statistics and Machine Learning Toolbox
ncfstat	Statistics and Machine Learning Toolbox
nchoosek*	MATLAB
nctcdf	Statistics and Machine Learning Toolbox
nctinv	Statistics and Machine Learning Toolbox
nctpdf	Statistics and Machine Learning Toolbox
nctrnd*	Statistics and Machine Learning Toolbox
nctstat	Statistics and Machine Learning Toolbox

Name	Product
ncx2cdf	Statistics and Machine Learning Toolbox
ncx2rnd*	Statistics and Machine Learning Toolbox
ncx2stat	Statistics and Machine Learning Toolbox
ndgrid	MATLAB
ndims	MATLAB
ndims	Fixed-Point Designer
ne*	MATLAB
ne*	Fixed-Point Designer
nearest	Fixed-Point Designer
nextpow2	MATLAB
nnz	MATLAB
noiseGate*	Audio System Toolbox
noisepow*	Phased Array System Toolbox
nonzeros	MATLAB
norm	MATLAB
normcdf	Statistics and Machine Learning Toolbox
normest	MATLAB
norminv	Statistics and Machine Learning Toolbox
normpdf	Statistics and Machine Learning Toolbox
normrnd*	Statistics and Machine Learning Toolbox
normstat	Statistics and Machine Learning Toolbox
not	MATLAB

Name	Product
npwgnthresh*	Phased Array System Toolbox
nthroot	MATLAB
null*	MATLAB
num2hex	MATLAB
numberofelements*	Fixed-Point Designer
numel	MATLAB
numel	Fixed-Point Designer
numerictype*	Fixed-Point Designer
nuttallwin*	Signal Processing Toolbox
objectDetection	Automated Driving System Toolbox
ocr*	Computer Vision System Toolbox
ocrText*	Computer Vision System Toolbox
oct2dec	Communications System Toolbox
octaveFilter*	Audio System Toolbox
ode23*	MATLAB
ode45 *	MATLAB
odeget *	MATLAB
odeset *	MATLAB
offsetstrel*	Image Processing Toolbox
ones *	MATLAB
opticalFlowFarneback*	Computer Vision System Toolbox
opticalFlowHS*	Computer Vision System Toolbox
opticalFlowLK*	Computer Vision System Toolbox
opticalFlowLKDoG*	Computer Vision System Toolbox
optimget*	MATLAB
optimget*	Optimization Toolbox
optimset*	MATLAB
optimset*	Optimization Toolbox

Name	Product
ordfilt2*	Image Processing Toolbox
or	MATLAB
orth*	MATLAB
otsuthresh*	Image Processing Toolbox
padarray*	Image Processing Toolbox
pambgfun*	Phased Array System Toolbox
parfor*	MATLAB
parzenwin*	Signal Processing Toolbox
pascal	MATLAB
pca*	Statistics and Machine Learning Toolbox
pchip*	MATLAB
pdf	Statistics and Machine Learning Toolbox
peak2peak	Signal Processing Toolbox
peak2rms	Signal Processing Toolbox
pearsrnd*	Statistics and Machine Learning Toolbox
permute*	MATLAB
permute*	Fixed-Point Designer
phased.ADPCACanceller*	Phased Array System Toolbox
phased.AngleDopplerResponse*	Phased Array System Toolbox
phased.ArrayGain*	Phased Array System Toolbox
phased.ArrayResponse*	Phased Array System Toolbox
phased.BackscatterRadarTarget*	Phased Array System Toolbox
phased.BackScatterSonarTarget*	Phased Array System Toolbox
phased.BarrageJammer*	Phased Array System Toolbox
phased.BeamscanEstimator*	Phased Array System Toolbox

Name	Product
phased.BeamscanEstimator2D*	Phased Array System Toolbox
phased.BeamspaceESPRITEstimator*	Phased Array System Toolbox
phased.CFARDetector*	Phased Array System Toolbox
phased.CFARDetector2D*	Phased Array System Toolbox
phased.Collector*	Phased Array System Toolbox
phased.ConformalArray*	Phased Array System Toolbox
phased.ConstantGammaClutter*	Phased Array System Toolbox
phased.CosineAntennaElement*	Phased Array System Toolbox
phased.CrossedDipoleAntennaElement*	Phased Array System Toolbox
phased.CustomAntennaElement*	Phased Array System Toolbox
phased.CustomMicrophoneElement*	Phased Array System Toolbox
phased.DopplerEstimator	Phased Array System Toolbox
phased.DPCACanceller*	Phased Array System Toolbox
phased.ElementDelay*	Phased Array System Toolbox
phased.ESPRITEstimator*	Phased Array System Toolbox
phased.FMCWWaveform*	Phased Array System Toolbox
phased.FreeSpace*	Phased Array System Toolbox
phased.FrostBeamformer*	Phased Array System Toolbox
phased.GSCBeamformer*	Phased Array System Toolbox
phased.GCCEstimator*	Phased Array System Toolbox
phased.IsoSpeedUnderWaterPaths*	Phased Array System Toolbox
phased.IsotropicAntennaElement*	Phased Array System Toolbox
phased.IsotropicHydrophone*	Phased Array System Toolbox
phased.IsotropicProjector*	Phased Array System Toolbox
phased.LCMVBeamformer*	Phased Array System Toolbox
phased.LOSChannel*	Phased Array System Toolbox
phased.LinearFMWaveform*	Phased Array System Toolbox
phased.MatchedFilter*	Phased Array System Toolbox

Name	Product
phased.MFSKWaveform*	Phased Array System Toolbox
phased.MUSICEstimator*	Phased Array System Toolbox
phased.MUSICEstimator2D*	Phased Array System Toolbox
phased.MVDRBeamformer*	Phased Array System Toolbox
phased.MVDREstimator*	Phased Array System Toolbox
phased.MVDREstimator2D*	Phased Array System Toolbox
phased.MultipathChannel*	Phased Array System Toolbox
phased.OmnidirectionalMicrophoneElement*	Phased Array System Toolbox
phased.PartitionedArray*	Phased Array System Toolbox
phased.PhaseCodedWaveform*	Phased Array System Toolbox
phased.PhaseShiftBeamformer*	Phased Array System Toolbox
phased.Platform*	Phased Array System Toolbox
phased.RadarTarget*	Phased Array System Toolbox
phased.Radiator*	Phased Array System Toolbox
phased.RangeDopplerResponse*	Phased Array System Toolbox
phased.RangeEstimator*	Phased Array System Toolbox
phased.RangeResponse*	Phased Array System Toolbox
phased.RectangularWaveform*	Phased Array System Toolbox
phased.ReceiverPreamp*	Phased Array System Toolbox
phased.ReplicatedSubarray*	Phased Array System Toolbox
phased.RootMUSICEstimator*	Phased Array System Toolbox
phased.RootWSFEstimator	Phased Array System Toolbox
phased.ScatteringMIMOChannel*	Phased Array System Toolbox
phased.ShortDipoleAntennaElement*	Phased Array System Toolbox
phased.STAPSMIBeamformer*	Phased Array System Toolbox
phased.SteeringVector*	Phased Array System Toolbox
phased.SteppedFMWaveform*	Phased Array System Toolbox
phased.StretchProcessor*	Phased Array System Toolbox

Name	Product
phased.SubbandMVDRBeamformer*	Phased Array System Toolbox
phased.SubbandPhaseShiftBeamformer*	Phased Array System Toolbox
phased.SumDifferenceMonopulseTracker*	Phased Array System Toolbox
phased.SumDifferenceMonopulseTracker2D*	Phased Array System Toolbox
phased.TimeDelayBeamformer*	Phased Array System Toolbox
phased.TimeDelayLCMVBeamformer*	Phased Array System Toolbox
phased.TimeVaryingGain*	Phased Array System Toolbox
phased.Transmitter*	Phased Array System Toolbox
phased.TwoRayChannel*	Phased Array System Toolbox
phased.UCA*	Phased Array System Toolbox
phased.ULA*	Phased Array System Toolbox
phased.URA*	Phased Array System Toolbox
phased.WidebandBackscatterRadarTarget*	Phased Array System Toolbox
phased.WidebandCollector*	Phased Array System Toolbox
phased.WidebandFreeSpace	Phased Array System Toolbox
phased.WidebandLOSChannel*	Phased Array System Toolbox
phased.WidebandRadiator*	Phased Array System Toolbox
phased.WidebandTwoRayChannel*	Phased Array System Toolbox
phitheta2azel*	Phased Array System Toolbox
phitheta2azelpat*	Phased Array System Toolbox
phitheta2uv*	Phased Array System Toolbox
phitheta2uvpat*	Phased Array System Toolbox
physconst*	Phased Array System Toolbox
pi	MATLAB
pilotcalib*	Phased Array System Toolbox
pinv	MATLAB
planerot*	MATLAB
plus	MATLAB

Name	Product
plus*	Fixed-Point Designer
poisscdf	Statistics and Machine Learning Toolbox
poissinv	Statistics and Machine Learning Toolbox
poisspdf	Statistics and Machine Learning Toolbox
poissrnd*	Statistics and Machine Learning Toolbox
poisstat	Statistics and Machine Learning Toolbox
pol2cart	MATLAB
pol2circpol*	Phased Array System Toolbox
polellip*	Phased Array System Toolbox
polloss*	Phased Array System Toolbox
polratio*	Phased Array System Toolbox
polsignature*	Phased Array System Toolbox
poly*	MATLAB
polyarea	MATLAB
poly2trellis	Communications System Toolbox
polyder*	MATLAB
polyeig*	MATLAB
polyfit*	MATLAB
polyint	MATLAB
polyval	MATLAB
polyvalm	MATLAB
pow2	Fixed-Point Designer
pow2db	Signal Processing Toolbox
power*	MATLAB

Name	Product
power*	Fixed-Point Designer
ppval*	MATLAB
prctile*	Statistics and Machine Learning Toolbox
predict* method of ClassificationECOC and CompactClassificationECOC	Statistics and Machine Learning Toolbox
predict* method of ClassificationEnsemble, ClassificationBaggedEnsemble, and CompactClassificationEnsemble	Statistics and Machine Learning Toolbox
predict* method of ClassificationLinear	Statistics and Machine Learning Toolbox
predict* method of ClassificationSVM and CompactClassificationSVM	Statistics and Machine Learning Toolbox
predict* method of ClassificationTree and CompactClassificationTree	Statistics and Machine Learning Toolbox
predict* method of GeneralizedLinearModel and CompactGeneralizedLinearModel	Statistics and Machine Learning Toolbox
predict* method of LinearModel and CompactLinearModel	Statistics and Machine Learning Toolbox
predict* method of RegressionTree or CompactRegressionTree	Statistics and Machine Learning Toolbox
primes*	MATLAB
prod*	MATLAB
projective2d*	Image Processing Toolbox
psi	MATLAB
psnr*	Image Processing Toolbox
pulsint*	Phased Array System Toolbox
qamdemod	Communications System Toolbox
qammod	Communications System Toolbox
qmf*	Wavelet Toolbox

Name	Product
qr	MATLAB
qr	Fixed-Point Designer
quad2d*	MATLAB
quadgk	MATLAB
quantile	Statistics and Machine Learning Toolbox
quantize	Fixed-Point Designer
quat2axang	Robotics System Toolbox
quat2eul	Robotics System Toolbox
quat2rotm	Robotics System Toolbox
quat2tform	Robotics System Toolbox
quatconj*	Aerospace Toolbox
quatdivide*	Aerospace Toolbox
quatinv*	Aerospace Toolbox
quatmod*	Aerospace Toolbox
quatmultiply*	Aerospace Toolbox
quatnorm*	Aerospace Toolbox
quatnormalize*	Aerospace Toolbox
rad2deg	MATLAB
radareqpow*	Phased Array System Toolbox
radareqrng*	Phased Array System Toolbox
radareqsnr*	Phased Array System Toolbox
radarvcd*	Phased Array System Toolbox
radialspeed*	Phased Array System Toolbox
rainpl*	Phased Array System Toolbox
rand*	MATLAB
randg	Statistics and Machine Learning Toolbox

Name	Product
randi*	MATLAB
randn*	MATLAB
random	Statistics and Machine Learning Toolbox
random* method of GeneralizedLinearModel and CompactGeneralizedLinearModel	Statistics and Machine Learning Toolbox
random* method of LinearModel and CompactLinearModel	Statistics and Machine Learning Toolbox
randperm	MATLAB
randsample*	Statistics and Machine Learning Toolbox
range	Fixed-Point Designer
range2beat*	Phased Array System Toolbox
range2bw*	Phased Array System Toolbox
range2time*	Phased Array System Toolbox
rangeangle*	Phased Array System Toolbox
rank	MATLAB
raylcdf	Statistics and Machine Learning Toolbox
raylinv	Statistics and Machine Learning Toolbox
raylpdf	Statistics and Machine Learning Toolbox
raylrnd*	Statistics and Machine Learning Toolbox
raylstat	Statistics and Machine Learning Toolbox
rcond	MATLAB
rcosdesign*	Signal Processing Toolbox
rdcoupling*	Phased Array System Toolbox

Name	Product
rdivide	MATLAB
rdivide	Fixed-Point Designer
real	MATLAB
real	Fixed-Point Designer
reallog	MATLAB
realmax	MATLAB
realmax	Fixed-Point Designer
realmin	MATLAB
realmin	Fixed-Point Designer
realpow	MATLAB
realsqrt	MATLAB
reconstructScene*	Computer Vision System Toolbox
rectifyStereoImages*	Computer Vision System Toolbox
rectint	MATLAB
rectwin*	Signal Processing Toolbox
recursiveAR*	System Identification Toolbox
recursiveARMA*	System Identification Toolbox
recursiveARMAX*	System Identification Toolbox
recursiveARX*	System Identification Toolbox
recursiveBJ*	System Identification Toolbox
recursiveLS*	System Identification Toolbox
recursiveOE*	System Identification Toolbox
regionprops*	Image Processing Toolbox
RegressionTree* or CompactRegressionTree*	Statistics and Machine Learning Toolbox
reinterpretcast	Fixed-Point Designer
relativeCameraPose*	Computer Vision System Toolbox
release*	MATLAB

Name	Product
rem*	MATLAB
removefimath	Fixed-Point Designer
repelem*	MATLAB
repmat*	MATLAB
repmat*	Fixed-Point Designer
resample*	Signal Processing Toolbox
rescale	Fixed-Point Designer
reset*	MATLAB
reshape*	MATLAB
reshape	Fixed-Point Designer
return	MATLAB
reverberator*	Audio System Toolbox
rgb2gray	MATLAB
rgb2lab*	Image Processing Toolbox
rgb2ycbcr*	Image Processing Toolbox
rms	Signal Processing Toolbox
rng*	MATLAB
robotics.AimingConstraint	Robotics System Toolbox
robotics.BinaryOccupancyGrid*	Robotics System Toolbox
robotics.Cartesianbounds	Robotics System Toolbox
robotics.GeneralizedInverseKinematics*	Robotics System Toolbox
robotics.InverseKinematics*	Robotics System Toolbox
robotics.Joint	Robotics System Toolbox
robotics.JointPositionBounds	Robotics System Toolbox
robotics.OccupancyGrid*	Robotics System Toolbox
robotics.OdometryMotionModel*	Robotics System Toolbox
robotics.OrientationTarget	Robotics System Toolbox
robotics.ParticleFilter*	Robotics System Toolbox

Name	Product
robotics.PoseTarget	Robotics System Toolbox
robotics.PositionTarget	Robotics System Toolbox
robotics.PRM*	Robotics System Toolbox
robotics.PurePursuit*	Robotics System Toolbox
robotics.RigidBody	Robotics System Toolbox
robotics.RigidBodyTree*	Robotics System Toolbox
robotics.VectorFieldHistogram*	Robotics System Toolbox
rocpfa*	Phased Array System Toolbox
rocsnr*	Phased Array System Toolbox
rootmusicdoa*	Phased Array System Toolbox
roots*	MATLAB
rosser	MATLAB
rot90*	MATLAB
rot90*	Fixed-Point Designer
rotationMatrixToVector	Computer Vision System Toolbox
rotationVectorToMatrix	Computer Vision System Toolbox
rotm2axang	Robotics System Toolbox
rotm2eul	Robotics System Toolbox
rotm2quat	Robotics System Toolbox
rotm2tform	Robotics System Toolbox
rotx*	Phased Array System Toolbox
roty*	Phased Array System Toolbox
rotz*	Phased Array System Toolbox
round*	MATLAB
round	Fixed-Point Designer
rsf2csf	MATLAB
rsgenpoly	Communications System Toolbox
rsgenpolycoeffs	Communications System Toolbox

Name	Product
schur*	MATLAB
sec	MATLAB
secd*	MATLAB
sech	MATLAB
selectStrongestBbox*	Computer Vision System Toolbox
sensorcov*	Phased Array System Toolbox
sensorsig*	Phased Array System Toolbox
setdiff*	MATLAB
setfimath	Fixed-Point Designer
setxor*	MATLAB
sfi*	Fixed-Point Designer
sgolay	Signal Processing Toolbox
sgolayfilt	Signal Processing Toolbox
shiftdim*	MATLAB
shiftdim*	Fixed-Point Designer
shnidman*	Phased Array System Toolbox
sign	MATLAB
sign	Fixed-Point Designer
sin	MATLAB
sin	Fixed-Point Designer
sinc	Signal Processing Toolbox
sind	MATLAB
single	MATLAB
single*	Fixed-Point Designer
sinh	MATLAB
size	MATLAB
size	Fixed-Point Designer

Name	Product
skewness	Statistics and Machine Learning Toolbox
sort*	MATLAB
sort*	Fixed-Point Designer
sortrows*	MATLAB
sosfilt	Signal Processing Toolbox
scatteringchanmtx*	Phased Array System Toolbox
speed2dop*	Phased Array System Toolbox
sph2cart	MATLAB
sph2cartvec*	Phased Array System Toolbox
spline*	MATLAB
spsmooth*	Phased Array System Toolbox
squeeze*	MATLAB
squeeze	Fixed-Point Designer
sqrt*	MATLAB
sqrt*	Fixed-Point Designer
sqrtm	MATLAB
std*	MATLAB
steervec*	Phased Array System Toolbox
step*	MATLAB
stereoAnaglyph	Computer Vision System Toolbox
stereoParameters*	Computer Vision System Toolbox
stokes*	Phased Array System Toolbox
storedInteger	Fixed-Point Designer
storedIntegerToDouble	Fixed-Point Designer
str2double*	MATLAB
str2func*	MATLAB
strcmp*	MATLAB

Name	Product
strcmpi*	MATLAB
strel*	Image Processing Toolbox
stretchfreq2rng*	Phased Array System Toolbox
stretchlim*	Image Processing Toolbox
strfind*	MATLAB
strjoin*	MATLAB
strjust*	MATLAB
strncmp*	MATLAB
strncmpi*	MATLAB
strrep*	MATLAB
strtok*	MATLAB
strtrim*	MATLAB
struct*	MATLAB
struct2cell*	MATLAB
structfun*	MATLAB
sub*	Fixed-Point Designer
sub2ind*	MATLAB
subsasgn	Fixed-Point Designer
subspace	MATLAB
subsref	Fixed-Point Designer
sum*	MATLAB
sum*	Fixed-Point Designer
superpixels*	Image Processing Toolbox
surfacegamma*	Phased Array System Toolbox
surfclutterrcs*	Phased Array System Toolbox
SURFPoints*	Computer Vision System Toolbox
svd*	MATLAB
swapbytes*	MATLAB

Name	Product
switch, case, otherwise*	MATLAB
systemp*	Phased Array System Toolbox
tan	MATLAB
tand*	MATLAB
tanh	MATLAB
taylortaperc*	Phased Array System Toolbox
taylorwin*	Signal Processing Toolbox
tcdf	Statistics and Machine Learning Toolbox
tf2ca*	DSP System Toolbox
tf2cl*	DSP System Toolbox
tform2axang	Robotics System Toolbox
tform2eul	Robotics System Toolbox
tform2quat	Robotics System Toolbox
tform2rotm	Robotics System Toolbox
tform2trvec	Robotics System Toolbox
thselect*	Wavelet Toolbox
time2range*	Phased Array System Toolbox
times*	MATLAB
times*	Fixed-Point Designer
tinv	Statistics and Machine Learning Toolbox
toeplitz	MATLAB
tpdf	Statistics and Machine Learning Toolbox
trace	MATLAB
trackingEKF	Automated Driving System Toolbox
trackingKF	Automated Driving System Toolbox

Name	Product
trackingUKF	Automated Driving System Toolbox
transformScan	Robotics System Toolbox
transpose	MATLAB
transpose	Fixed-Point Designer
trapz*	MATLAB
triang*	Signal Processing Toolbox
triangulate*	Computer Vision System Toolbox
tril*	MATLAB
tril*	Fixed-Point Designer
triu*	MATLAB
triu*	Fixed-Point Designer
trnd*	Statistics and Machine Learning Toolbox
true*	MATLAB
trvec2tform	Robotics System Toolbox
tstat	Statistics and Machine Learning Toolbox
tukeywin*	Signal Processing Toolbox
typecast*	MATLAB
ufi*	Fixed-Point Designer
uint8, uint16, uint32, uint64	MATLAB
uint8, uint16, uint32, uint64	Fixed-Point Designer
uminus	MATLAB
uminus	Fixed-Point Designer
undistortImage*	Computer Vision System Toolbox
unidcdf	Statistics and Machine Learning Toolbox
unidinv	Statistics and Machine Learning Toolbox

Name	Product
unidpdf	Statistics and Machine Learning Toolbox
unidrnd*	Statistics and Machine Learning Toolbox
unidstat	Statistics and Machine Learning Toolbox
unifcdf	Statistics and Machine Learning Toolbox
unifinv	Statistics and Machine Learning Toolbox
unifpdf	Statistics and Machine Learning Toolbox
unifrnd*	Statistics and Machine Learning Toolbox
unifstat	Statistics and Machine Learning Toolbox
unigrid*	Phased Array System Toolbox
union*	MATLAB
unique*	MATLAB
unmkpp*	MATLAB
unscentedKalmanFilter*	Control System Toolbox
unscentedKalmanFilter*	System Identification Toolbox
unwrap*	MATLAB
upfirdn*	Signal Processing Toolbox
uplus	MATLAB
uplus	Fixed-Point Designer
upper*	MATLAB
upperbound	Fixed-Point Designer
upsample*	Signal Processing Toolbox
uv2azel*	Phased Array System Toolbox

Name	Product
uv2azelpat*	Phased Array System Toolbox
uv2phitheta*	Phased Array System Toolbox
uv2phithetapat*	Phased Array System Toolbox
val2ind*	Phased Array System Toolbox
vander	MATLAB
var*	MATLAB
vertcat	Fixed-Point Designer
vision.AlphaBlender*	Computer Vision System Toolbox
vision.Autocorrelator*	Computer Vision System Toolbox
vision.BlobAnalysis*	Computer Vision System Toolbox
vision.CascadeObjectDetector*	Computer Vision System Toolbox
vision.ChromaResampler*	Computer Vision System Toolbox
vision.Convolver*	Computer Vision System Toolbox
vision.Crosscorrelator*	Computer Vision System Toolbox
vision.DemosaicInterpolator*	Computer Vision System Toolbox
vision.DCT*	Computer Vision System Toolbox
vision.Deinterlacer*	Computer Vision System Toolbox
vision.DeployableVideoPlayer *	Computer Vision System Toolbox
vision.FFT*	Computer Vision System Toolbox
vision.ForegroundDetector*	Computer Vision System Toolbox
vision.GammaCorrector*	Computer Vision System Toolbox
vision.GeometricShearer*	Computer Vision System Toolbox
vision.HistogramBasedTracker*	Computer Vision System Toolbox
vision.HoughLines*	Computer Vision System Toolbox
vision.IDCT*	Computer Vision System Toolbox
vision.IFFT*	Computer Vision System Toolbox
vision.ImageDataTypeConverter*	Computer Vision System Toolbox
vision.KalmanFilter*	Computer Vision System Toolbox

Name	Product
vision.LocalMaximaFinder*	Computer Vision System Toolbox
vision.MarkerInserter*	Computer Vision System Toolbox
vision.Maximum*	Computer Vision System Toolbox
vision.Median*	Computer Vision System Toolbox
vision.Mean*	Computer Vision System Toolbox
vision.Minimum*	Computer Vision System Toolbox
vision.PeopleDetector*	Computer Vision System Toolbox
vision.PointTracker*	Computer Vision System Toolbox
vision.Pyramid*	Computer Vision System Toolbox
vision.ShapeInserter*	Computer Vision System Toolbox
vision.StandardDeviation*	Computer Vision System Toolbox
vision.TemplateMatcher*	Computer Vision System Toolbox
vision.Variance*	Computer Vision System Toolbox
vision.VideoFileReader*	Computer Vision System Toolbox
vision.VideoFileWriter*	Computer Vision System Toolbox
vitdec	Communications System Toolbox
waterfill*	Phased Array System Toolbox
watershed*	Image Processing Toolbox
wavedec*	Wavelet Toolbox
wavedec2*	Wavelet Toolbox
waverec*	Wavelet Toolbox
waverec2*	Wavelet Toolbox
wavetableSynthesizer*	Audio System Toolbox
wblcdf	Statistics and Machine Learning Toolbox
wblinv	Statistics and Machine Learning Toolbox

Name	Product
wblpdf	Statistics and Machine Learning Toolbox
wblrnd*	Statistics and Machine Learning Toolbox
wblstat	Statistics and Machine Learning Toolbox
wden*	Wavelet Toolbox
wdencmp*	Wavelet Toolbox
weightingFilter*	Audio System Toolbox
wextend*	Wavelet Toolbox
while	MATLAB
wilkinson*	MATLAB
wlanCoarseCFOEstimate*	WLAN System Toolbox™
wlanDMGConfig*	WLAN System Toolbox
wlanFieldIndices*	WLAN System Toolbox
wlanFineCFOEstimate*	WLAN System Toolbox
wlanFormatDetect*	WLAN System Toolbox
wlanHTConfig*	WLAN System Toolbox
wlanHTData*	WLAN System Toolbox
wlanHTDataRecover*	WLAN System Toolbox
wlanHTLTFChannelEstimate*	WLAN System Toolbox
wlanHTLTFDemodulate*	WLAN System Toolbox
wlanHTLTF*	WLAN System Toolbox
wlanHTSIG*	WLAN System Toolbox
wlanHTSIGRecover*	WLAN System Toolbox
wlanHTSTF*	WLAN System Toolbox
wlanLLTF*	WLAN System Toolbox
wlanLLTFChannelEstimate*	WLAN System Toolbox

Name	Product
wlanLLTFDemodulate*	WLAN System Toolbox
wlanLSIG*	WLAN System Toolbox
wlanLSIGRecover*	WLAN System Toolbox
wlanLSTF*	WLAN System Toolbox
wlanNonHTConfig*	WLAN System Toolbox
wlanNonHTData*	WLAN System Toolbox
wlanNonHTDataRecover*	WLAN System Toolbox
wlanPacketDetect*	WLAN System Toolbox
wlanRecoveryConfig*	WLAN System Toolbox
wlanS1GConfig*	WLAN System Toolbox
wlanSymbolTimingEstimate*	WLAN System Toolbox
wlanTGacChannel*	WLAN System Toolbox
wlanTGahChannel*	WLAN System Toolbox
wlanTGnChannel*	WLAN System Toolbox
wlanVHTConfig*	WLAN System Toolbox
wlanVHTData*	WLAN System Toolbox
wlanVHTDataRecover*	WLAN System Toolbox
wlanVHTLTF*	WLAN System Toolbox
wlanVHTLTFChannelEstimate*	WLAN System Toolbox
wlanVHTLTFDemodulate*	WLAN System Toolbox
wlanVHTSIGA*	WLAN System Toolbox
wlanVHTSIGARecover*	WLAN System Toolbox
wlanVHTSIGBRecover*	WLAN System Toolbox
wlanVHTSIGB*	WLAN System Toolbox
wlanVHTSTF*	WLAN System Toolbox
wlanWaveformGenerator*	WLAN System Toolbox
wnoisest*	Wavelet Toolbox
wthcoef*	Wavelet Toolbox

Name	Product
wthcoef2*	Wavelet Toolbox
wthresh*	Wavelet Toolbox
xcorr*	Signal Processing Toolbox
xcorr2	Signal Processing Toolbox
xcov	Signal Processing Toolbox
xor	MATLAB
ycbcr2rgb*	Image Processing Toolbox
yulewalk*	Signal Processing Toolbox
zeros*	MATLAB
zp2tf	MATLAB
zscore	Statistics and Machine Learning Toolbox

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

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

For an alphabetical list of supported functions, classes, and System objects, see "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List" on page 3-2.

Note: For more information on code generation for fixed-point algorithms, refer to "Code Acceleration and Code Generation from MATLAB" (Fixed-Point Designer).

In this section...

In this section...

In this section...

Aerospace Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`quatconj*`
`quatdivide*`
`quatinv*`
`quatmod*`
`quatmultiply*`
`quatnorm*`
`quatnormalize*`

Arithmetic Operations in MATLAB

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`ctranspose`

idivide*
isa
ldivide
minus
mldivide
mpower*
mrdivide
mtimes*
plus
power*
rdivide
times*
transpose
uminus
uplus

Audio System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Name
Audio I/O and Waveform Generation
audioDeviceReader*
audioDeviceWriter*
audioPlayerRecorder*
wavetableSynthesizer*
audioOscillator*
Audio Processing Algorithm Design

Name
designVarSlopeFilter
designParamEQ
designShelvingEQ
integratedLoudness
crossoverFilter*
compressor*
expander*
noiseGate*
limiter*
multibandParametricEQ*
octaveFilter*
weightingFilter*
loudnessMeter*
reverberator*
Audio Plugins
audioPluginInterface
audioPluginParameter
audioPlugin
audioPluginSource

Automated Driving System Toolbox

C and C++ code generation for the following functions and classes requires the Automated Driving System Toolbox software.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

cameas
cameasjac
constacc

constaccjac
constturn
constturnjac
constvel
constveljac
ctmeas
ctmeasjac
cvmeas
cvmeasjac
getTrackPositions
getTrackVelocities
initcaekf
initcakf
initcaukf
initctekf
initctukf
initcvekf
initcvkf
initcvukf
multiObjectTracker*
objectDetection
trackingEKF
trackingKF
trackingUKF

Bit-Wise Operations MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

flintmax

swapbytes*

Casting in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

cast
char*
class
double
int8, int16, int32, int64
logical
single
typecast*
uint8, uint16, uint32, uint64

Character Functions in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

bin2dec*
blanks
char
deblank*
dec2bin*
dec2hex*
hex2dec*
hex2num*
iscellstr
ischar
isletter*

isspace*
isstrprop*
lower*
num2hex
str2double*
strcmp*
strcmpi*
strfind*
strjoin*
strjust*
strncmp*
strncmpi*
strrep*
strtok*
strtrim*
upper*

Communications System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Input and Output
comm.BasebandFileReader*
comm.BasebandFileWriter*
comm.BarkerCode*
comm.GoldSequence*
comm.HadamardCode*
comm.KasamiSequence*

comm.RBDSWaveformGenerator*
comm.WalshCode*
comm.PNSequence*
lteZadoffChuSeq
Signal and Delay Management
bi2de
de2bi
Display and Visual Analysis
comm.ConstellationDiagram*
comm.EyeDiagram*
dsp.ArrayPlot*
dsp.SpectrumAnalyzer*
dsp.TimeScope*
Source Coding
comm.DifferentialDecoder*
comm.DifferentialEncoder*
Cyclic Redundancy Check Coding
comm.CRCDetector*
comm.CRCGenerator*
comm.HDLCRCDetector*
comm.HDLCRCGenerator*
BCH Codes
bchgenpoly*
comm.BCHDecoder*
comm.BCHEncoder*
Reed-Solomon Codes
comm.RSDecoder*
comm.RSEncoder*
comm.HDLRSDecoder*

comm.HDLRSEncoder*
rsgenpoly*
rsgenpolycoeffs*
LDPC Codes
comm.LDPCDecoder*
comm.LDPCEncoder*
dvbs2ldpc*
Convolutional Coding
comm.APPDecoder*
comm.ConvolutionalEncoder*
comm.TurboDecoder*
comm.TurboEncoder*
comm.ViterbiDecoder*
convenc
istrellis
oct2dec
poly2trellis
vitdec
Signal Operations
bin2gray
comm.Descrambler*
comm.Scrambler*
gray2bin
Interleaving
comm.AlgebraicDeinterleaver*
comm.AlgebraicInterleaver*
comm.BlockDeinterleaver*
comm.BlockInterleaver*
comm.ConvolutionalDeinterleaver*

comm.ConvolutionalInterleaver*
comm.HelicalDeinterleaver*
comm.HelicalInterleaver*
comm.MatrixDeinterleaver*
comm.MatrixInterleaver*
comm.MatrixHelicalScanDeinterleaver*
comm.MatrixHelicalScanInterleaver*
comm.MultiplexedDeinterleaver*
comm.MultiplexedInterleaver*
Frequency Modulation
comm.FSKDemodulator*
comm.FSKModulator*
Phase Modulation
comm.BPSKDemodulator*
comm.BPSKModulator*
comm.DBPSKDemodulator*
comm.DBPSKModulator*
comm.DPSKDemodulator*
comm.DPSKModulator*
comm.DQPSKDemodulator*
comm.DQPSKModulator*
comm.OQPSKDemodulator*
comm.OQPSKModulator*
comm.PSKDemodulator*
comm.PSKModulator*
comm.QPSKDemodulator*
comm.QPSKModulator*
`dpskdemod`
`dpskmod`

Amplitude Modulation
comm.GeneralQAMDemodulator*
comm.GeneralQAMModulator*
comm.PAMDemodulator*
comm.PAMModulator*
comm.RectangularQAMDemodulator*
comm.RectangularQAMModulator*
genqamdemod
qammod
qamdemod
Continuous Phase Modulation
comm.CPFSKDemodulator*
comm.CPFSKModulator*
comm.CPMDemodulator*
comm.CPMModulator*
comm.GMSKDemodulator*
comm.GMSKModulator*
comm.MSKDemodulator*
comm.MSKModulator*
Trellis Coded Modulation
comm.GeneralQAMTCMDemodulator*
comm.GeneralQAMTCMModulator*
comm.PSKTCMDemodulator*
comm.PSKTCMModulator*
comm.RectangularQAMTCMDemodulator*
comm.RectangularQAMTCMModulator*
Orthogonal Frequency-Division Modulation
comm.OFDMDemodulator*
comm.OFDMModulator*

Analog Baseband Modulation
comm.FMBroadcastDemodulator*
comm.FMBroadcastModulator*
comm.FMDemodulator*
comm.FMModulator*
Filtering
comm.IntegrateAndDumpFilter*
comm.RaisedCosineReceiveFilter*
comm.RaisedCosineTransmitFilter*
Carrier Phase Synchronization
comm.CarrierSynchronizer*
comm.CPMCarrierPhaseSynchronizer*
comm.CoarseFrequencyCompensator*
Timing Phase Synchronization
comm.SymbolSynchronizer*
comm.PreambleDetector*
comm.GMSKTimingSynchronizer*
comm.MSKTimingSynchronizer*
Synchronization Utilities
comm.DiscreteTimeVCO*
Equalization
comm.MLSEEqualizer*
MIMO
comm.LTEMIMOChannel*
comm.MIMOChannel*
comm.OSTBCCombiner*
comm.OSTBCEncoder*
comm.SphereDecoder*
Channel Modeling and RF Impairments

comm.AGC*
comm.AWGNChannel*
comm.BinarySymmetricChannel*
comm.IQImbalanceCompensator*
comm.LTEMIMOChannel*
comm.MemorylessNonlinearity*
comm.MIMOChannel*
comm.PhaseFrequencyOffset*
comm.PhaseNoise*
comm.RayleighChannel*
comm.RicianChannel*
comm.ThermalNoise*
comm.PSKCoarseFrequencyEstimator*
comm.QAMCoarseFrequencyEstimator*
doppler*
iqcoef2imbal
iqimbal
iqimbal2coef
Measurements and Analysis
comm.ACPR*
comm.CCDF*
comm.ErrorRate*
comm.EVM*
comm.MER*

Complex Numbers in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

complex

conj
cplxpair
imag
isnumeric
isreal
isscalar
real
unwrap*

Computer Vision System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Name
Feature Detection, Extraction, and Matching
BRISKPoints*
cornerPoints*
detectBRISKFeatures*
detectFASTFeatures*
detectHarrisFeatures*
detectMinEigenFeatures*
detectMSERFeatures*
detectSURFFeatures*
extractFeatures*
extractHOGFeatures*
extractLBPFeatures*
matchFeatures*
MSERRegions*

Name
SURFPoints*
Image Registration and Geometric Transformations
estimateGeometricTransform*
vision.GeometricShearer*
Object Detection and Recognition
ocr*
ocrText*
vision.PeopleDetector*
vision.CascadeObjectDetector*
Tracking and Motion Estimation
assignDetectionsToTracks
opticalFlowFarneback*
opticalFlowHS*
opticalFlowLKDoG*
opticalFlowLK*
vision.ForegroundDetector*
vision.HistogramBasedTracker*
vision.KalmanFilter*
vision.PointTracker*
vision.TemplateMatcher*
Camera Calibration and Stereo Vision
bboxOverlapRatio*
bbox2points
disparity*
cameraPoseToExtrinsics
cameraMatrix*
cameraPose*
cameraParameters*

Name
detectCheckerboardPoints*
epipolarline
estimateEssentialMatrix*
estimateFundamentalMatrix*
estimateUncalibratedRectification
estimateWorldCameraPose*
extrinsics*
extrinsicsToCameraPose
generateCheckerboardPoints*
isEpipoleInImage
lineToBorderPoints
reconstructScene*
rectifyStereoImages*
relativeCameraPose*
rotationMatrixToVector
rotationVectorToMatrix
selectStrongestBbox*
stereoAnaglyph
stereoParameters*
triangulate*
undistortImage*
Statistics
vision.Autocorrelator*
vision.BlobAnalysis*
vision.Crosscorrelator*
vision.LocalMaximaFinder*
vision.Maximum*
vision.Mean*

Name
vision.Median*
vision.Minimum*
vision.StandardDeviation*
vision.Variance*
Filters, Transforms, and Enhancements
integralImage
vision.Convolver*
vision.DCT*
vision.Deinterlacer*
vision.FFT*
vision.HoughLines*
vision.IDCT*
vision.IFFT*
vision.Pyramid*
Video Loading, Saving, and Streaming
vision.DeployableVideoPlayer*
vision.VideoFileReader*
vision.VideoFileWriter*
Color Space Formatting and Conversions
vision.ChromaResampler*
vision.DemosaicInterpolator*
vision.GammaCorrector*
Graphics
insertMarker*
insertShape*
insertObjectAnnotation*
insertText*
vision.AlphaBlender*

Name
vision.MarkerInserter*
vision.ShapeInserter*

Control Flow in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`break`
`continue`
`end`
`for`
`if, elseif, else`
`parfor*`
`return`
`switch, case, otherwise*`
`while`

Control System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`extendedKalmanFilter*`
`unscentedKalmanFilter*`

Data and File Management in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`computer*`

fclose
feof
fopen*
fprintf*
fread*
frewind
fseek*
ftell*
fwrite*
load*

Data Types in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

cell*
deal
fieldnames*
iscell
isenum
isfield*
ismethod
isobject
isstruct
narginchk
nargoutchk
str2func*
struct*
struct2cell*
structfun*

Desktop Environment in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

ismac*
ispc*
isunix*

Discrete Math in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

factor*
gcd
isprime*
lcm
nchoosek*
primes*

DSP System Toolbox

C code generation for the following functions and System objects requires the DSP System Toolbox license. Many DSP System Toolbox functions require constant inputs for code generation. See “Define Constant Input Parameters Using the App” on page 17-27 and “Specify Constant Inputs at the Command Line” on page 20-53.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Name
Estimation
dsp.BurgAREstimator*
dsp.BurgSpectrumEstimator*
dsp.CepstralToLPC*

Name
dsp.CrossSpectrumEstimator*
dsp.LevinsonSolver*
dsp.LPCToAutocorrelation*
dsp.LPCToCepstral*
dsp.LPCToLSF*
dsp.LPCToLSP*
dsp.LPCToRC*
dsp.LSFToLPC*
dsp.LSPToLPC*
dsp.RCToAutocorrelation*
dsp.RCToLPC*
dsp.SpectrumEstimator*
dsp.TransferFunctionEstimator*
Filters
`ca2tf*`
`cl2tf*`
dsp.AdaptiveLatticeFilter*
dsp.AffineProjectionFilter*
dsp.AllpassFilter*
dsp.AllpoleFilter*
dsp.BiquadFilter*
dsp.Channelizer*
dsp.ChannelSynthesizer*
dsp.CICCompensationDecimator*
dsp.CICCompensationInterpolator*
dsp.CICDecimator*
dsp.CICInterpolator*
dsp.Differentiator*

Name
dsp.FarrowRateConverter*
dsp.FastTransversalFilter*
dsp.FilterCascade*
dsp.FilteredXLMSFilter*
dsp.FIRDecimator*
dsp.FIRFilter*
dsp.FIRHalfbandDecimator*
dsp.FIRHalfbandInterpolator*
dsp.FIRInterpolator*
dsp.FIRRateConverter*
dsp.FrequencyDomainAdaptiveFilter*
dsp.HampelFilter*
dsp.HighpassFilter*
dsp.IIRFilter*
dsp.IIRHalfbandDecimator*
dsp.IIRHalfbandInterpolator*
dsp.KalmanFilter*
dsp.LMSFilter*
dsp.LowpassFilter*
dsp.MedianFilter*
dsp.RLSFilter*
dsp.SampleRateConverter*
dsp.SubbandAnalysisFilter*
dsp.SubbandSynthesisFilter*
dsp.VariableBandwidthFIRFilter*
dsp.VariableBandwidthIIRFilter*
firceqrip*
fireqint*

Name
firgr*
firhalfband*
firlpnorm*
firminphase*
firnyquist*
firpr2chfb*
ifir*
iircomb*
iirgrpdelay*
iirlpnorm*
iirlpnormc*
iirnotch*
iirpeak*
tf2ca*
tf2cl*
Filter Design
designMultirateFIR*
Math Operations
dsp.ArrayVectorAdder*
dsp.ArrayVectorDivider*
dsp.ArrayVectorMultiplier*
dsp.ArrayVectorSubtractor*
dsp.CumulativeProduct *
dsp.CumulativeSum*
dsp.LDLFactor*
dsp.LevinsonSolver*
dsp.LowerTriangularSolver*
dsp.LUFactor*

Name
dsp.Normalizer*
dsp.UpperTriangularSolver*
Quantizers
dsp.ScalarQuantizerDecoder*
dsp.ScalarQuantizerEncoder*
dsp.VectorQuantizerDecoder*
dsp.VectorQuantizerEncoder*
Scopes
dsp.ArrayPlot*
dsp.SpectrumAnalyzer*
dsp.TimeScope*
Signal Management
dsp.AsyncBuffer*
dsp.Counter*
dsp.DelayLine*
Signal Operations
dsp.Convolver*
dsp.DCBlocker*
dsp.Delay*
dsp.DigitalDownConverter*
dsp.DigitalUpConverter*
dsp.Interpolator*
dsp.NCO*
dsp.PeakFinder*
dsp.PhaseExtractor*
dsp.PhaseUnwrapper*
dsp.VariableFractionalDelay*
dsp.VariableIntegerDelay*

Name
dsp.Window*
dsp.ZeroCrossingDetector*
Sinks
audioDeviceWriter*
dsp.AudioFileWriter*
dsp.BinaryFileWriter*
dsp.UDPSender*
Sources
dsp.AudioFileReader*
dsp.BinaryFileReader*
dsp.SignalSource*
dsp.SineWave*
dsp.UDPReceiver*
Statistics
dsp.Autocorrelator*
dsp.Crosscorrelator*
dsp.Histogram*
dsp.Maximum*
dsp.Mean*
dsp.Median*
dsp.MedianFilter*
dsp.Minimum*
dsp.MovingAverage*
dsp.MovingMaximum*
dsp.MovingMinimum*
dsp.MovingRMS*
dsp.MovingStandardDeviation*
dsp.MovingVariance*

Name
dsp.PeakToPeak*
dsp.PeakToRMS*
dsp.RMS*
dsp.StandardDeviation*
dsp.StateLevels*
dsp.Variance*
Transforms
dsp.AnalyticSignal*
dsp.DCT*
dsp.FFT*
dsp.IDCT*
dsp.IFFT*

Error Handling in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

assert*
error*

Exponents in MATLAB

exp
expm
expm1
factorial
log*
log2
log10

log1p
nextpow2
nthroot
reallog
realpow
realsqrt
sqrt*

Filtering and Convolution in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

conv*
conv2
convn
deconv*
detrend*
filter*
filter2

Fixed-Point Designer

The following general limitations apply to the use of Fixed-Point Designer functions in generated code, with `fiaccel`:

- `fipref` and `quantizer` objects are not supported.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numerictype` of a given `fi` variable after that variable has been created.
- The `boolean` value of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.

- You can use parallel for (`parfor`) loops in code compiled with `fiaccel`, but those loops are treated like regular `for` loops.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.
- The general limitations of C/C++ code generated from MATLAB apply. For more information, see “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-20.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

abs
accumneg
accumpos
add*
all
any
atan2
bitand*
bitandreduce
bitcmp
bitconcat
bitget
bitor*
bitorreduce
bitreplicate
bitrol
bitror
bitset
bitshift
bitsliceget

bitsll*
bitsra*
bitsrl*
bitxor*
bitxorreduce
ceil
complex
conj
conv*
convergent
cordicabs*
cordicangle*
cordicatan2*
cordiccart2pol*
cordiccexp*
cordiccos*
cordicpol2cart*
cordicrotate*
cordicsin*
cordicsincos*
cordicsqrt*
cos
ctranspose
diag*
divide*
double*
end
eps*

eq*
fi*
filter*
fimath*
fix
fixed.Quantizer
flip*
fliplr
flipud
floor
for
ge*
get*
getlsb
getmsb
gt*
horzcat
imag
int8, int16, int32, int64
ipermute
iscolumn
isempty
isequal
isfi*
isfimath
isfimathlocal
isfinite
isinf

isnan
isnumeric
isnumerictype
isreal
isrow
isscalar
issigned
isvector
le*
length
logical
lowerbound
lsb*
lt*
max
mean
median
min
minus*
mpower*
mpy*
mrdivide
mtimes*
ndims
ne*
nearest
numberofelements*
numel

numerictype*
permute*
plus*
pow2
power*
qr
quantize
range
rdivide
real
realmax
realmin
reinterpretcast
removefimath
repmat*
rescale
reshape
rot90*
round
setfimath
sfi*
shiftdim*
sign
sin
single*
size
sort*
sqrt*

squeeze
storedInteger
storedIntegerToDouble
sub*
subsasgn
subsref
sum*
times*
transpose
tril*
triu*
ufi*
uint8, uint16, uint32, uint64
uminus
uplus
upperbound
vertcat

Histograms in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

hist*
histc*
histcounts*

Image Acquisition Toolbox

If you install Image Acquisition Toolbox software, you can generate C and C++ code for the VideoDevice System object™. See imaq.VideoDevice and "Code Generation with VideoDevice System Object" (Image Acquisition Toolbox).

Image Processing in MATLAB

Function	Remarks and Limitations
`im2double`	—
`rgb2gray`	—

Image Processing Toolbox

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

Image Processing Toolbox provides three types of code generation support:

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

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`adaptthresh*`
affine2d*

boundarymask*
bwareaopen*
bwboundaries*
bwconncomp*
bwdist*
bweuler*
bwlabel*
bwlookup*
bwmorph*
bwpack*
bwperim*
bwselect*
bwtraceboundary*
bwunpack*
conndef*
demosaic*
edge*
fitgeotrans*
fspecial*
getrangefromclass*
grayconnected*
histeq*
hough*
houghlines*
houghpeaks*
im2int16*
im2uint8*
im2uint16*

im2single*
im2double*
imabsdiff*
imadjust*
imbinarize*
imbothat*
imboxfilt*
imclearborder*
imclose*
imcomplement*
imcrop*
imdilate*
imerode*
imextendedmax*
imextendedmin*
imfill*
imfilter*
imfindcircles*
imgaborfilt*
imgaussfilt*
imgradient3*
imgradientxyz*
imhist*
imhmax*
imhmin*
imlincomb*
immse*
imopen*

imoverlay*
impyramid*
imquantize*
imread*
imreconstruct*
imref2d*
imref3d*
imregionalmax*
imregionalmin*
imresize*
imrotate*
imtophat*
imtranslate*
imwarp*
integralBoxFilter*
intlut*
iptcheckconn*
iptcheckmap*
lab2rgb*
label2idx*
label2rgb*
mean2*
medfilt2*
multithresh*
offsetstrel*
ordfilt2*
otsuthresh*
padarray*
projective2d*

psnr*
regionprops*
rgb2gray*
rgb2lab*
rgb2ycbcr*
strel*
stretchlim*
superpixels*
watershed*
ycbcr2rgb*

Input and Output Arguments in MATLAB

Function	Remarks and Limitations
nargin*	—
nargout*	• For a function with no output arguments, returns 1 if called without a terminating semicolon. **Note:** This behavior also affects extrinsic calls with no terminating semicolon. nargout is 1 for the called function in MATLAB.

Interpolation and Computational Geometry in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

cart2pol
cart2sph
inpolygon*
interp1*

interp1q*
interp2*
interp3*
interpn*
meshgrid
mkpp*
pchip*
pol2cart
polyarea
ppval*
rectint
sph2cart
spline*
unmkpp*

Linear Algebra in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

bandwidth
cholupdate
isbanded
isdiag
ishermitian
istril
istriu
issymmetric
linsolve*
lsqnonneg*

`null*`
`orth*`
`rsf2csf`
`schur*`
`sqrtm`

Logical and Bit-Wise Operations in MATLAB

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

MATLAB Compiler

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`isdeployed*`
`ismcc*`

Matrices and Arrays in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

abs
all*
angle
any*
blkdiag
bsxfun*
cat*
circshift
colon*
compan
cond
cov*
cross*
cumprod*
cumsum*
det
diag*
diff*
dot
eig*
eye*
false*
find*
flip*
flipdim*

fliplr*
flipud*
full
hadamard*
hankel
hilb
ind2sub*
inv*
invhilb
ipermute*
iscolumn
isempty
isequal
isequaln
isfinite
isfloat
isinf
isinteger
islogical
ismatrix
isnan
isrow
issparse
isvector
kron
length
linspace
logspace

lu
magic*
max*
min*
ndgrid
ndims
nnz
nonzeros
norm
normest
numel
ones*
pascal
permute*
pinv
planerot*
prod*
qr
rand*
randi*
randn*
randperm
rank
rcond
repelem*
repmat*
reshape*
rng*

rosser
rot90*
shiftdim*
sign
size
sort*
sortrows*
squeeze*
sub2ind*
subspace
sum*
toeplitz
trace
tril*
triu*
true*
vander
wilkinson*
zeros*

Neural Network Toolbox

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

Numerical Integration and Differentiation in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

cumtrapz
del2
diff
gradient
ode23*
ode45*
odeget*
odeset*
quad2d*
quadgk
trapz*

Optimization Functions in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

fminbnd*
fminsearch*
fzero*
lsqnonneg*
optimget*
optimset*

Optimization Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

fminbnd*
fminsearch*

fzero*
lsqnonneg*
optimget*
optimset*

Phased Array System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Antenna and Microphone Elements
aperture2gain*
azel2phithetapat*
azel2uvpat*
circpol2pol*
gain2aperture*
phased.CosineAntennaElement*
phased.CrossedDipoleAntennaElement*
phased.CustomAntennaElement*
phased.CustomMicrophoneElement*
phased.IsotropicAntennaElement*
phased.IsotropicHydrophone*
phased.IsotropicProjector*
phased.OmnidirectionalMicrophoneElement*
phased.ShortDipoleAntennaElement*
phitheta2azelpat*
phitheta2uvpat*
pol2circpol*
polellip*

polloss*
polratio*
polsignature*
stokes*
uv2azelpat*
uv2phithetapat*
Array Geometries and Analysis
az2broadside*
broadside2az*
pilotcalib*
phased.ArrayGain*
phased.ArrayResponse*
phased.ConformalArray*
phased.ElementDelay*
phased.PartitionedArray*
phased.ReplicatedSubarray*
phased.SteeringVector*
phased.UCA*
phased.ULA*
phased.URA*
taylortaperc*
Signal Radiation and Collection
phased.Collector*
phased.Radiator*
phased.WidebandCollector*
phased.WidebandRadiator*
sensorsig*
Transmitters and Receivers
delayseq*

noisepow*
phased.ReceiverPreamp*
phased.Transmitter*
systemp*
Waveform Design and Analysis
ambgfun*
pambgfun*
phased.FMCWWaveform*
phased.LinearFMWaveform*
phased.MFSKWaveform*
phased.PhaseCodedWaveform*
phased.RectangularWaveform*
phased.SteppedFMWaveform*
range2bw*
range2time*
time2range*
unigrid*
Beamforming
cbfweights*
lcmvweights*
mvdrweights*
phased.FrostBeamformer*
phased.GSCBeamformer*
phased.LCMVBeamformer*
phased.MVDRBeamformer*
phased.PhaseShiftBeamformer*
phased.SteeringVector*
phased.SubbandMVDRBeamformer*
phased.SubbandPhaseShiftBeamformer*

phased.TimeDelayBeamformer*
phased.TimeDelayLCMVBeamformer*
sensorcov*
steervec*
Direction of Arrival (DOA) Estimation
aictest*
espritdoa*
gccphat*
mdltest*
musicdoa*
phased.BeamscanEstimator*
phased.BeamscanEstimator2D*
phased.BeamspaceESPRITEstimator*
phased.ESPRITEstimator*
phased.GCCEstimator*
phased.MUSICEstimator*
phased.MUSICEstimator2D*
phased.MVDREstimator*
phased.MVDREstimator2D*
phased.RootMUSICEstimator*
phased.RootWSFEstimator*
phased.SumDifferenceMonopulseTracker*
phased.SumDifferenceMonopulseTracker2D*
rootmusicdoa*
spsmooth*
Space-Time Adaptive Processing (STAP)
dopsteeringvec*
phased.ADPCACanceller*
phased.AngleDopplerResponse*

phased.DPCACanceller*
phased.STAPSMIBeamformer*
val2ind*
Detection, Range, and Doppler Estimation
albersheim*
beat2range*
bw2range*
dechirp*
npwgnthresh*
phased.CFARDetector*
phased.CFARDetector2D*
phased.DopplerEstimator*
phased.MatchedFilter*
phased.RangeDopplerResponse*
phased.RangeEstimator*
phased.RangeResponse*
phased.StretchProcessor*
phased.TimeVaryingGain*
pulsint*
radareqpow*
radareqrng*
radareqsnr*
radarvcd*
range2beat*
rdcoupling*
rocpfa*
rocsnr*
shnidman*
stretchfreq2rng*

Targets, Interference, and Signal Propagation
billingsleyicm*
depressionang*
diagbfweights*
effearthradius*
fspl*
fogpl*
gaspl*
grazingang*
horizonrange*
phased.BackscatterRadarTarget*
phased.BackScatterSonarTarget*
phased.BarrageJammer*
phased.ConstantGammaClutter*
phased.FreeSpace*
phased.IsoSpeedUnderWaterPaths*
phased.LOSChannel*
phased.MultipathChannel*
phased.RadarTarget*
phased.ScatteringMIMOChannel*
phased.TwoRayChannel*
phased.WidebandFreeSpace*
phased.WidebandBackscatterRadarTarget*
phased.WidebandLOSChannel*
phased.WidebandTwoRayChannel*
physconst*
scatteringchanmtx*
surfacegamma*
surfclutterrcs*

rainpl*
waterfill*
Motion Modeling and Coordinate Systems
azel2phitheta*
azel2uv*
azelaxes*
cart2sphvec*
dop2speed*
global2localcoord*
local2globalcoord*
phased.Platform*
phitheta2azel*
phitheta2uv*
radialspeed*
rangeangle*
rotx*
roty*
rotz*
speed2dop*
sph2cartvec*
uv2azel*
uv2phitheta*

Polynomials in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

poly*
polyder*
polyeig*

polyfit*
polyint
polyval
polyvalm
roots*

Programming Utilities in MATLAB

Function	Remarks and Limitations
mfilename	—

Relational Operators in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

eq*
ge
gt
le
lt
ne*

Robotics System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Algorithm Design
robotics.AimingConstraint
robotics.BinaryOccupancyGrid*
robotics.Cartesianbounds

robotics.GeneralizedInverseKinematics*
robotics.InverseKinematics*
robotics.Joint
robotics.JointPositionBounds
robotics.OccupancyGrid*
robotics.OdometryMotionModel*
robotics.OrientationTarget
robotics.ParticleFilter*
robotics.PoseTarget
robotics.PositionTarget
robotics.PRM*
robotics.PurePursuit*
robotics.RigidBody
robotics.RigidBodyTree*
`transformScan`
robotics.VectorFieldHistogram*
Coordinate System Transformations
`angdiff`
`axang2quat`
`axang2rotm`
`axang2tform`
`cart2hom`
`eul2quat`
`eul2rotm`
`eul2tform`
`hom2cart`
`quat2axang`
`quat2eul`
`quat2rotm`

quat2tform
rotm2axang
rotm2eul
rotm2quat
rotm2tform
tform2axang
tform2eul
tform2quat
tform2rotm
tform2trvec
trvec2tform

Rounding and Remainder Functions in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

ceil
fix
floor
mod*
rem*
round*

Set Operations in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

intersect*
ismember*
issorted*
setdiff*

setxor*
union*
unique*

Signal Processing in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

chol
conv
fft*
fft2
fftn*
fftshift
filter
freqspace
ifft*
ifft2*
ifftn*
ifftshift
svd*
zp2tf

Signal Processing Toolbox

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

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

alignsignals
barthannwin*
bartlett*
besselap*
bitrevorder
blackman*
blackmanharris*
bohmanwin*
buttap*
butter*
buttord*
cconv
cfirpm*
cheb1ap*
cheb2ap*
cheb1ord*
cheb2ord*
chebwin*
cheby1*
cheby2*
convmtx
corrmtx
db2pow
dct*
downsample
dpss*
ellip*

ellipap*
ellipord*
envelope*
filtfilt*
finddelay
findpeaks
fir1*
fir2*
fircls*
fircls1*
firls*
firpm*
firpmord*
flattopwin*
freqz*
gausswin*
hamming*
hann*
hilbert
idct*
intfilt*
kaiser
kaiserord
levinson*
maxflat*
nuttallwin*
parzenwin*
peak2peak

peak2rms
pow2db
rcosdesign*
rectwin*
resample*
rms
sgolay
sgolayfilt
sinc
sosfilt
taylorwin*
triang*
tukeywin*
upfirdn*
upsample*
xcorr*
xcorr2
xcov
yulewalk*
zp2tf*

Special Values in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

eps
inf*
intmax
intmin
NaN or nan*

pi
realmax
realmin

Specialized Math in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

airy*
besseli*
besselj*
beta
betainc*
betaincinv*
betaln
ellipke
erf
erfc
erfcinv
erfcx
erfinv
expint
gamma
gammainc*
gammaincinv*
gammaln
psi

Statistics in MATLAB

corrcoef*

cummin
cummax
mean*
median*
mode*
std*
var*

Statistics and Machine Learning Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Descriptive Statistics and Visualization
geomean
harmmean
iqr
kurtosis
mad*
moment*
nancov*
nanmax
nanmean
nanmedian
nanmin
nanstd
nansum
nanvar
prctile*

quantile
skewness
zscore
Probability Distributions
betacdf
betafit
betainv
betalike
betapdf
betarnd*
betastat
binocdf
binoinv
binopdf
binornd*
binostat
cdf
chi2cdf
chi2inv
chi2pdf
chi2rnd*
chi2stat
evcdf
evinv
evpdf
evrnd*
evstat
expcdf

expinv
exppdf
exprnd*
expstat
fcdf
finv
fpdf
frnd*
fstat
gamcdf
gaminv
gampdf
gamrnd*
gamstat
geocdf
geoinv
geopdf
geornd*
geostat
gevcdf
gevinv
gevpdf
gevrnd*
gevstat
gpcdf
gpinv
gppdf
gprnd*

gpstat
hygecdf
hygeinv
hygepdf
hygernd*
hygestat
icdf
logncdf
logninv
lognpdf
lognrnd*
lognstat
mnpdf
nbincdf
nbininv
nbinpdf
nbinrnd*
nbinstat
ncfcdf
ncfinv
ncfpdf
ncfrnd*
ncfstat
nctcdf
nctinv
nctpdf
nctrnd*
nctstat

ncx2cdf
ncx2rnd*
ncx2stat
normcdf
norminv
normpdf
normrnd*
normstat
pdf
pearsrnd*
poisscdf
poissinv
poisspdf
poissrnd*
poisstat
randg
random
randsample*
raylcdf
raylinv
raylpdf
raylrnd*
raylstat
tcdf
tinv
tpdf
trnd*
tstat

unidcdf
unidinv
unidpdf
unidrnd*
unidstat
unifcdf
unifinv
unifpdf
unifrnd*
unifstat
wblcdf
wblinv
wblpdf
wblrnd*
wblstat
Cluster Analysis
kmeans*
Regression
GeneralizedLinearModel* or CompactGeneralizedLinearModel*
glmval*
LinearModel* or CompactLinearModel*
predict* method of GeneralizedLinearModel and CompactGeneralizedLinearModel
predict* method of LinearModel and CompactLinearModel
predict* method of RegressionTree or CompactRegressionTree
random* method of GeneralizedLinearModel and CompactGeneralizedLinearModel
random* method of LinearModel and CompactLinearModel
RegressionTree* or CompactRegressionTree*
Classification
loadCompactModel*

ClassificationECOC* or CompactClassificationECOC*
ClassificationEnsemble*, ClassificationBaggedEnsemble*, or CompactClassificationEnsemble*
ClassificationLinear*
ClassificationSVM* or CompactClassificationSVM*
ClassificationTree* or CompactClassificationTree*
predict* method of `ClassificationECOC` and `CompactClassificationECOC`
predict* method of `ClassificationEnsemble`, `ClassificationBaggedEnsemble`, and `CompactClassificationEnsemble`
predict* method of `ClassificationLinear`
predict* method of `ClassificationSVM` and `CompactClassificationSVM`
predict* method of `ClassificationTree` and `CompactClassificationTree`
Dimensionality Reduction
`pca*`

System Identification Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`extendedKalmanFilter*`
`recursiveAR*`
`recursiveARMA*`
`recursiveARMAX*`
`recursiveARX*`
`recursiveBJ*`
`recursiveLS*`
`recursiveOE*`
`unscentedKalmanFilter*`

System object Methods

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

getNumInputs*
getNumOutputs*
isLocked*
release*
reset*
step*

Trigonometry in MATLAB

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

acos*
acosd
acosh*
acot
acotd
acoth
acsc
acscd
acsch
asec
asecd
asech
asin*
asind
asinh

atan
atan2
atan2d
atand
atanh*
cos
cosd
cosh
cot
cotd*
coth
csc
cscd*
csch
deg2rad
hypot
rad2deg
sec
secd*
sech
sin
sind
sinh
tan
tand*
tanh

Wavelet Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Signal Analysis
appcoef*
detcoef*
dwt*
dyadup*
idwt*
imodwpt*
imodwt*
modwpt*
modwptdetails*
modwt*
modwtmra*
wavedec*
waverec*
wextend*
Image Analysis
appcoef2*
detcoef2*
dwt2*
idwt2*
wavedec2*
waverec2*
Denoising

ddencmp*
thselect*
wden*
wdencmp*
wnoisest*
wthcoef*
wthcoef2*
wthresh*
Orthogonal and Biorthogonal Filter Banks
qmf*

WLAN System Toolbox

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

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

WLAN Modeling
wlanHTConfig*
wlanNonHTConfig*
wlanRecoveryConfig*
wlanS1GConfig*
wlanVHTConfig*
Signal Transmission
wlanDMGConfig*
wlanHTData*
wlanHTLTF*
wlanHTSIG*
wlanHTSTF*
wlanLLTF*

wlanLSIG*
wlanLSTF*
wlanNonHTData*
wlanVHTData*
wlanVHTLTF*
wlanVHTSIGA*
wlanVHTSIGB*
wlanVHTSTF*
wlanWaveformGenerator*
Signal Reception
wlanCoarseCFOEstimate*
wlanFormatDetect*
wlanFieldIndices*
wlanFineCFOEstimate*
wlanHTDataRecover*
wlanHTLTFChannelEstimate*
wlanHTLTFDemodulate*
wlanHTSIGRecover*
wlanLLTFChannelEstimate*
wlanLLTFDemodulate*
wlanLSIGRecover*
wlanNonHTDataRecover*
wlanPacketDetect*
wlanSymbolTimingEstimate*
wlanVHTDataRecover*
wlanVHTLTFChannelEstimate*
wlanVHTLTFDemodulate*
wlanVHTSIGARecover*

wlanVHTSIGBRecover*
Propagation Channel
wlanTGacChannel*
wlanTGahChannel*
wlanTGnChannel*

Note: WLAN System Toolbox functionality with the MATLAB Function block is not supported.

4

Defining MATLAB Variables for C/C++ Code Generation

Variables Definition for Code Generation

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

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

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

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

For more information, see "Best Practices for Defining Variables for C/C++ Code Generation" on page 4-3.

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

In this section...

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

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

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

Initializing the new variable to the value of the assigned data sometimes results in redundant copies in the generated code. To avoid redundant copies, you can define variables without initializing their values by using the `coder.nullcopy` construct as described in "Eliminate Redundant Copies of Variables in Generated Code" on page 4-7.

When you define variables, they are local by default; they do not persist between function calls. To make variables persistent, see `persistent`.

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 <= 0`),.

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

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

Defining Fields in a Structure

Consider the following MATLAB code:

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

```
end
% Try to use s
use(s);
...
```

Here, the first part of the `if` statement uses only the field *a*, and the `else` clause uses fields *a* and *b*. This code works in MATLAB, but generates a compilation error during C/C++ code generation because it detects a structure type mismatch. To prevent this error, do not add fields to a structure after you perform certain operations on the structure. For more information, see “Structure Definition for Code Generation” on page 7-2.

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

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

Use Caution When Reassigning Variables

In general, you should adhere to the "one variable/one type" rule for C/C++ code generation; that is, each variable must have a specific class, size and complexity. Generally, if you reassign variable properties after the initial assignment, you get a compilation error during code generation, but there are exceptions, as described in “Reassignment of Variable Properties” on page 4-9.

Use Type Cast Operators in Variable Definitions

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

```
...
x = 15; % x is of type double by default.
```

```
y = uint8(x); % y has the value of x, but cast to uint8.
...
```

Define Matrices Before Assigning Indexed Variables

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

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

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

For more information about indexing matrices, see “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 6-30.

Eliminate Redundant Copies of Variables in Generated Code

In this section...

When Redundant Copies Occur

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

Note, however, that variable assignments not only copy the properties of the assigned data to the new variable, but also initialize the new variable to the assigned value. This forced initialization sometimes results in redundant copies in C/C++ code. To eliminate redundant copies, define uninitialized variables by using the `coder.nullcopy` function, as described in “How to Eliminate Redundant Copies by Defining Uninitialized Variables” on page 4-7.

How to Eliminate Redundant Copies by Defining Uninitialized Variables

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

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

   ### What happens if you access uninitialized data?

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

Defining Uninitialized Variables

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

```
function X = fcn %#codegen

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

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

```
function X = fcn2 %#codegen

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

Reassignment of Variable Properties

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

Dynamically sized variables

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

Variables reused in the code for different purposes

You can reassign the type (class, size, and complexity) of a variable after the initial assignment if each occurrence of the variable can have only one type. In this case, the variable is renamed in the generated code to create multiple independent variables. For more information, see “Reuse the Same Variable with Different Properties” on page 4-10.

Reuse the Same Variable with Different Properties

In this section...

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 the code generator 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 "MATLAB Code Variables in a Report" on page 21-17).

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

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

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

When You Cannot Reuse Variables

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

For example, the following `example2` function assigns a fixed-point value to *x* in the `if` statement and reuses *x* to store a matrix of doubles in the `else` clause. It then uses *x* after the `if-else` statement. This function generates a compilation error because after the `if-else` statement, variable *x* can have different properties depending on which `if-else` clause executes.

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

Variable Reuse in an if Statement

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

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

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

2 Compile `example1`.

For example, to generate a MEX function, enter:

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

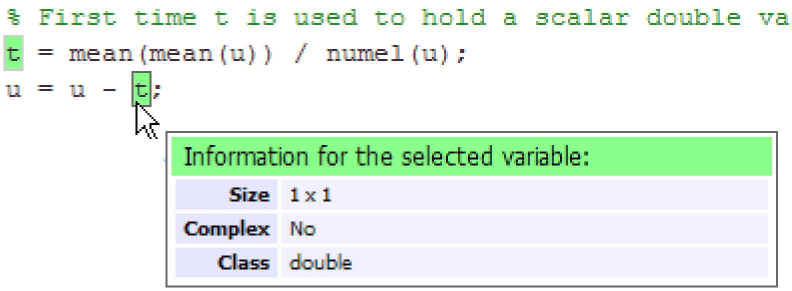
Note: `codegen` requires a MATLAB Coder license.

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

3 Open the code generation report.

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

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



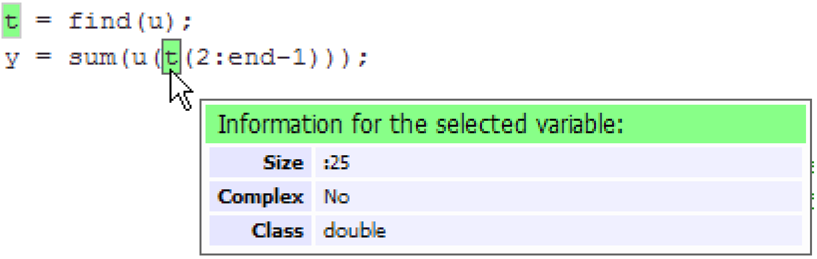
```
% First time t is used to hold a scalar double value.
t = mean(mean(u)) / numel(u);
u = u - t;
```

Information for the selected variable:

Size	1 x 1
Complex	No
Class	double

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

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



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

Information for the selected variable:

Size	:25
Complex	No
Class	double

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

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

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

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

8. In the list of variables, place your pointer over *t>2*

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

Limitations of Variable Reuse

The following variables cannot be renamed in generated code:

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

Avoid Overflows in for-Loops

When memory integrity checks are enabled, if the code generator 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

• 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
	Rewrite the loop so that the end value is not equal to the maximum value of the integer type. For example, replace:
	`N=intmax('int16')`
	`for k=N-10:N`
	with:
	`for k=1:10`

• 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
	Rewrite the loop so that the end value is not equal to the minimum value of the integer type. For example, replace:
	`N=intmin('int32')`
	`for k=N+10:-1:N`
	with:
	`for k=10:-1:1`

• 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
The loop covers the full range of the integer type.
	Rewrite the loop casting the type of the loop counter start, step, and end values to a bigger integer or to double For example, rewrite:
	`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`

Loop conditions causing the error	Workaround
	`end`
• 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 **Note:** The software error checking is conservative. It may incorrectly report a loop as being potentially infinite.	Rewrite the loop so that the loop variable on the last loop iteration is equal to the end value.

Supported Variable Types

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

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

Defining Data for Code Generation

Data Definition for Code Generation

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

Data	What's Different	More Information
Arrays	Maximum number of elements is restricted	"Array Size Restrictions for Code Generation" on page 5-10
Complex numbers	• Complexity of variables must be set at time of assignment and before first use • Expressions containing a complex number or variable evaluate to a complex result, even if the result is zero **Note:** Because MATLAB does not support complex integer arithmetic, you cannot generate code for functions that use complex integer arithmetic	"Code Generation for Complex Data" on page 5-4
Characters	Restricted to 8 bits of precision	"Code Generation for Character Arrays" on page 5-9
Enumerated data	• Supports integer-based enumerated types only • Restricted use in `switch` statements and `for`-loops	"Enumerations"
Function handles	• Using the same bound variable to reference different function	"Function Handles"

Data	What's Different	More Information
	handles can cause a compile-time error. - Cannot pass function handles to or from primary or extrinsic functions - Cannot view function handles from the debugger	

Code Generation for Complex Data

In this section...

Restrictions When Defining Complex Variables

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

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

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

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

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

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

Code Generation for Complex Data with Zero-Valued Imaginary Parts

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

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

Functions That Sort Complex Values by Absolute Value

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

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

  1. Write this function:

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

  2. Call `mysort` in MATLAB.

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

     ans =

         -2    -1     0     1     2
     ```

  3. Generate a MEX function for complex inputs.

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

  4. Call the MEX Function with real inputs.

     ```
     mysort_mex(A)

     ans =
     ```

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

You generated the MEX function for complex inputs, therefore, it treats the real inputs as complex numbers with zero-valued imaginary parts. It sorts the numbers by the absolute values of the complex numbers. Because the imaginary parts are zero, the MEX function returns the results to the MATLAB workspace as real numbers. See "Inputs and Outputs for MEX Functions Generated for Complex Arguments" on page 5-7.

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

1. Write this function:

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

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

2. Call `myfun` in MATLAB.

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

   ans =

      12.5777
       2.0000
      -3.5777
   ```

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

3. Generate a MEX function for complex inputs.

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

4. Call the MEX function.

   ```
   myfun_mex(A)

   ans =

      12.5777
      -3.5777
       2.0000
   ```

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

Inputs and Outputs for MEX Functions Generated for Complex Arguments

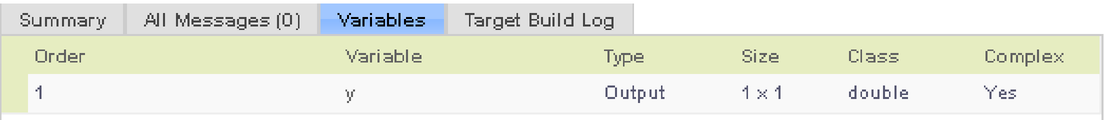
For MEX functions created by MATLAB Coder:

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

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

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

  ```
  codegen foo -report
  ```

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

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

  ```
  z = foo_mex

  ans =

       1
  ```

Results of Expressions That Have Complex Operands

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

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

Suppose that at run time, x has the value 2 + 3i and y has the value 2 - 3i. In MATLAB, this code produces the real result z = 4. During code generation, the types for x and y are known, but their values are not known. Because either or both operands in this expression are complex, z is defined as a complex variable requiring storage for a real and an imaginary part. z equals the complex result 4 + 0i in generated code, not 4, as in MATLAB code.

Exceptions to this behavior are:

- When the imaginary parts of complex results are zero, MEX functions return the results to the MATLAB workspace as real values. See “Inputs and Outputs for MEX Functions Generated for Complex Arguments” on page 5-7.
- When the imaginary part of the argument is zero, complex arguments to extrinsic functions are real .

  ```
  function y = foo()
      coder.extrinsic('sqrt')
      x = 1 + 0i;   % x is complex
      y = sqrt(x);  % x is real, y is real
  end
  ```
- Functions that take complex arguments but produce real results return real values.

  ```
  y = real(x); % y is the real part of the complex number x.
  y = imag(x); % y is the real-valued imaginary part of x.
  y = isreal(x); % y is false (0) for a complex number x.
  ```
- Functions that take real arguments but produce complex results return complex values.

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

Code Generation for Character Arrays

The code generator translates the 16-bit Unicode encoding of a character in MATLAB to an 8-bit encoding that the locale setting determines. The code generator does not support characters that require more than 1 byte in MATLAB. This restriction applies to character arrays that are passed between MATLAB and the code generator. For example, the restriction applies to entry-point function inputs and to outputs from extrinsic calls. For code generation, some MATLAB functions accept only 7-bit ASCII characters. See “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 3-2.

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

More About

- “Locale Settings for MATLAB Process” (MATLAB)

Array Size Restrictions for Code Generation

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

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

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

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

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

These restrictions apply even on a 64-bit platform.

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

See Also

- "Variable-Size Data"
- coder.HardwareImplementation

Code Generation for Constants in Structures and Arrays

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

Fields or elements are assigned inside control constructs

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

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

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

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

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

Constants are assigned to array elements using non-scalar indexing

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

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

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

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

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

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

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

function s = create_structure(x)
s.a = 10;
s.b = x;
```

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

6

Code Generation for Variable-Size Data

Code Generation for Variable-Size Arrays

For code generation, an array dimension is *fixed-size* or *variable-size*. If the code generator can determine the size of the dimension and that the size of the dimension does not change, then the dimension is fixed-size. When all dimensions of an array are fixed-size, the array is a *fixed-size* array. In the following example, Z is a fixed-size array.

```
function Z = myfcn()
Z = zeros(1,4);
end
```

The size of the first dimension is 1 and the size of the second dimension is 4.

If the code generator cannot determine the size of a dimension or the code generator determines that the size changes, then the dimension is variable-size. When at least one of its dimensions is variable-size, an array is a *variable-size* array.

A variable-size dimension is either *bounded* or *unbounded*. A bounded dimension has a fixed upper size. An unbounded dimension does not have a fixed upper size.

In the following example, the second dimension of Z is bounded, variable-size. It has an upper bound of 16.

```
function s = myfcn(n)
if (n > 0)
    Z = zeros(1,4);
else
    Z = zeros(1,16);
end
s = length(Z);
```

In the following example, if the value of n is unknown at compile time, then the second dimension of Z is unbounded.

```
function s = myfcn(n)
Z = rand(1,n);
s = sum(Z);
end
```

You can define variable-size arrays by:

- Using constructors, such as zeros, with a nonconstant dimension

- Assigning multiple, constant sizes to the same variable before using it
- Declaring all instances of a variable to be variable-size by using `coder.varsize`

For more information, see “Define Variable-Size Data for Code Generation” on page 6-9.

You can control whether variable-size arrays are allowed for code generation. See “Enabling and Disabling Support for Variable-Size Arrays” on page 6-3.

Memory Allocation for Variable-Size Arrays

For fixed-size arrays and variable-size arrays whose size is less than a threshold, the code generator allocates memory statically on the stack. For unbounded, variable-size arrays and variable-size arrays whose size is greater than or equal to a threshold, the code generator allocates memory dynamically on the heap.

You can control whether dynamic memory allocation is allowed or when it is used for code generation. See “Control Memory Allocation for Variable-Size Arrays” on page 6-5.

The code generator represents dynamically allocated data as a structure type called `emxArray`. The code generator generates utility functions that create and interact with emxArrays. If you use Embedded Coder, you can customize the generated identifiers for the `emxArray` types and utility functions. See “Identifier Format Control” (Embedded Coder).

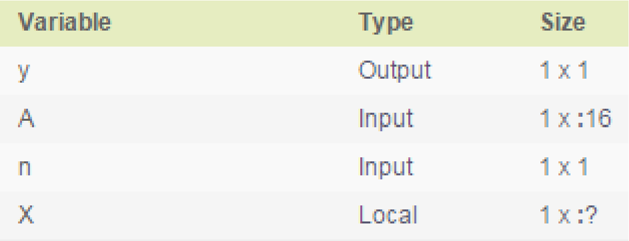
Enabling and Disabling Support for Variable-Size Arrays

By default, support for variable-size arrays is enabled. To modify this support:

- In a code configuration object, set the `EnableVariableSizing` parameter to `true` or `false`.
- In the MATLAB Coder app, in the **Memory** settings, select or clear the **Enable variable-sizing** check box.

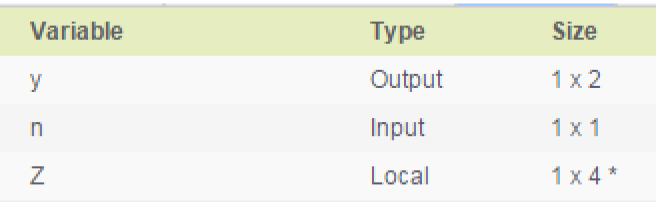
Variable-Size Arrays in a Code Generation Report

You can tell whether an array is fixed-size or variable-size by looking at the **Size** column of the **Variables** tab in a code generation report.



Variable	Type	Size
y	Output	1 x 1
A	Input	1 x :16
n	Input	1 x 1
X	Local	1 x :?

A colon (:) indicates that a dimension is variable-size. A question mark (?) indicates that the size is unbounded. For example, a size of 1-by-:? indicates that the size of the first dimension is fixed-size 1 and the size of the second dimension is unbounded, variable-size. An asterisk (*) indicates that the code generator produced a variable-size array, but the size of the array does not change during execution.



Variable	Type	Size
y	Output	1 x 2
n	Input	1 x 1
Z	Local	1 x 4 *

More About

- "Control Memory Allocation for Variable-Size Arrays" on page 6-5
- "Specify Upper Bounds for Variable-Size Arrays" on page 6-7
- "Define Variable-Size Data for Code Generation" on page 6-9

Control Memory Allocation for Variable-Size Arrays

Dynamic memory allocation allocates memory on the heap as needed at run time, instead of allocating memory statically on the stack. Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

Dynamic memory allocation and the freeing of this memory can result in slower execution of the generated code. To control the use of dynamic memory allocation for variable-size arrays in a MATLAB Function block, you can:

- Provide upper bounds for variable-size arrays.
- Disable dynamic memory allocation.
- Configure the code generator to use dynamic memory allocation for arrays bigger than a threshold.

Provide Upper Bounds for Variable-Size Arrays

For an unbounded variable-size array, the code generator allocates memory dynamically on the heap. For a variable-size array with upper bound, whose size, in bytes, is less than the dynamic memory allocation threshold, the code generator allocates memory statically on the stack. To prevent dynamic memory allocation:

1. Specify upper bounds for a variable-size array. See “Specify Upper Bounds for Variable-Size Arrays” on page 6-7.
2. Make sure that the size of the array, in bytes, is less than the dynamic memory allocation threshold. See “Configure Code Generator to Use Dynamic Memory Allocation for Arrays Bigger Than a Threshold” on page 6-6.

Disable Dynamic Memory Allocation

By default, dynamic memory allocation is enabled. To disable it:

- In a configuration object for code generation, set the `DynamicMemoryAllocation` parameter to `'Off'`.
- In the MATLAB Coder app, in the **Memory** settings, set **Dynamic memory allocation** to `Never`.

If you disable dynamic memory allocation, you must provide upper bounds for variable-size arrays.

Configure Code Generator to Use Dynamic Memory Allocation for Arrays Bigger Than a Threshold

Instead of disabling dynamic memory allocation for all variable-size arrays, you can specify for which size arrays the code generator uses dynamic memory allocation.

Use the dynamic memory allocation threshold to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, static memory allocation can speed up generated code. However, static memory allocation can lead to unused storage space. You can decide that the unused storage space is not a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, when you use dynamic memory allocation, you can significantly reduce storage requirements.

The default dynamic memory allocation threshold is 64 kilobytes. To change the threshold:

- In a configuration object for code generation, set the `DynamicMemoryAllocationThreshold`.
- In the MATLAB Coder app, in the **Memory settings**, set **Dynamic memory allocation threshold**.

To instruct the code generator to use dynamic memory allocation for variable-size arrays whose size is greater than or equal to the threshold:

- In the configuration object, set the `DynamicMemoryAllocationThreshold` to `'Threshold'`.
- In the MATLAB Coder app, in the **Memory settings**, set **Dynamic memory allocation threshold** to `For arrays with max size at or above threshold`.

More About

- “Code Generation for Variable-Size Arrays” on page 6-2
- “Configure Build Settings” on page 20-26

Specify Upper Bounds for Variable-Size Arrays

Specify upper bounds for an array when:

- Dynamic memory allocation is disabled.

  If dynamic memory allocation is disabled, you must specify upper bounds for all arrays.
- You do not want the code generator to use dynamic memory allocation for the array.

  Specify upper bounds that result in an array size (in bytes) that is less than the dynamic memory allocation threshold.

Specify Upper Bounds for Variable-Size Inputs

If you generate code by using `codegen`, to specify upper bounds for variable-size inputs, use the `coder.typeof` construct with the `-args` option. 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.

If you generate code by using the MATLAB Coder app, see “Specify Properties of Entry-Point Function Inputs Using the App” on page 17-3 and “Make Dimensions Variable-Size When They Meet Size Threshold” on page 17-5.

Specify Upper Bounds for Local Variables

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

Constrain the Value of Variables That Specify the Dimensions of Variable-Size Arrays

To constrain the value of variables that specify the dimensions of variable-size arrays, use the `assert` function with relational operators. 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. `L` is variable-size with upper bounds of 5 in each dimension. `M` is variable-size with an upper bound of 10 in the first dimension and 5 in the second dimension.

Specify the Upper Bounds for All Instances of a Local Variable

To specify the upper bounds for all instances of a local variable in a function, use the `coder.varsize` 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`:

- The first dimension is fixed at size 1.
- The second dimension can grow to an upper bound of 10.

See Also

`coder.typeof` | `coder.varsize`

More About

- "Code Generation for Variable-Size Arrays" on page 6-2
- "Define Variable-Size Data for Code Generation" on page 6-9

Define Variable-Size Data for Code Generation

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

Method	See
Assign the data from a variable-size matrix constructor such as: • `ones` • `zeros` • `repmat`	"Use a Matrix Constructor with Nonconstant Dimensions" on page 6-9
Assign multiple, constant sizes to the same variable before using (reading) the variable.	"Assign Multiple Sizes to the Same Variable" on page 6-10
Define all instances of a variable to be variable-size.	"Define Variable-Size Data Explicitly by Using coder.varsize" on page 6-10

Use a Matrix Constructor with Nonconstant Dimensions

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

```
function s = var_by_assign(u) %#codegen
y = ones(3,u);
s = numel(y);
```

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 s = var_by_assign(u) %#codegen
assert (u < 20);
y = ones(3,u);
s = numel(y);
```

Assign Multiple Sizes to the Same Variable

Before you use (read) a variable in your code, you can make it variable-size by assigning multiple, constant sizes to it. When the code generator uses static allocation on the stack, 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, the code generator assumes that the dimension is fixed at that size. The assignments can specify different shapes and sizes.

When the code generator uses dynamic memory allocation, it does not check for upper bounds. It assumes that the variable-size data is unbounded.

Inferring Upper Bounds from Multiple Definitions with Different Shapes

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

When the code generator uses static allocation, it infers that `y` is a matrix with three dimensions:

- The first dimension is fixed at size 3
- The second dimension is variable-size with an upper bound of 4
- The third dimension is variable-size with an upper bound of 5

When the code generator uses dynamic allocation, it analyzes the dimensions of `y` differently:

- The first dimension is fixed at size 3.
- The second and third dimensions are unbounded.

Define Variable-Size Data Explicitly by Using coder.varsize

To explicitly define variable-size data, use the function `coder.varsize`. Optionally, you can also specify which dimensions vary along with their upper bounds. For example:

- Define B as a variable-size 2-dimensional array, where each dimension has an upper bound of 64.

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

- Define B as a variable-size array:

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

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

Specify Which Dimensions Vary

You can use the function coder.varsize to specify which dimensions vary. For example, the following statement defines B as an array whose first dimension is fixed at 2, but whose second dimension can grow to a size 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 “Define Variable-Size Matrices with Singleton Dimensions” on page 6-12.

Allow a Variable to Grow After Defining Fixed Dimensions

Function var_by_if defines matrix Y with fixed 2-by-2 dimensions before the 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`, the code generator infers `Y` to be a fixed-size, 2-by-2 matrix. It generates a size mismatch error.

Define Variable-Size Matrices with Singleton Dimensions

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

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

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

  ```
  function Y = dim_singleton(u) %#codegen
  Y = [1 2];
  coder.varsize('Y', [1 10]);
  if (u > 0)
      Y = [Y 3];
  else
      Y = [Y u];
  end
  ```
- You initialize variable-size data with singleton dimensions by using matrix constructor expressions or matrix functions.

  For example, in this function, `X` and `Y` behave like vectors where only their second dimensions are variable-size.

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

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

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

```
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

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

Define Variable-Size Structure Fields

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

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

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

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

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

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

Here are other examples:

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

  In this example, `data` is a scalar variable that contains matrix A. Each element of matrix A contains a variable-size field B.
- `coder.varsize('data(:).A(:).B')`

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

See Also

`coder.typeof` | `coder.varsize`

More About

- “Code Generation for Variable-Size Arrays” on page 6-2
- “Specify Upper Bounds for Variable-Size Arrays” on page 6-7

C Code Interface for Arrays

In this section...

C Code Interface for Statically Allocated Arrays

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

For example, consider the MATLAB function `myuniquetol`.

```
function B = myuniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 100], [0 1]);
B = A(1);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

The statement `coder.varsize('B', [1 100], [0 1])` specifies that `B` is a variable-size array whose first dimension is fixed at 1 and second dimension can vary up to 100 elements. Without this statement, `B` is a dynamically allocated array.

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

In the generated code, the function declaration is:

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

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

The following C main function shows how to call myuniquetol.

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

C Code Interface for Dynamically Allocated Arrays

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

emxArray Structure Definition

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

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

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

The predefined type corresponding to `double` is `real_T`. For more information on the correspondence between built-in data types and predefined types in `rtwtypes.h`, see "How MATLAB Coder Infers C/C++ Data Types" on page 26-9.

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

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

C Code Interface for Structure Fields

Field	Description
`*data`	Pointer to data of type <*baseType*>.
`*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: • `true` – MATLAB deallocates memory automatically • `false` – 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 generator exports a set of utility functions that you can use to create and interact with `emxArrays` in your generated code. To call these functions in your main C function, include the generated header file. For example, when you generate code for function `foo`, include `foo_emxAPI.h` in your main C function. For more information, see the “Write a C Main Function” section in “Using Dynamic Memory Allocation for an "Atoms" Simulation” on page 20-118.

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

Function	Arguments	Description

`emxArray_<baseType> *emxCreateWrapper_<baseType> (...)`	`*data`
	`num_rows`
	`num_cols`
		Creates a new two-dimensional `emxArray`, but does not allocate it on the heap. Instead uses memory provided by the user and sets `canFreeData` to `false` so it does not inadvertently free user memory, such as the stack.

`emxArray_<baseType> *emxCreateWrapperND_<baseType> (...)`	`*data`
	`numDimensions`
	`*size`
		Same as `emxCreateWrapper_<baseType>`, except it creates a new N-dimensional `emxArray`.

`emxArray_<baseType> *emxCreate_<baseType> (...)`	`num_rows` `num_cols`	Creates a new two-dimensional `emxArray` on the heap, initialized to zero. All data elements have the

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

By default, when you generate C/C++ source code, static libraries, dynamic libraries, and executables, MATLAB Coder generates an example C/C++ main function. The example main function is a template that can help you to incorporate generated C/C++ code into your application. If you generate code that uses dynamically allocated data, the example main function includes calls to `emxArray` utility functions that create `emxArrays` required for this data. The example main function also initializes `emxArray` data to zero values. For more information, see “Incorporate Generated Code Using an Example Main Function” on page 24-20.

Diagnose and Fix Variable-Size Data Errors

In this section...

Diagnosing and Fixing Size Mismatch Errors

Check your code for these issues:

Assigning Variable-Size Matrices to Fixed-Size Matrices

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

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

Compiling this function produces this error:

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

There are several ways to fix this error:

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

  ```
  function Y = example_mismatch1_fix1(n) %#codegen
  coder.varsize('A');
  assert(n < 10);
  B = ones(n,n);
  A = magic(3);
  A(1) = mean(A(:));
  if (n == 3)
  ```

```
    A = B;
end
Y = A;
```

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

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

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

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

Empty Matrix Reshaped to Match Variable-Size Specification

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

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

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

Performing Binary Operations on Fixed and Variable-Size Operands

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

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

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

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

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

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

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

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

There are several ways to fix the problem:

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

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

Incompatibilities with MATLAB in Variable-Size Support for Code Generation

In this section...

Incompatibility with MATLAB for Scalar Expansion

Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. If one operand is a scalar and the other is not, scalar expansion applies the scalar to every element of the other operand.

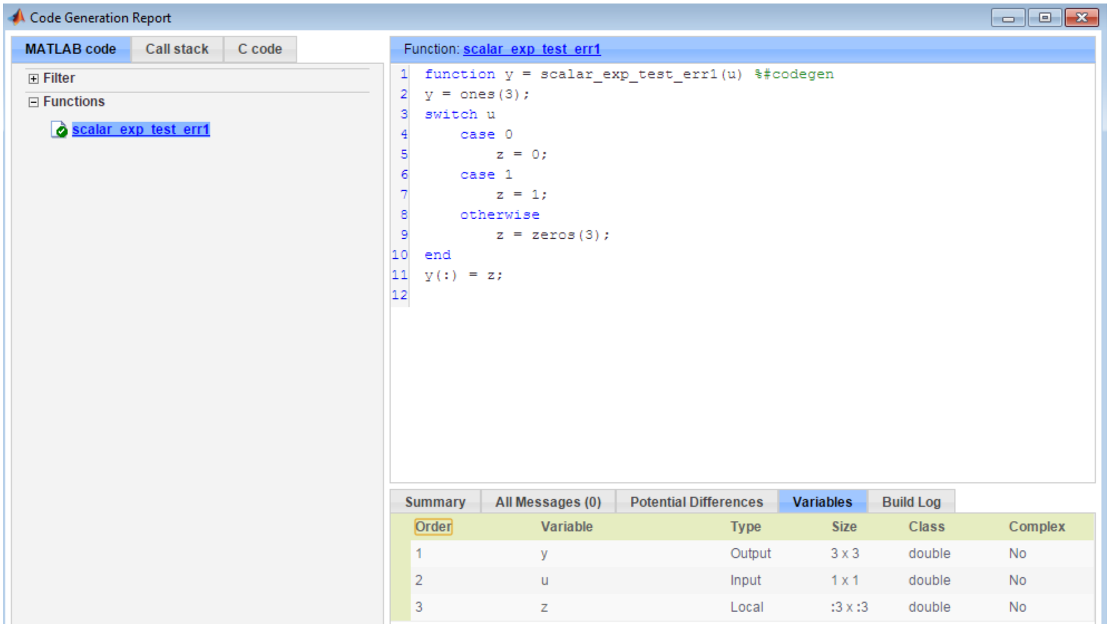
During code generation, scalar expansion rules apply except when operating on two variable-size expressions. In this case, both operands must be the same size. The generated code does not perform scalar expansion even if one of the variable-size expressions turns out to be scalar at run time. Therefore, when run-time error checks are enabled, a run-time error can occur.

Consider this function:

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
```

```
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z;
```

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



Function: scalar_exp_test_err1

```
1  function y = scalar_exp_test_err1(u) %#codegen
2  y = ones(3);
3  switch u
4      case 0
5          z = 0;
6      case 1
7          z = 1;
8      otherwise
9          z = zeros(3);
10 end
11 y(:) = z;
12
```

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

If you run the MEX function with u equal to 0 or 1, the generated code does not perform scalar expansion, even though z is scalar at run time. Therefore, when run-time error checks are enabled, a run-time error can occur.

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

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

To avoid this issue, use indexing to force z to be a scalar value.

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

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

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

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

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

Workarounds

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

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

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

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

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

Incompatibility with MATLAB in Determining Size of Empty Arrays

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

For example, consider the following code:

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

Concatenation requires its operands to match on the size of the dimension that is not being concatenated. In the preceding concatenation, the scalar value has size 1x1 and x has size 0x0. To support this use case, the code generator 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.

For incompatibilities with MATLAB in determining the size of an empty array that results from deleting elements of an array, see “Size of Empty Array That Results from Deleting Elements of an Array” on page 2-10.

Workaround

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

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

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

Incompatibility with MATLAB in Determining Class of Empty Arrays

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

For example, consider the following code:

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

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

Workaround

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

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

```
end
y=class(x);
end
```

Incompatibility with MATLAB in Matrix-Matrix Indexing

In matrix-matrix indexing, you use one matrix to index into another matrix. In MATLAB, the general rule for matrix-matrix indexing is that the size and orientation of the result match the size and orientation of the index matrix. For example, if A and B are matrices, size(A(B)) equals size(B). When A and B are vectors, MATLAB applies a special rule. The special vector-vector indexing rule is that the orientation of the result is the orientation of the data matrix. For example, iA is 1-by-5 and B is 3-by-1, then A(B) is 1-by-3.

The code generator applies the same matrix-matrix indexing rules as MATLAB. If A and B are variable-size matrices, to apply the matrix-matrix indexing rules, the code generator assumes that the size(A(B)) equals size(B). If, at run time, A and B become vectors and have different orientations, then the assumption is incorrect. Therefore, when run-time error checks are enabled, an error can occur.

To avoid this issue, force your data to be a vector by using the colon operator for indexing. For example, suppose that 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. Therefore, the code generator applies the indexing rule for indexing one vector with another vector. The orientation of the result is the orientation of the data vector, C.

Incompatibility with MATLAB in Vector-Vector Indexing

In MATLAB, the special rule for vector-vector indexing is that the orientation of the result is the orientation of the data vector. For example, if A is 1-by-5 and B is 3-by-1, then A(B) is 1-by-3. If, however, the data vector A is a scalar, then the orientation of A(B) is the orientation of the index vector B.

The code generator applies the same vector-vector indexing rules as MATLAB. If A and B are variable-size vectors, to apply the indexing rules, the code generator assumes that the orientation of B matches the orientation of A. At run time, if A is scalar and the orientation of A and B do not match, then the assumption is incorrect. Therefore, when run-time error checks are enabled, a run-time error can occur.

To avoid this issue, make the orientations of the vectors match. Alternatively, index single elements by specifying the row and column. For example, A(row, column).

Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation

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

- Initialization of the following style:

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

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

  For code generation, preallocate M.

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

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

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

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

  ```
  ...
  ```

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

Note: The matrix `M` must be defined before entering the loop.

Incompatibility with MATLAB in Concatenating Variable-Size Matrices

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

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

Suppose that:

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

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

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

  - If the corresponding dimension in the base type is variable size, the dimension has size 0 in the result.
  - If the corresponding dimension in the base type is fixed size, the dimension has that size in the result.

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

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

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

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

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

Variable-Sizing Restrictions for Code Generation of Toolbox Functions

In this section...

Common Restrictions

The following common restrictions apply to multiple toolbox functions, but only for code generation. To determine which of these restrictions apply to specific library functions, see the table in “Toolbox Functions with Restrictions for Variable-Size Data” on page 6-34.

Variable-length vector restriction

Inputs to the library function must be variable-length vectors or fixed-size vectors. A variable-length vector is a variable-size array that has the shape `1x:n` or `:nx1` (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

This restriction applies to functions that take the working dimension (the dimension along which to operate) as input. In MATLAB and in code generation, if you do not supply the working dimension, the function selects it. In MATLAB, the function selects the first dimension whose size does not equal 1. For code generation, the function selects the first dimension that has a variable size or that has a fixed size that does not equal 1. If the working dimension has a variable size and it becomes 1 at run time, then the working dimension is different from the working dimension in MATLAB. Therefore, when run-time error checks are enabled, an error can occur.

For example, suppose that `X` is a variable-size matrix with dimensions `1x:3x:5`. In the generated code, `sum(X)` behaves like `sum(X,2)`. In MATLAB, `sum(X)` behaves like `sum(X,2)` unless `size(X,2)` is 1. In MATLAB, when `size(X,2)` is 1, `sum(X)` behaves like `sum(X,3)`.

To avoid this issue, specify the intended working dimension explicitly as a constant value. For example, `sum(X,2)`.

Array-to-vector restriction

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

Array-to-scalar restriction

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

Toolbox Functions with Restrictions for Variable-Size Data

The following table list functions that have code generation restrictions for variable-size data. For additional restrictions for these functions, and restrictions for all functions and objects supported for code generation, see "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List" on page 3-2.

Function	Restrictions for Variable-Size Data
`all`	• See "Automatic dimension restriction" on page 6-33. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`any`	• See "Automatic dimension restriction" on page 6-33. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`bsxfun`	• Dimensions expand only where one input array or the other has a fixed length of 1.
`cat`	• Dimension argument must be a constant. • An error occurs if variable-size inputs are empty at run time.
`conv`	• See "Variable-length vector restriction" on page 6-33. • Input vectors must have the same orientation, either both row vectors or both column vectors.
`cov`	• For `cov(X)`, see "Array-to-vector restriction" on page 6-34.

Function	Restrictions for Variable-Size Data
`cross`	• Variable-size array inputs that become vectors at run time must have the same orientation.
`deconv`	• For both arguments, see "Variable-length vector restriction" on page 6-33.
`detrend`	• For first argument for row vectors only, see "Array-to-vector restriction" on page 6-34 .
`diag`	• See "Array-to-vector restriction" on page 6-34 .
`diff`	• See "Automatic dimension restriction" on page 6-33. • 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`	• See "Automatic dimension restriction" on page 6-33.
`filter`	• For first and second arguments, see "Variable-length vector restriction" on page 6-33. • See "Automatic dimension restriction" on page 6-33.
`hist`	• For second argument, see "Variable-length vector restriction" on page 6-33. • For second input argument, see "Array-to-scalar restriction" on page 6-34.
`histc`	• See "Automatic dimension restriction" on page 6-33.
`ifft`	• See "Automatic dimension restriction" on page 6-33.
`ind2sub`	• First input (the size vector input) must be fixed size.
`interp1`	• For the `xq` input, see "Array-to-vector restriction" on page 6-34. • If `v` becomes a row vector at run time, the array to vector restriction applies. If `v` becomes a column vector at run time, this restriction does not apply.
`ipermute`	• Order input must be fixed size.
`issorted`	• For optional rows input, see "Variable-length vector restriction" on page 6-33.

Function	Restrictions for Variable-Size Data
`magic`	• Argument must be a constant. • Output can be fixed-size matrices only.
`max`	• See “Automatic dimension restriction” on page 6-33.
`mean`	• See “Automatic dimension restriction” on page 6-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`median`	• See “Automatic dimension restriction” on page 6-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`min`	• See “Automatic dimension restriction” on page 6-33.
`mode`	• See “Automatic dimension restriction” on page 6-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`mtimes`	Consider the multiplication `A*B`. If the code generator is aware that `A` is scalar and `B` is a matrix, the code generator produces code for scalar-matrix multiplication. However, if the code generator is aware that `A` and `B` are variable-size matrices, it produces code for a general matrix multiplication. At run time, if `A` turns out to be scalar, the generated code does not change its behavior. Therefore, when run-time error checks are enabled, a size mismatch error can occur.
`nchoosek`	• The second input, `k`, must be a fixed-size scalar. • The second input, `k`, must be a constant for static allocation. If you enable dynamic allocation, the second input can be a variable. • You cannot create a variable-size array by passing in a variable, `k`, unless you enable dynamic allocation.
`permute`	• Order input must be fixed-size.
`planerot`	• 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.

Function	Restrictions for Variable-Size Data
`poly`	- See "Variable-length vector restriction" on page 6-33.
`polyfit`	- For first and second arguments, see "Variable-length vector restriction" on page 6-33.
`prod`	- See "Automatic dimension restriction" on page 6-33. - An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`rand`	- For an upper-bounded variable `N`, `rand(1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. - For an upper-bounded variable `N`, `rand([1 N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`randi`	- For an upper-bounded variable `N`, `randi(imax,1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. - For an upper-bounded variable `N`, `randi(imax,[1 N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`randn`	- For an upper-bounded variable `N`, `randn(1,N)` produces a variable-length vector of `1x:M` where `M` is the upper bound on `N`. - For an upper-bounded variable `N`, `randn([1 N])` may produce a variable-length vector of `:1x:M` where `M` is the upper bound on `N`.
`reshape`	- If the input is a variable-size array and the output array has at least one fixed-length dimension, do not specify the output dimension sizes in a size vector `sz`. Instead, specify the output dimension sizes as scalar values, `sz1,...,szN`. Specify fixed-size dimensions as constants. - When the input is a variable-size empty array, the maximum dimension size of the output array (also empty) cannot be larger than that of the input.

Function	Restrictions for Variable-Size Data
`roots`	• See "Variable-length vector restriction" on page 6-33.
`shiftdim`	• 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. Therefore, at run time the number of shifts is constant. • An error occurs if the dimension that is shifted to the first dimension has length 1 at run time. To avoid the error, supply the number of shifts as the second input argument (must be a constant). • First input argument must have the same number of dimensions when you supply a positive number of shifts.
`std`	• See "Automatic dimension restriction" on page 6-33. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
`sub2ind`	• First input (the size vector input) must be fixed size.
`sum`	• See "Automatic dimension restriction" on page 6-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`trapz`	• See "Automatic dimension restriction" on page 6-33. • An error occurs if you pass as the first argument a variable-size matrix that is 0-by-0 at run time.
`typecast`	• See "Variable-length vector restriction" on page 6-33 on first argument.
`var`	• See "Automatic dimension restriction" on page 6-33. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.

Code Generation for MATLAB Structures

Structure Definition for Code Generation

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 7-3
Observe restrictions on properties and values of scalar structures.	"Define Scalar Structures for Code Generation" on page 7-4
Make structures uniform in arrays.	"Define Arrays of Structures for Code Generation" on page 7-6
Reference structure fields individually during indexing.	"Index Substructures and Fields" on page 7-8
Avoid type mismatch when assigning values to structures and fields.	"Assign Values to Structures and Fields" on page 7-10

Structure Operations Allowed for Code Generation

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

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

Define Scalar Structures for Code Generation

In this section...

Restrictions When Defining Scalar Structures by Assignment

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

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

Adding Fields in Consistent Order on Each Control Flow Path

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

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

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

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

Restriction on Adding New Fields After First Use

You cannot add fields to a structure after you perform the following operations on the structure:

- Reading from the structure
- Indexing into the structure array
- Passing the structure to a function

For example, consider this code:

```
...
x.c = 10; % Defines structure and creates field c
y = x; % Reads from structure
x.d = 20; % Generates an error
...
```

In this example, the attempt to add a new field `d` after reading from structure `x` generates an error.

This restriction extends across the structure hierarchy. For example, you cannot add a field to a structure after operating on one of its fields or nested structures, as in this example:

```
function y = fcn(u) %#codegen

x.c = 10;
y = x.c;
x.d = 20; % Generates an error
```

In this example, the attempt to add a new field `d` to structure `x` after reading from the structure's field `c` generates an error.

Define Arrays of Structures for Code Generation

In this section...

Ensuring Consistency of Fields

For code generation, when you create an array of MATLAB structures, corresponding fields in the array elements must have the same size, type, and complexity.

Once you have created the array of structures, you can make the structure fields variable-size using `coder.varsize`. For more information, see "Declare a Variable-Size Structure Field.".

Using repmat to Define an Array of Structures with Consistent Field Properties

You can create an array of structures from a scalar structure by using the MATLAB `repmat` function, which replicates and tiles an existing scalar structure:

1. Create a scalar structure, as described in "Define Scalar Structures for Code Generation" on page 7-4.
2. Call `repmat`, passing the scalar structure and the dimensions of the array.
3. Assign values to each structure using standard array indexing and structure dot notation.

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

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

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

Defining an Array of Structures by Using `struct`

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

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

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

Defining an Array of Structures Using Concatenation

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

For example, the following code uses concatenation and a local function to create the elements of a 1-by-3 structure array:

```
...
W = [ sab(1,2) sab(2,3) sab(4,5) ];

function s = sab(a,b)
  s.a = a;
  s.b = b;
...
```

Index Substructures and Fields

Use these guidelines when indexing substructures and fields for code generation:

Reference substructure field values individually using dot notation

For example, the following MATLAB code uses dot notation to index fields and substructures:

```
...
substruct1.a1 = 15.2;
substruct1.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                  'ele3',substruct1);

substruct2 = mystruct;
substruct2.ele3.a2 = 2*(substruct1.a2);
...
```

The generated code indexes elements of the structures in this example by resolving symbols as follows:

Dot Notation	Symbol Resolution
`substruct1.a1`	Field `a1` of local structure `substruct1`
`substruct2.ele3.a1`	Value of field `a1` of field `ele3`, a substructure of local structure `substruct2`
`substruct2.ele3.a2(1,1)`	Value in row 1, column 1 of field `a2` of field `ele3`, a substructure of local structure `substruct2`

Reference field values individually in structure arrays

To reference the value of a field in a structure array, you must index into the array to the structure of interest and then reference that structure's field individually using dot notation, as in this example:

```
...
y = X(1).a % Extracts the value of field a
           % of the first structure in array X
...
```

To reference all the values of a particular field for each structure in an array, use this notation in a `for` loop, as in this example:

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,5);
for i = 1:5
    X(i).a = i;
    X(i).b = i+1;
end
```

This example uses the `repmat` function to define an array of structures, each with two fields `a` and `b` as defined by `s`. See “Define Arrays of Structures for Code Generation” on page 7-6 for more information.

Do not reference fields dynamically

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

Assign Values to Structures and Fields

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

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

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

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

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

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

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

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

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

Do not assign mxArrays to structures

You cannot assign `mxArrays` to structure elements; convert `mxArrays` to known types before code generation (see “Working with mxArrays” on page 13-15).

Do not assign classes to structure fields

You cannot assign classes to structure fields.

Do not assign cell arrays to structure fields

You cannot assign cell arrays to structure fields.

Code Generation for Cell Arrays

Code Generation for Cell Arrays

When you generate code from MATLAB code that contains cell arrays, the code generator classifies the cell arrays as *homogeneous* or *heterogeneous*. This classification determines how a cell array is represented in the generated code. It also determines how you can use the cell array in MATLAB code from which you generate code.

When you use cell arrays in MATLAB code that is intended for code generation, you must adhere to certain restrictions. See “Cell Array Limitations for Code Generation” on page 8-10.

Homogeneous vs. Heterogeneous Cell Arrays

A homogeneous cell array has these characteristics:

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

A heterogeneous cell array has these characteristics:

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

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

Controlling Whether a Cell Array Is Homogeneous or Heterogeneous

For cell arrays with certain characteristics, you cannot control the classification as homogeneous or heterogeneous:

- If the elements have different classes, the cell array must be heterogeneous.
- If the cell array is variable-size, it must be homogeneous.
- If you index into the cell array with an index whose value is determined at run time, the cell array must be homogeneous.

For other cell arrays, you can control the classification as homogeneous or heterogeneous.

To control the classification of cell arrays that are entry-point function inputs:

- At the command line, use the coder.CellType methods makeHomogeneous or makeHeterogeneous.
- In the MATLAB Coder app, select **cell (Homogeneous)** or **cell (Heterogeneous)** from the type menu. See “Define or Edit Input Parameter Type by Using the App” on page 17-17.

To control the classification of cell arrays that are not entry-point function inputs:

- If the cell array is fixed-size, you can force an otherwise homogeneous cell array to be heterogeneous by using coder.cstructname. For example:

  ```
  function y =  mycell()
  %#codegen
  c = {1 2 3};
  coder.cstructname(c, 'myname');
  y = c;
  end
  ```

- If the cell array elements have the same class, you can force a cell array to be homogeneous by using coder.varsize. See “Control Whether a Cell Array Is Variable-Size” on page 8-6.

Naming the Structure Type That Represents a Heterogeneous Cell Array in the Generated Code

The code generator represents a heterogeneous cell array as a structure in the generated code. You can name the generated structure type. You cannot name the fields of the structure.

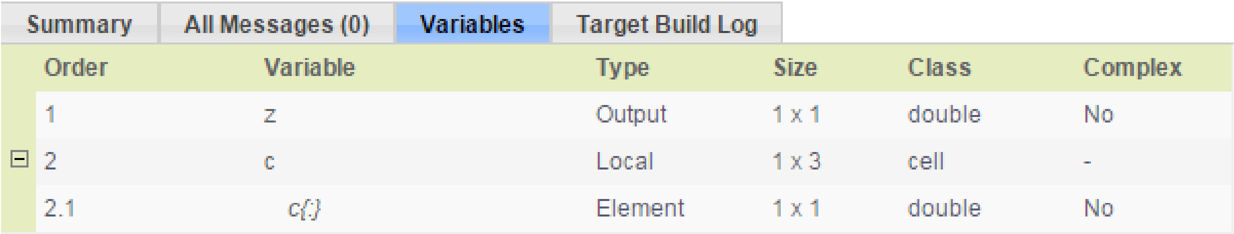
If the cell array is an entry-point function input, see "Define Cell Array Inputs" on page 8-9. If the cell array is not an entry-point function input, use `coder.cstructname` in the MATLAB function. For example:

```
function y =  mycell()
%#codegen
c = {1 'a'};
coder.cstructname(c, 'myname');
y = c;
end
```

Cell Arrays in Reports

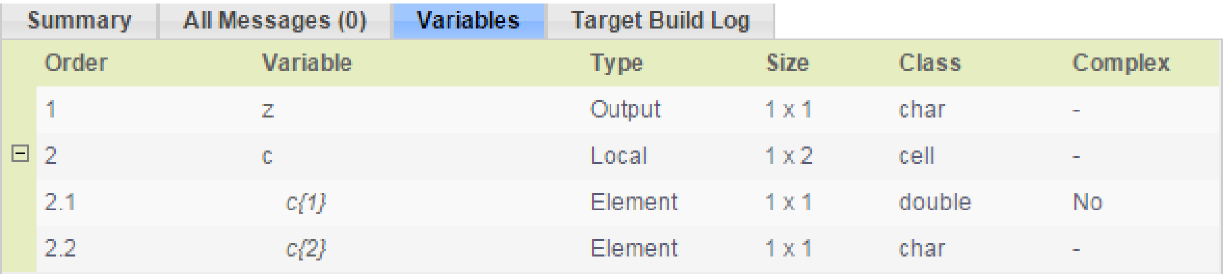
To see whether a cell array is homogeneous or heterogeneous, view the variable in the code generation report.

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



Order	Variable	Type	Size	Class	Complex
1	z	Output	1 x 1	double	No
⊟ 2	c	Local	1 x 3	cell	-
2.1	*c{:}*	Element	1 x 1	double	No

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



Order	Variable	Type	Size	Class	Complex
1	z	Output	1 x 1	char	-
⊟ 2	c	Local	1 x 2	cell	-
2.1	*c{1}*	Element	1 x 1	double	No
2.2	*c{2}*	Element	1 x 1	char	-

See Also

coder.CellType | `coder.cstructname` | `coder.varsize`

More About

- “Control Whether a Cell Array Is Variable-Size” on page 8-6
- “Cell Array Limitations for Code Generation” on page 8-10
- “Code Generation Reports” on page 21-9

Control Whether a Cell Array Is Variable-Size

The code generator classifies a variable-size cell array as homogeneous. The cell array elements must have the same class. In the generated code, the cell array is represented as an array.

If a cell array is an entry-point function input, to make it variable-size:

- At the command line, you can use the `coder.typeof` function or the `coder.newtype` function to create a type for a variable-size cell array. For example, to create a type for a cell array whose first dimension is fixed and whose second dimension has an upper bound of 10, use this code:

  ```
  t = coder.typeof({1 2 3}, [1 10], [0 1])
  ```

  See "Specify Variable-Size Cell Array Inputs" on page 20-61.
- In the MATLAB Coder app, select **Homogeneous cell array** as the type of the input. For the variable-size dimension, specify that it is unbounded or has an upper bound.

If a cell array is not an entry-point function input, to make it variable-size:

- Create the cell array by using the `cell` function. For example:

  ```
  function z = mycell(n, j)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  end
  z = x{j};
  end
  ```

  For code generation, when you create a variable-size cell array by using `cell`, you must adhere to certain restrictions. See "Definition of Variable-Size Cell Array by Using cell" on page 8-11.
- Grow the cell array. For example:

  ```
  function z = mycell(n)
  %#codegen
  c = {1 2 3};
  for i = 1:n
      c{end + 1} = 1;
  ```

```
end
z = c{n};
end
```

- Force the cell array to be variable-size by using `coder.varsize`. Consider this code:

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

  Without `coder.varsize`, `c` is fixed-size with dimensions 1-by-3. With `coder.varsize`, `c` is variable-size with an upper bound of 10.

  Sometimes, using `coder.varsize` changes the classification of a cell array from heterogeneous to homogeneous. Consider this code:

  ```
  function y =  mycell()
  %#codegen
  c = {1 [2 3]};
  y = c{2};
  end
  ```

  The code generator classifies `c` as heterogeneous because the elements have different sizes. `c` is fixed-size with dimensions 1-by-2. If you use `coder.varsize` with `c`, it becomes homogeneous. For example:

  ```
  function y =  mycell()
  %#codegen
  c = {1 [2 3]};
  coder.varsize('c', [1 10], [0 1]);
  y = c{2};
  end
  ```

  `c` becomes a variable-size homogeneous cell array with dimensions 1-by-:10.

  To force `c` to be homogeneous, but not variable-size, specify that none of the dimensions vary. For example:

  ```
  function y =  mycell()
  %#codegen
  c = {1 [2 3]};
  coder.varsize('c', [1 2], [0 0]);
  ```

```
y = c{2};
end
```

See Also

coder.CellType | `coder.varsize`

More About

- "Code Generation for Cell Arrays" on page 8-2
- "Cell Array Limitations for Code Generation" on page 8-10
- "Code Generation for Variable-Size Arrays" on page 6-2

Define Cell Array Inputs

To define types for cell arrays that are inputs to entry-point functions, use one of these approaches:

To Define Types:	See
At the command line	“Specify Cell Array Inputs at the Command Line” on page 20-57
Programmatically in the MATLAB file	“Define Input Properties Programmatically in the MATLAB File” on page 20-68
In the MATLAB Coder app	“Automatically Define Input Types by Using the App” on page 17-4 “Define Input Parameter by Example by Using the App” on page 17-7 “Define or Edit Input Parameter Type by Using the App” on page 17-17

See Also

coder.CellType

More About

- “Code Generation for Cell Arrays” on page 8-2

Cell Array Limitations for Code Generation

When you use cell arrays in MATLAB code that is intended for code generation, you must adhere to these restrictions:

- "Cell Array Element Assignment" on page 8-10
- "Definition of Variable-Size Cell Array by Using cell" on page 8-11
- "Cell Array Indexing" on page 8-14
- "Growing a Cell Array by Using {end + 1}" on page 8-15
- "Variable-Size Cell Arrays" on page 8-16
- "Cell Array Contents" on page 8-17
- "Cell Arrays in Structures" on page 8-17
- "Passing Cell Arrays to External C/C++ Functions" on page 8-17

Cell Array Element Assignment

You must assign a cell array element on all execution paths before you use it. For example:

```
function z = foo(n)
%#codegen
c = cell(1,3);
if n < 1
    c{2} = 1;

else
    c{2} = n;
end
z = c{2};
end
```

The code generator considers passing a cell array to a function or returning it from a function as a use of all elements of the cell array. Therefore, before you pass a cell array to a function or return it from a function, you must assign all of its elements. For example, the following code is not allowed because it does not assign a value to `c{2}` and `c` is a function output.

```
function c = foo()
```

```
%#codegen
c = cell(1,3);
c{1} = 1;
c{3} = 3;
end
```

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

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

Definition of Variable-Size Cell Array by Using `cell`

For code generation, before you use a cell array element, you must assign a value to it. When you use `cell` to create a variable-size cell array, for example, `cell(1,n)`, MATLAB assigns an empty matrix to each element. However, for code generation, the elements are unassigned. For code generation, after you use `cell` to create a variable-size cell array, you must assign all elements of the cell array before any use of the cell array. For example:

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

The code generator analyzes your code to determine whether all elements are assigned before the first use of the cell array. If the code generator detects that some elements are not assigned, code generation fails with a message like this message:

```
Unable to determine that every element of 'y' is assigned before this line.
```

Sometimes, even though your code assigns all elements of the cell array, the code generator reports this message because the analysis does not detect that all elements are assigned. See “Unable to Determine That Every Element of Cell Array Is Assigned” on page 30-12.

To avoid this error, follow these guidelines:

- When you use `cell` to define a variable-size cell array, write code that follows this pattern:

  ```
  function z = mycell(n, j)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  end
  z = x{j};
  end
  ```

  Here is the pattern for a multidimensional cell array:

  ```
  function z = mycell(m,n,p)
  %#codegen
  x = cell(m,n,p);
  for i = 1:m
      for j =1:n
          for k = 1:p
              x{i,j,k} = i+j+k;
          end
      end
  end
  z = x{m,n,p};
  end
  ```
- Increment or decrement the loop counter by `1`.
- Define the cell array within one loop or one set of nested loops. For example, this code is not allowed:

  ```
  function z = mycell(n, j)
  x = cell(1,n);
  for i = 1:5
      x{i} = 5;
  ```

```
end
for i = 6:n
    x{i} = 5;
end
z = x{j};
end
```

- Use the same variables for the cell dimensions and loop initial and end values. For example, code generation fails for the following code because the cell creation uses `n` and the loop end value uses `m`:

  ```
  function z = mycell(n, j)
  x = cell(1,n);
  m = n;
  for i = 1:m
      x{i} = 2;
  end
  z = x{j};
  end
  ```

  Rewrite the code to use `n` for the cell creation and the loop end value:

  ```
  function z = mycell(n, j)
  x = cell(1,n);
  for i = 1:n
      x{i} = 2;
  end
  z = x{j};
  end
  ```

- Create the cell array with this pattern:

  ```
  x = cell(1,n)
  ```

  Do not assign the cell array to a field of a structure or a property of an object. For example, this code is not allowed:

  ```
  myobj.prop = cell(1,n)
  for i = 1:n
  ...
  end
  ```

  Do not use the `cell` function inside the cell array constructor `{}`. For example, this code is not allowed:

  ```
  x = {cell(1,n)};
  ```

- The cell array creation and the loop that assigns values to the cell array elements must be together in a unique execution path. For example, the following code is not allowed.

  ```
  function z = mycell(n)
  if n > 3
      c = cell(1,n);
  else
      c = cell(n,1);
  end
  for i = 1:n
      c{i} = i;
  end
  z = c{n};
  end
  ```

  To fix this code, move the assignment loop inside the code block that creates the cell array.

  ```
  function z = cellerr(n)
  if n > 3
      c = cell( 1,n);
      for i = 1:n
          c{i} = i;
      end
  else
      c = cell(n,1);
      for i = 1:n
          c{i} = i;
      end
  end
  z = c{n};
  end
  ```

Cell Array Indexing

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

  For example, the following code is not allowed.

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

You can index into a heterogeneous cell array in a `for`-loop with constant bounds because the code generator unrolls the loop. Unrolling creates a separate copy of the loop body for each loop iteration, which makes the index in each loop iteration constant. However, if the `for`-loop has a large body or it has many iterations, the unrolling can increase compile time and generate inefficient code.

If `A` and `B` are constant, the following code shows indexing into a heterogeneous cell array in a `for`-loop with constant bounds.

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

Growing a Cell Array by Using {end + 1}

To grow a cell array `X`, you can use `X{end + 1}`. For example:

```
...
X = {1 2};
X{end + 1} = 'a';
...
```

When you use `{end + 1}` to grow a cell array, follow these restrictions:

- Use only `{end + 1}`. Do not use `{end + 2}`, `{end + 3}`, and so on.
- Use `{end + 1}` with vectors only. For example, the following code is not allowed because `X` is a matrix, not a vector:

  ```
  ...
  X = {1 2; 3 4};
  X{end + 1} = 5;

  ...
  ```
- Use `{end + 1}` only with a variable. In the following code, `{end + 1}` does not cause `{1 2 3}` to grow. In this case, the code generator treats `{end + 1}` as an out-of-bounds index into `X{2}`.

  ```
  ...
  ```

```
X = {'a' { 1 2 3 }};
X{2}{end + 1} = 4;
...
```

- When `{end + 1}` grows a cell array in a loop, the cell array must be variable-size. Therefore, the cell array must be homogeneous.

  This code is allowed because `X` is homogeneous.

  ```
  ...
  X = {1  2};
  for i=1:n
      X{end + 1} = 3;
  end
  ...
  ```

  This code is not allowed because `X` is heterogeneous.

  ```
  ...
  X = {1 'a' 2 'b'};
  for i=1:n
      X{end + 1} = 3;
  end
  ...
  ```

Variable-Size Cell Arrays

- Heterogeneous cell arrays cannot be variable-size. See “Control Whether a Cell Array Is Variable-Size” on page 8-6.
- If you use `coder.varsize` to make a variable-size cell array, define the cell array with curly braces. For example:

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

  Do not use the `cell` function. For example, this code is not allowed:

  ```
  ...
  c = cell(1,3);
  coder.varsize('c')
  ...
  ```

Cell Array Contents

Cell arrays cannot contain `mxarrays`. In a cell array, you cannot store a value that an extrinsic function returns.

Cell Arrays in Structures

Structures cannot contain cell arrays.

Passing Cell Arrays to External C/C++ Functions

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

More About

- "Code Generation for Cell Arrays" on page 8-2
- "Differences Between Generated Code and MATLAB Code" on page 2-8

Code Generation for Enumerated Data

Code Generation for Enumerations

Enumerations represent a fixed set of named values. Enumerations help make your MATLAB code and generated C/C++ code more readable. For example, the generated code can test equality with code such as `if (x == Red)` instead of using `strcmp`.

For code generation, when you use enumerations, adhere to these restrictions:

- "Define Enumerations for Code Generation" on page 9-2
- "Use Allowed Operations on Enumerations" on page 9-3
- "Use MATLAB Toolbox Functions That Support Enumerations" on page 9-4

Define Enumerations for Code Generation

For code generation, the enumeration class must derive from one of these base types: `int8`, `uint8`, `int16`, `uint16`, or `int32`. For example:

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

You can use the base type to control the size of an enumerated type in generated C/C++ code. You can:

- Represent an enumerated type as a fixed-size integer that is portable to different targets.
- Reduce memory usage.
- Interface with legacy code.
- Match company standards.

The base type determines the representation of the enumerated type in generated C/C++ code.

If the base type is `int32`, the code generator produces a C enumerated type. Consider the following MATLAB enumerated type definition:

```
classdef LEDcolor < int32
    enumeration
```

```
        GREEN(1),
        RED(2)
    end
end
```

This enumerated type definition results in the following C code:

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

typedef enum LEDcolor LEDcolor;
```

For built-in integer base types other than `int32`, the code generator produces a `typedef` statement for the enumerated type and `#define` statements for the enumerated values. Consider the following MATLAB enumerated type definition:

```
classdef LEDcolor < int16
    enumeration
        GREEN(1),
        RED(2)
    end

end
```

This enumerated type definition results in the following C code:

```
typedef short LEDcolor;
#define GREEN ((LEDcolor)1)
#define RED ((LEDcolor)2)
```

The C type in the `typedef` statement depends on:

- The integer sizes defined for the production hardware in the hardware implementation object or the project settings. See coder.HardwareImplementation.
- The setting that determines the use of built-in C types or MathWorks typedefs in the generated code. See “Specify Data Types Used in Generated Code” on page 20-38 and “How MATLAB Coder Infers C/C++ Data Types” on page 26-9 .

Use Allowed Operations on Enumerations

For code generation, you are restricted to the operations on enumerations listed in this table.

Operation	Example

assignment operator, =	`xon = LEDcolor.GREEN`
	`xoff = LEDcolor.RED`

relational operators,`< > <= >= == ~=` You cannot use == or ~= to test equality between an enumeration member and a character array or cell array of character arrays.	`xon == xoff`
cast operation	`double(LEDcolor.RED)`

indexing operation	`m = [1 2]`
	`n = LEDcolor(m)`
	`p = n(LEDcolor.GREEN)`

control flow statements: if, switch, while	`if state == sysMode.ON`
	`    led = LEDcolor.GREEN;`
	`else`
	`    led = LEDcolor.RED;`
	`end`

Use MATLAB Toolbox Functions That Support Enumerations

For code generation, you can use enumerations with these MATLAB toolbox functions:

- `cast`
- `cat`
- `circshift`
- `fliplr`
- `flipud`
- `histc`
- `intersect`
- `ipermute`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`

- `ismember`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `setdiff`
- `setxor`
- `shiftdim`
- `sort`
- `sortrows`
- `squeeze`
- `union`
- `unique`

More About

- "Generate Code for an LED Control Function That Uses Enumerated Types" on page 20-188
- "Customize Enumerated Types in Generated Code" on page 9-6

Customize Enumerated Types in Generated Code

For code generation, to customize an enumeration, in the static methods section of the class definition, include customized versions of the methods listed in this table.

Method	Description	Default Value Returned or Specified	When to Use
`getDefaultValue`	Returns the default enumerated value.	First value in the enumeration class definition.	For a default value that is different than the first enumeration value, provide a `getDefaultValue` method that returns the default value that you want. See "Specify a Default Enumeration Value" on page 9-7.
`getHeaderFile`	Specifies the file that defines an externally defined enumerated type.	`''`	To use an externally defined enumerated type, provide a `getHeaderFile` method that returns the path to the header file that defines the type. In this case, the code generator does not produce the class definition. See "Specify a Header File" on page 9-8
`addClassNameToEnumNames`	Specifies whether the class name becomes a prefix in the generated code.	`false` — prefix is not used.	If you want the class name to become a prefix in the generated code, set the return value of the

Method	Description	Default Value Returned or Specified	When to Use
			addClassNameToEnumNames method to true. See "Include Class Name Prefix in Generated Enumerated Type Value Names" on page 9-8.

Specify a Default Enumeration Value

If the value of a variable that is cast to an enumerated type does not match one of the enumerated type values:

- Generated MEX reports an error.
- Generated C/C++ code replaces the value of the variable with the enumerated type default value.

Unless you specify otherwise, the default value for an enumerated type is the first value in the enumeration class definition. To specify a different default value, add your own getDefaultValue method to the methods section. In this example, the first enumeration member value is LEDcolor.GREEN, but the getDefaultValue method returns LEDcolor.RED:

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

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

Specify a Header File

To specify that an enumerated type is defined in an external file, provide a customized getHeaderFile method. This example specifies that LEDcolor is defined in the external file my_LEDcolor.h.

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

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

You must provide my_LEDcolor.h. For example:

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

Include Class Name Prefix in Generated Enumerated Type Value Names

By default, the generated enumerated type value name does not include the class name prefix. For example:

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

typedef enum LEDcolor LEDcolor;
```

To include the class name prefix, provide an addClassNameToEnumNames method that returns true. For example:

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

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

In the generated type definition, the enumerated value names include the class prefix `LEDcolor`.

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

typedef enum LEDcolor LEDcolor;
```

More About

- Modifying Superclass Methods and Properties (MATLAB)
- “Code Generation for Enumerations” on page 9-2

Code Generation for MATLAB Classes

MATLAB Classes Definition for Code Generation

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

What's Different	More Information
Class must be in a single file. Because of this limitation, code generation is not supported for a class definition that uses an `@`-folder.	"Create a Single, Self-Contained Class Definition File" (MATLAB)
Restricted set of language features.	"Language Limitations" on page 10-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 10-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 10-4
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 10-14 "Handle Object Limitations for Code Generation" on page 10-22
Calls to base class constructor.	"Calls to Base Class Constructor" on page 10-6
Global variables containing MATLAB objects are not supported for code generation.	N/A
Inheritance from built-in MATLAB classes is not supported.	"Inheritance from Built-In MATLAB Classes Not Supported" on page 10-7

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, classes have a built-in static method, `empty`, which creates an empty array of the class. Code generation does not support this method.
- The following MATLAB handle class methods:

  - `addlistener`
  - `delete`
  - `eq`
  - `findobj`
  - `findpro`
- The `AbortSet` property attribute

Code Generation Features Not Compatible with Classes

- You can generate code for entry-point MATLAB functions that use classes, but you cannot generate code directly for a MATLAB class.

  For example, if `ClassNameA` is a class definition, you cannot generate code by executing:

  ```
  codegen ClassNameA
  ```
- A handle class object cannot be an entry-point function input or output.
- A value class object can be an entry-point function input or output. However, if a value class object contains a handle class object, then the value class object cannot be

an entry-point function input or output. A handle class object cannot be an entry-point function input or output.

- Code generation does not support classes in matrices or structures. As a workaround, consider using cell arrays because code generation supports classes in cell arrays.
- Code generation does not support assigning an object of a value class into a nontunable property. For example, `obj.prop=v;` is invalid when `prop` is a nontunable property and `v` is an object based on a value class.
- You cannot use `coder.extrinsic` to declare a class or method as extrinsic.
- You cannot pass a MATLAB class to the `coder.ceval` function.
- If you use classes in code in the MATLAB Function block, you cannot use the debugger to view class information.
- The `coder.nullcopy` function does not support MATLAB classes as inputs.
- The `coder.Constant` function does not support MATLAB classes as inputs.

Defining Class Properties for Code Generation

For code generation, you must define class properties differently than you do when running your code in the MATLAB environment:

- Code generation does not support the property restriction syntax. For example, the following class definition is not allowed because it uses the property restriction syntax to restrict the types of the `Number` and `Today` properties.

  ```
  classdef Myclass
     properties
        Number double
        Today char = date;
     end
  end
  ```
- After defining a property, do not assign it an incompatible type. Do not use a property before attempting to grow it.

  When you define class properties for code generation, consider the same factors that you take into account when defining variables. In the MATLAB language, variables can change their class, size, or complexity dynamically at run time so you can use the same variable to hold a value of varying class, size, or complexity. C and C++ use static typing. Before using variables, to determine their type, the code generator requires a complete assignment to each variable. Similarly, before using properties, you must explicitly define their class, size, and complexity.

- Initial values:
  - If the property does not have an explicit initial value, the code generator assumes that it is undefined at the beginning of the constructor. The code generator does not assign an empty matrix as the default.
  - If the property does not have an initial value and the code generator cannot determine that the property is assigned prior to first use, the software generates a compilation error.
  - For System objects, if a nontunable property is a structure, you must completely assign the structure. You cannot do partial assignment using subscripting.

    For example, for a nontunable property, you can use the following assignment:

    ```
    mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
    ```

    You cannot use the following partial assignments:

    ```
    mySystemObject.nonTunableProperty.fieldA = 'a';
    mySystemObject.nonTunableProperty.fieldB = 'b';
    ```
  - `coder.varsize` is not supported for class properties.
  - If the initial value of a property is an object, then the property must be constant. To make a property constant, declare the `Constant` attribute in the property block. For example:

    ```
    classdef MyClass
        properties (Constant)
            p1 = MyClass2;
        end
    end
    ```
  - MATLAB computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable computed when the class loads belongs to MATLAB; it is not the value used at code generation time. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns `true (1)`.
- 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.
- If a property is constant and its value is an object, you cannot change the value of a property of that object. For example, suppose that:

- obj is an object of myClass1.
- myClass1 has a constant property p1 that is an object of myClass2.
- myClass2 has a property p2.

Code generation does not support the following code:

```
obj.p1.p2 = 1;
```

Calls to Base Class Constructor

If a class constructor contains a call to the constructor of the base class, the call to the base class constructor must come before for, if, return, switch or while statements.

For example, if you define a class B based on class A:

```
classdef B < A
    methods
        function obj = B(varargin)
            if nargin == 0
                a = 1;
                b = 2;
            elseif nargin == 1
                a = varargin{1};
                b = 1;
            elseif nargin == 2
                a = varargin{1};
                b = varargin{2};
            end
            obj = obj@A(a,b);
        end

    end
end
```

Because the class definition for B uses an if statement before calling the base class constructor for A, you cannot generate code for function callB:

```
function [y1,y2] = callB
x = B;
y1 = x.p1;
y2 = x.p2;
end
```

However, you can generate code for `callB` if you define class `B` as:

```
classdef B < A
    methods
        function obj = NewB(varargin)
            [a,b] = getaandb(varargin{:});
            obj = obj@A(a,b);
        end

    end
end

function [a,b] = getaandb(varargin)
if nargin == 0
    a = 1;
    b = 2;
elseif nargin == 1
    a = varargin{1};
    b = 1;
elseif nargin == 2
    a = varargin{1};
    b = varargin{2};
end
end
```

Inheritance from Built-In MATLAB Classes Not Supported

You cannot generate code for classes that inherit from built-in MATLAB classes. For example, you cannot generate code for the following class:

```
classdef myclass < double
```

Classes That Support Code Generation

You can generate code for MATLAB value and handle classes and user-defined System objects. Your class can have multiple methods and properties and can inherit from multiple classes.

To generate code for:	Example:
Value classes	“Generate Code for MATLAB Value Classes” on page 10-9
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 10-14

For more information, see:

- “Role of Classes in MATLAB” (MATLAB)
- “MATLAB Classes Definition for Code Generation” on page 10-2

Generate Code for MATLAB Value Classes

This example shows how to generate code for a MATLAB value class and then view the generated code in the code generation report.

1 In a writable folder, create a MATLAB value class, Shape. Save the code as Shape.m.

```
classdef Shape
% SHAPE Create a shape at coordinates
% centerX and centerY
    properties
        centerX;
        centerY;
    end
    properties (Dependent = true)
        area;
    end
    methods
        function out = get.area(obj)
            out =  obj.getarea();
        end
        function obj = Shape(centerX,centerY)
            obj.centerX = centerX;
            obj.centerY = centerY;
        end
    end
    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.

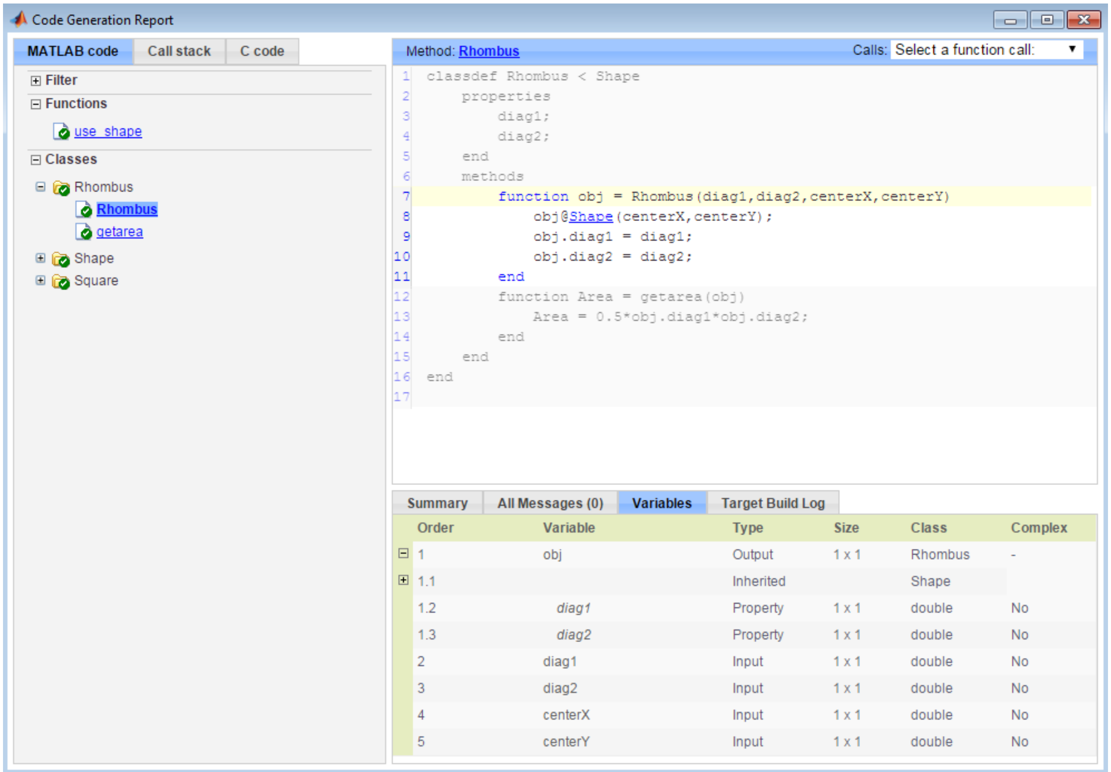
```
codegen -config:lib -report use_shape
```

`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 the variables used in the class. If a variable is a MATLAB object, by default, the report displays the object without displaying its properties. To view the list of properties, expand the list. Within the list of properties, the list of inherited properties is collapsed. In the following report, the lists of properties and inherited properties are expanded.

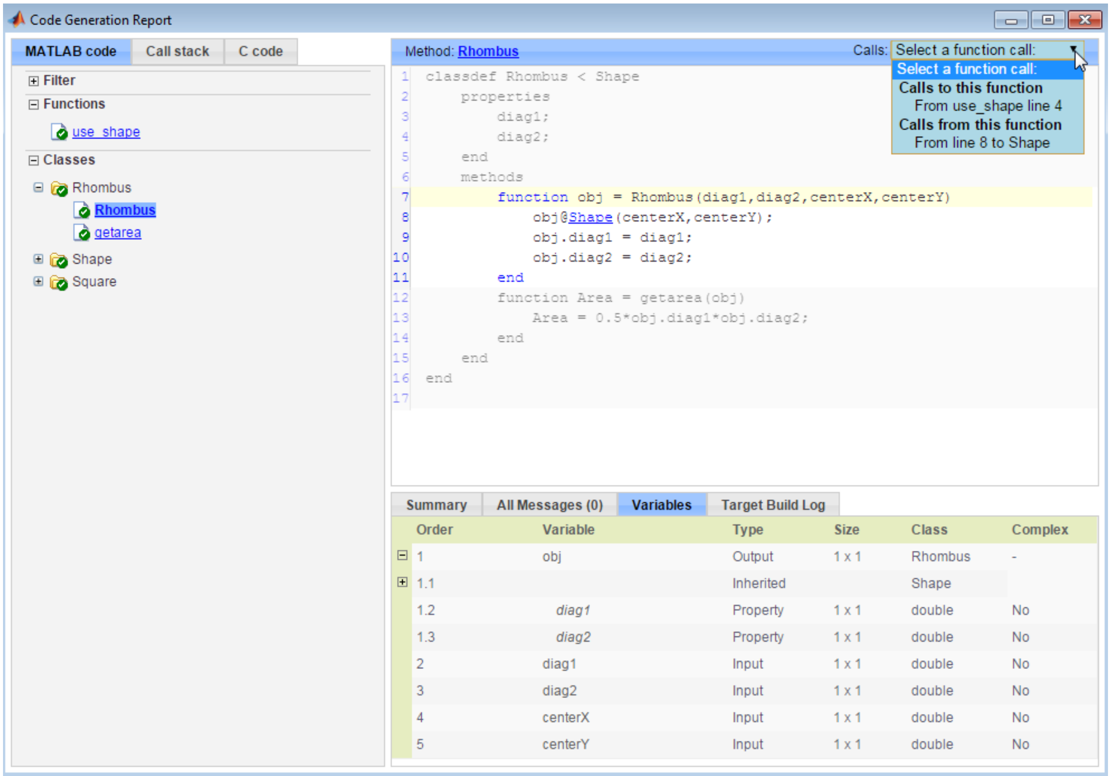


```
classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)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

	Order	Variable	Type	Size	Class	Complex
⊟	1	obj	Output	1 x 1	Rhombus	-
⊞	1.1		Inherited		Shape	
	1.2	*diag1*	Property	1 x 1	double	No
	1.3	*diag2*	Property	1 x 1	double	No
	2	diag1	Input	1 x 1	double	No
	3	diag2	Input	1 x 1	double	No
	4	centerX	Input	1 x 1	double	No
	5	centerY	Input	1 x 1	double	No

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

The **Calls** list shows that there is a call to the `Rhombus` constructor from `use_shape` and that this constructor calls the `Shape` constructor.

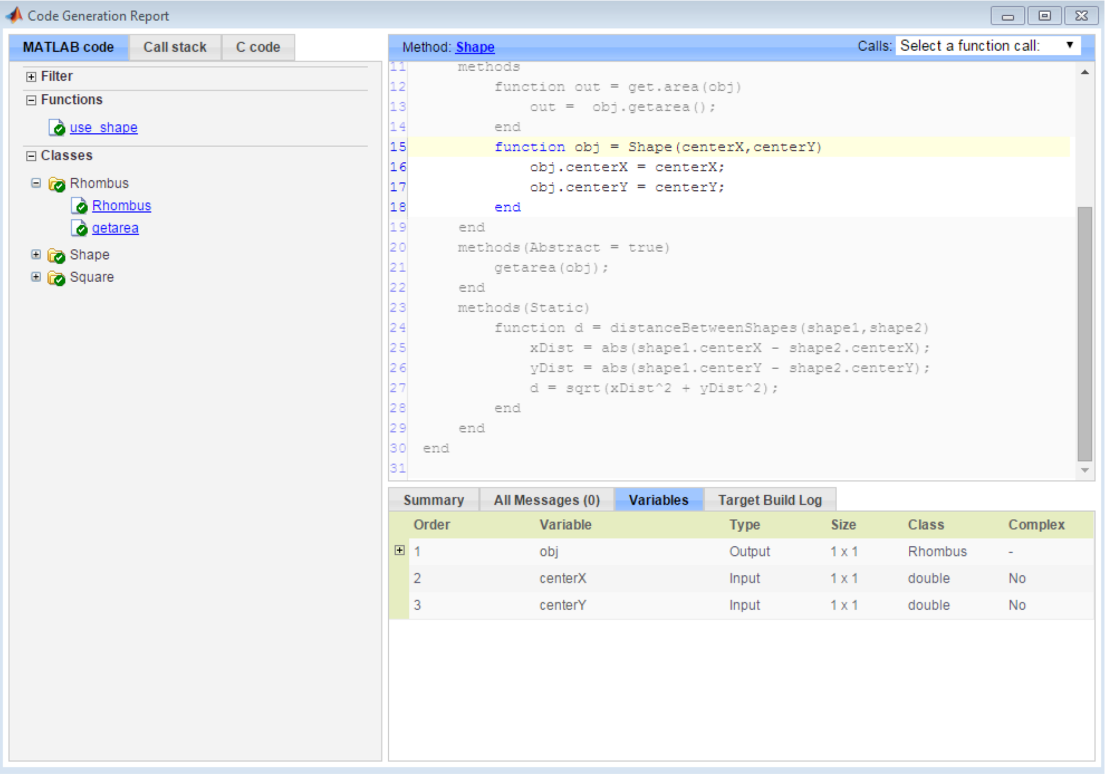


```
classdef Rhombus < Shape
    properties
        diag1;
        diag2;
    end
    methods
        function obj = Rhombus(diag1,diag2,centerX,centerY)
            obj@Shape(centerX,centerY);
            obj.diag1 = diag1;
            obj.diag2 = diag2;
        end
        function Area = getarea(obj)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

Order	Variable	Type	Size	Class	Complex
1	obj	Output	1 x 1	Rhombus	-
1.1		Inherited		Shape	
1.2	*diag1*	Property	1 x 1	double	No
1.3	*diag2*	Property	1 x 1	double	No
2	diag1	Input	1 x 1	double	No
3	diag2	Input	1 x 1	double	No
4	centerX	Input	1 x 1	double	No
5	centerY	Input	1 x 1	double	No

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

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



```
    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
```

	Order	Variable	Type	Size	Class	Complex
⊞	1	obj	Output	1 x 1	Rhombus	-
	2	centerX	Input	1 x 1	double	No
	3	centerY	Input	1 x 1	double	No

Generate Code for MATLAB Handle Classes and System Objects

This example shows how to generate code for a user-defined System object and then view the generated code in the code generation report.

1. In a writable folder, create a System object, `AddOne`, which subclasses from `matlab.System`. Save the code as `AddOne.m`.

   ```
   classdef AddOne < matlab.System
   % ADDONE Compute an output value that increments the input by one

     methods (Access=protected)
       % stepImpl method is called by the step method
       function y = stepImpl(~,x)
         y = x+1;
       end
     end
   end
   ```

2. Write a function that uses this System object.

   ```
   function y = testAddOne(x)
   %#codegen
     p = AddOne();
     y = p.step(x);
   end
   ```

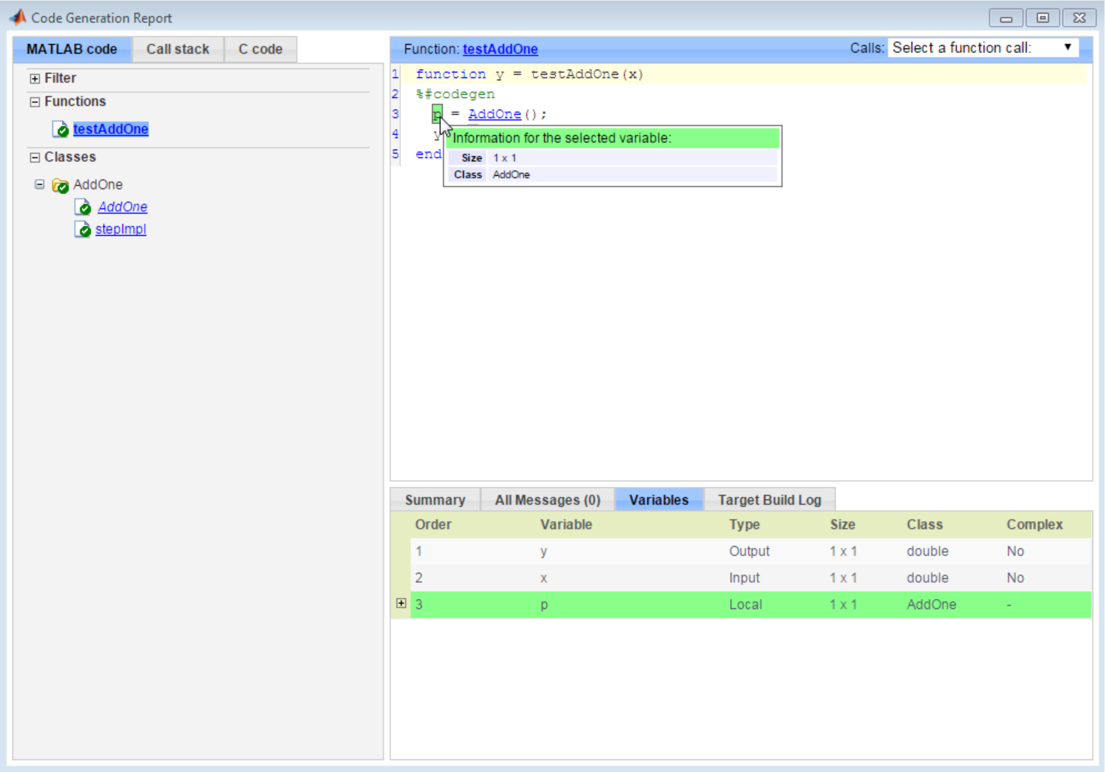
3. Generate a MEX function for this code.

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

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

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.



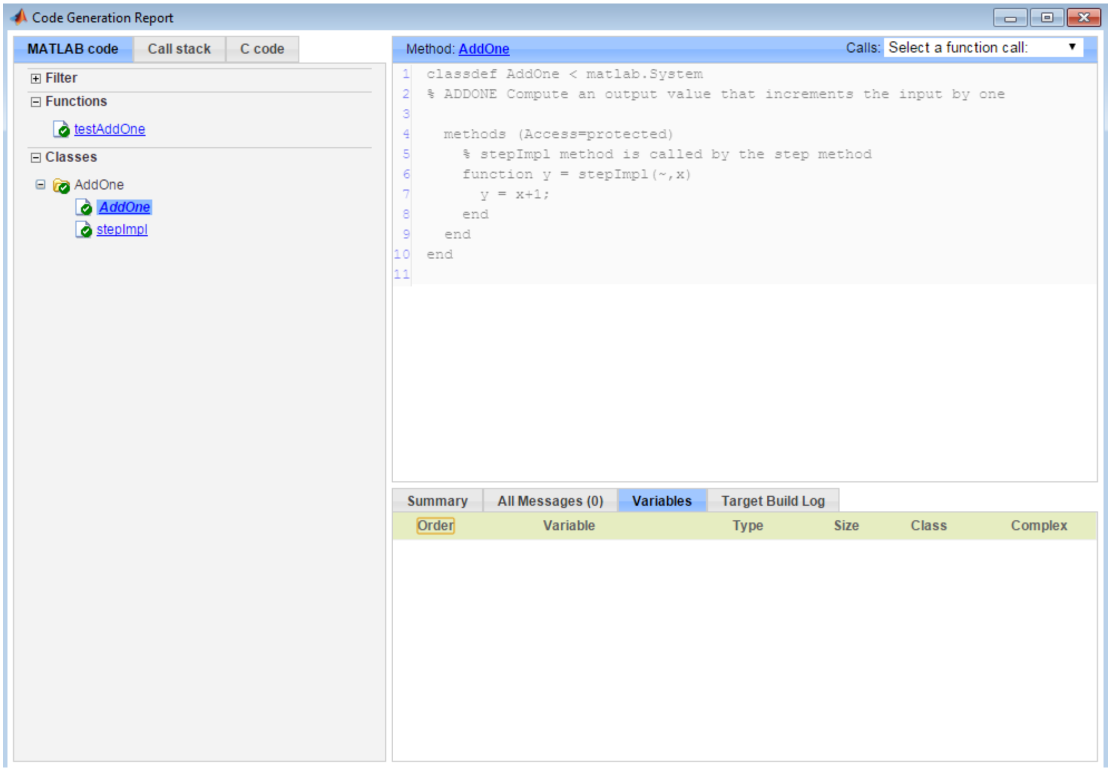
```
function y = testAddOne(x)
%#codegen
p = AddOne();
y
end
```

Information for the selected variable:

Size	1 x 1
Class	AddOne

Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	No
2	x	Input	1 x 1	double	No
3	p	Local	1 x 1	AddOne	-

6 To view the class definition, on the **Classes** panel, click `AddOne`.



```
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
```

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.

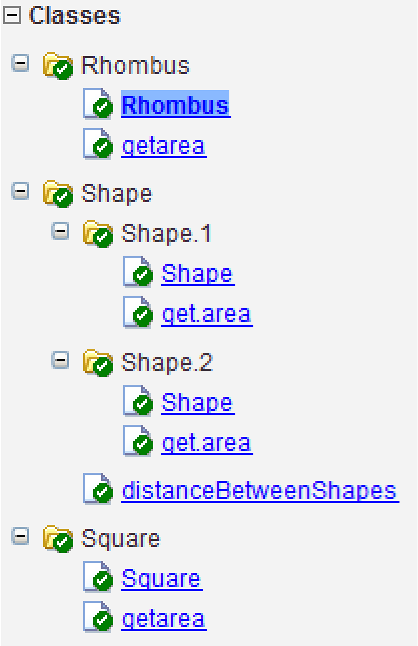
Default Constructors

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.



- ⊟ Classes
  - ⊟ Rhombus
    - Rhombus
    - getarea
  - ⊟ Shape
    - ⊟ Shape.1
      - Shape
      - get.area
    - ⊟ Shape.2
      - Shape
      - get.area
    - distanceBetweenShapes
  - ⊟ Square
    - Square
    - getarea

Packages

If you define classes as part of a package, the report displays the package in the list of classes. You can expand the package to view the classes that it contains. For more information about packages, see "Packages Create Namespaces" (MATLAB).

In the Variables Tab

The report displays the objects in the selected function or class. By default, for classes that have properties, the list of properties is collapsed. To expand the list, click the + symbol next to the object name. Within the list of properties, the list of inherited properties is collapsed. To expand the list of inherited properties, click the + symbol next to Inherited.

The report displays the properties using just the base property name, not the fully qualified name. For example, if your code uses variable obj1 that is a MATLAB object with property `prop1`, then the report displays the property as `prop1` not `obj1.prop1`. When you sort the **Variables** column, the sort order is based on the fully qualified property name.

In the Call Stack

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

Troubleshooting Code Generation Issues with MATLAB Classes

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

h = MyClass;

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

In this function, `h.mymethod()` returns a handle object of type `MyClass2`. In MATLAB, the assignment `h.mymethod().aa = 12;` changes the property of that object. Code generation does not support this assignment.

Solution

Rewrite the code to return the object and then assign a value to a property of the object.

```
function foo

h = MyClass;

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

Handle Object Limitations for Code Generation

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

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

- "A Variable Outside a Loop Cannot Refer to a Handle Object Created Inside a Loop" on page 10-22
- "A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object" on page 10-22

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

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

```
classdef mycls < handle
   properties
       prop
   end
end
```

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

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

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

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

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

```
classdef mycls < handle
   properties
       prop
   end
end
```

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

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

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

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

```
function usehandle3(x)
```

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

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

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

doinit(myobj);

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

System Objects in MATLAB Code Generation

In this section...

You can generate C/C++ code in MATLAB from your system that contains System objects by using MATLAB Coder. You can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms.

Usage Rules and Limitations for System Objects for Generating Code

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

Object Construction and Initialization

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

Inputs and Outputs

- System objects accept a maximum of 1024 inputs. A maximum of eight dimensions per input is supported.
- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that support for variable-size is enabled. Code generation support for variable-size data also requires that variable-

size support is enabled. By default in MATLAB, support for variable-size data is enabled.

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

Properties

- In MATLAB System blocks, you cannot use variable-size for discrete state properties of System objects. Private properties can be variable-size.
- Objects cannot be used as default values for properties.
- You can only assign values to nontunable properties once, including the assignment in the constructor.
- Nontunable property values must be constant.
- If a tunable property has dependant data type properties, you can set tunable properties only at construction time or after the object is locked.
- For `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.

Global Variables

- Global variables are allowed in a System object, unless you are using that System object in Simulink via the MATLAB System block. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

```
f = coder.MEXConfig;
f.GlobalSyncMethod = 'NoSync'
```

Then, include `'-config f'` in your `codegen` command.

Methods

- Code generation support is available only for these System object methods:
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone` (for sources only)
  - `isLocked`
  - `release`
  - `reset`
  - `set` (for tunable properties)
  - `step`
- For System objects that you define, code generation support is available only for these methods:
  - getDiscreteStateImpl
  - getNumInputsImpl
  - getNumOutputsImpl
  - infoImpl
  - isDoneImpl
  - isInputDirectFeedthroughImpl
  - outputImpl
  - processTunedPropertiesImpl
  - releaseImpl — Code is not generated automatically for this method. To release an object, you must explicitly call the `release` method in your code.
  - resetImpl
  - setupImpl
  - stepImpl
  - updateImpl
  - validateInputsImpl

- validatePropertiesImpl

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Getting Started with MATLAB Coder” and “MATLAB Classes” for more information.

Note: Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects in the MATLAB Function Block

Using the MATLAB Function block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see “What Is a MATLAB Function Block?” (Simulink).

System Objects in the MATLAB System Block

Using the MATLAB System block, you can include in a Simulink model individual System objects that you create with a class definition file. The model can then generate embeddable code. For more information, see “MATLAB System Block” (Simulink).

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

Specify Objects as Inputs at the Command Line

If you generate code by using `codegen`, to specify the type of an input that is a value class object, you can provide an example object with the `-args` option.

1 Define the value class. For example, define a class `myRectangle`.

```
classdef myRectangle
    properties
        length;
        width;
    end
    methods
        function obj = myRectangle(l,w)
            if nargin > 0
                obj.length = l;
                obj.width = w;
            end
        end
        function area = calcarea(obj)
            area = obj.length * obj.width;
        end
    end
end
```

2 Define a function that takes an object of the value class as an input. For example:

```
function z = getarea(r)
%#codegen
z = calcarea(r);
end
```

3 Create an object of the class.

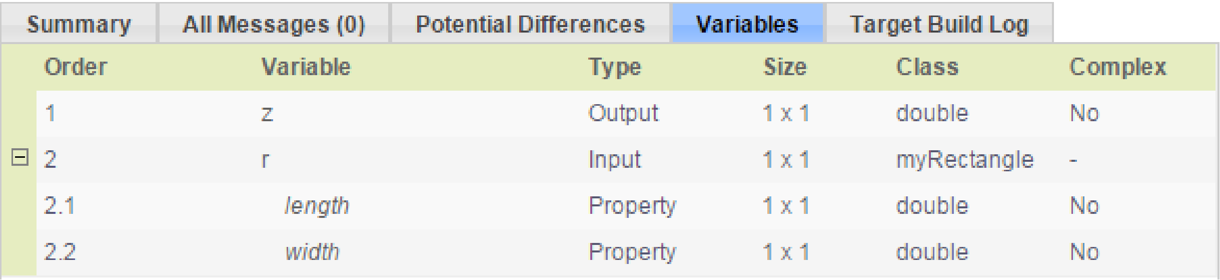
```
rect_obj = myRectangle(4,5)
```

```
rect_obj = 

  myRectangle with properties:

    length: 4
     width: 5
```

4 Pass the example object to `codegen` by using the `-args` option.

```
codegen getarea -args {rect_obj} -report
```

In the code generation report, you see that `r` has the same properties, `length` and `width`, as the example object `rect_object`. The properties have the same size and type as they do in the example object, `rect_object`.



	Order	Variable	Type	Size	Class	Complex
	1	z	Output	1 x 1	double	No
⊟	2	r	Input	1 x 1	myRectangle	-
	2.1	*length*	Property	1 x 1	double	No
	2.2	*width*	Property	1 x 1	double	No

Instead of providing an example object, you can create a type for an object of the value class, and then provide the type with the `-args` option.

1 Create an object of the class:

```
rect_obj = myRectangle(4,5)

rect_obj =

  myRectangle with properties:

    length: 4
     width: 5
```

2 To create a type for an object of `myRectangle` that has the same property types as `rect_obj`, use `coder.typeof`.

`coder.typeof` creates a `coder.ClassType` object that defines a type for a class.

```
t= coder.typeof(rect_obj)

t =

coder.ClassType
   1×1 myRectangle
      length: 1×1 double
      width : 1×1 double
```

3 Pass the type to `codegen` by using the `-args` option.

```
    codegen getarea -args {t} -report
```

After you create a type for a value class, you can change the types of the properties. For example, to make the properties of t 16-bit integers:

```
t.Properties.length = coder.typeof(int16(1))
t.Properties.width = coder.typeof(int16(1))
```

You can also add or delete properties. For example, to add a property newprop:

```
t.Properties.newprop = coder.typeof(int16(1))
```

Consistency Between coder.ClassType Object and Class Definition File

When you generate code, the properties of the coder.ClassType object that you pass to codegen must be consistent with the properties in the class definition file. If the class definition file has properties that your code does not use, the coder.ClassType object does not have to include those properties. The code generator removes properties that you do not use.

Limitations for Using Objects as Entry-Point Function Inputs

Entry-point function inputs that are objects have these limitations:

- An object that is an entry-point function input must be an object of a value class. Objects of handle classes cannot be entry-point function inputs. Therefore, a value class that contains a handle class cannot be an entry-point function input.
- coder.Constant does not support objects.
- An object cannot be an input or output argument for an extrinsic function.
- An object cannot be a global variable.

See Also

coder.ClassType

More About

- “Automatically Define Input Types by Using the App” on page 17-4

- “Define Input Parameter by Example by Using the App” on page 17-7
- “MATLAB Classes Definition for Code Generation” on page 10-2
- “Specify Objects as Inputs in the MATLAB Coder App” on page 10-33

Specify Objects as Inputs in the MATLAB Coder App

In the MATLAB Coder app, to specify the type of an input that is a value class object:

1. Define the value class. For example, define a class `myRectangle`.

   ```
   classdef myRectangle
       properties
           length;
           width;
       end
       methods
           function obj = myRectangle(l,w)
               if nargin > 0
                   obj.length = l;
                   obj.width = w;
               end
           end
           function area = calcarea(obj)
               area = obj.length * obj.width;
           end
       end
   end
   ```
2. Define a function that takes an object of the value class as an input. For example:

   ```
   function z = getarea(r)
   %#codegen
   z = calcarea(r);
   end
   ```
3. In the app, create a project for `getarea`. On the **Define Input Types** page, specify the type of the object in one of these ways:

   - Automatically define a value class input type.
   - Provide an Example Object.

Automatically Define an Object Input Type

- Write a test file `getarea_test` that creates an object of the `myRectangle` class and passes it to `getarea`. For example:

  ```
  rect_obj = myRectangle(4,5);
  rect_area = getarea(rect_obj);
  ```

```
disp(ar);
```

- In the app, on the **Define Input Types** page, specify the test file`getarea_test`.
- Click **Autodefine Input Types**.

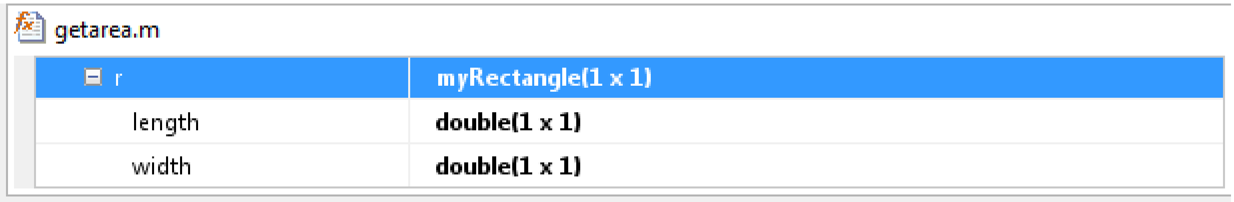
Provide an Example

If you provide an object of the value class, the app uses the sizes and types of the properties of the example object.

1. In MATLAB, define an object of the value class `myRectangle`.

   ```
   rect_obj = myRectangle(4,5)
   ```
2. In the app, on the **Define Input Types** page, click **Let me enter input or global types directly**.
3. Click the field to the right of the input parameter `r`.
4. Select **Define by Example**.
5. Enter `rect_obj` or select it from the list of workspace variables.

   The app determines the properties and their sizes and types from the example object.



getarea.m

⊟ r	**myRectangle(1 x 1)**
length	**double(1 x 1)**
width	**double(1 x 1)**

Alternatively, you can provide the name of the value class, `myRectangle`, or a `coder.ClassType` object for that class. To define a `coder.ClassType` object, use `coder.typeof`. For example:

1. In MATLAB, define a `coder.ClassType` object that has the same properties as `rect_obj`.

   ```
   t = coder.typeof(rect_obj)
   ```
2. In the app, provide `t` as the example.

To change the size or type of a property, click the field to the right of the property.

Consistency Between the Type Definition and Class Definition File

When you generate code, the properties that you define in the app must be consistent with the properties in the class definition file. If the class definition file has properties that your code does not use, your type definition in the app does not have to include those properties. The code generator removes properties that your code does not use.

Limitations for Using Objects as Entry-Point Function Inputs

Entry-point function inputs that are objects have these limitations:

- An object that is an entry-point function input must be an object of a value class. Objects of handle classes cannot be entry-point function inputs. Therefore, a value class that contains a handle class cannot be an entry-point function input.
- `coder.Constant` does not support objects.
- An object cannot be an input or output argument for an extrinsic function.
- An object cannot be a global variable.

See Also

coder.ClassType

More About

- “Automatically Define Input Types by Using the App” on page 17-4
- “Define Input Parameter by Example by Using the App” on page 17-7
- “Specify Objects as Inputs at the Command Line” on page 10-29
- “MATLAB Classes Definition for Code Generation” on page 10-2

11

Code Generation for Function Handles

Function Handle Limitations for Code Generation

When you use function handles in MATLAB code intended for code generation, adhere to the following restrictions:

Do not use the same bound variable to reference different function handles

In some cases, using the same bound variable to reference different function handles causes a compile-time error. For example, this code does not compile:

```
function y = foo(p)
x = @plus;
if p
  x = @minus;
end
y = x(1, 2);
```

Do not pass function handles to or from `coder.ceval`

You cannot pass function handles as inputs to or outputs from `coder.ceval`. For example, suppose that `f` and `str.f` are function handles:

```
f = @sin;
str.x = pi;
str.f = f;
```

The following statements result in compilation errors:

```
coder.ceval('foo', @sin);
coder.ceval('foo', f);
coder.ceval('foo', str);
```

Do not associate a function handle with an extrinsic function

You cannot create a function handle that references an extrinsic MATLAB function.

Do not pass function handles to or from extrinsic functions

You cannot pass function handles to or from `feval` and other extrinsic MATLAB functions.

Do not pass function handles to or from entry-point functions

You cannot pass function handles as inputs to or outputs from entry-point functions. For example, consider this function:

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle,'function_handle') && isa(data,'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

In this example, the function `plotFcn` receives a function handle and its data as inputs. `plotFcn` attempts to call the function referenced by the `fhandle` with the input `data` and plot the results. However, this code generates a compilation error. The error indicates that the function `isa` does not recognize `'function_handle'` as a class name when called inside a MATLAB function to specify properties of inputs.

More About

- "Declaring MATLAB Functions as Extrinsic Functions" on page 13-10

12

Defining Functions for Code Generation

Code Generation for Variable Length Argument Lists

When you use `varargin` and `varargout` for code generation, there are these restrictions:

- In the MATLAB Coder app, you cannot generate code for an entry-point function that uses `varargout`.
- You cannot use `varargin` or `varargout` in the function definition for a top-level function in a MATLAB Function block or in a Stateflow chart that uses MATLAB as the action language.
- If you use `varargin` to define an argument to an entry-point function, the code generator produces the function with a fixed number of arguments. This fixed number of arguments is based on the number of arguments that you specify when you generate code.
- You cannot write to `varargin`. If you want to write to input arguments, copy the values into a local variable.
- To index into `varargin` and `varargout`, use curly braces `{}`, not parentheses `()`.
- The code generator must be able to determine the value of the index into `varargin` or `varargout`.

More About

- "Nonconstant Index into vargin or vargout in a for-Loop" on page 30-16
- "Specify Number of Entry-Point Function Input or Output Arguments to Generate" on page 12-3

Specify Number of Entry-Point Function Input or Output Arguments to Generate

You can control the number of input or output arguments in a generated entry-point function. From one MATLAB function, you can generate entry-point functions that have different signatures.

Control Number of Input Arguments

If your entry-point function uses `varargin`, specify the properties for the arguments that you want in the generated function.

Consider this function:

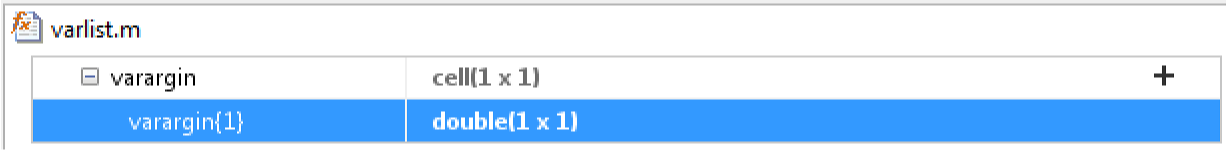
```
function [x, y] = myops(varargin)
%#codegen
if (nargin > 1)
    x = varargin{1} + varargin{2};
    y = varargin{1} * varargin{2};
else
    x = varargin{1};
    y = -varargin{1};
end
```

To generate a function that takes only one argument, provide one argument with `-args`.

```
codegen myops -args {3} -report
```

If you use the MATLAB Coder app:

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. In the variables table, click `varargin`.
3. Specify the properties for the arguments that you want in the generated function.



varlist.m

⊟ varargin	cell(1 x 1) +
varargin{1}	double(1 x 1)

If you generate code by using `codegen`, you can also control the number of input arguments when the MATLAB function does not use `varargin`.

Consider this function:

```
function [x, y] = myops(a,b)
%#codegen
if (nargin > 1)
    x = a + b;
    y = a * b;
else
    x = a;
    y = -a;
end
```

To generate a function that takes only one argument, provide one argument with `-args`.

```
codegen myops -args {3} -report
```

Control the Number of Output Arguments

If you generate code by using `codegen`, you can specify the number of output arguments by using the `-nargout` option.

Consider this function:

```
function [x, y] = myops(a,b)
%#codegen
if (nargin > 1)
    x = a + b;
    y = a * b;
else
    x = a;
    y = -a;
end
```

Generate a function that has one output argument.

```
codegen myops -args {2 3} -nargout 1 -report
```

You can also use `-nargout` to specify the number of arguments for an entry-point function that uses `varargout`.

Rewrite `myops` to use `varargout`.

```
function varargout = myops(a,b)
%#codegen
if (nargin > 1)
    varargout{1} = a + b;
    varargout{2} = a * b;
else
    varargout{1} = a;
    varargout{2} = -a;
end
```

Generate code for one output argument.

```
codegen myops -args {2 3} -nargout 1 -report
```

More About

- “Code Generation for Variable Length Argument Lists” on page 12-2
- “Specify Properties of Entry-Point Function Inputs” on page 20-46

Code Generation for Anonymous Functions

You can use anonymous functions in MATLAB code intended for code generation. For example, you can generate code for the following MATLAB code that defines an anonymous function that finds the square of a number.

```
sqr = @(x) x.^2;
a = sqr(5);
```

Anonymous functions are useful for creating a function handle to pass to a MATLAB function that evaluates an expression over a range of values. For example, this MATLAB code uses an anonymous function to create the input to the `fzero` function:

```
b = 2;
c = 3.5;
x = fzero(@(x) x^3 + b*x + c,0);
```

Anonymous Function Limitations for Code Generation

Anonymous functions have the code generation limitations of value classes and cell arrays.

More About

- “MATLAB Classes Definition for Code Generation” on page 10-2
- “Cell Array Limitations for Code Generation” on page 8-10
- “Parameterizing Functions” (MATLAB)

Code Generation for Nested Functions

You can generate code for MATLAB functions that contain nested functions. For example, you can generate code for the function `parent_fun`, which contains the nested function `child_fun`.

```
function parent_fun
x = 5;
child_fun

    function child_fun
        x = x + 1;
    end

end
```

Nested Function Limitations for Code Generation

When you generate code for nested functions, you must adhere to the code generation restrictions for value classes, cell arrays, and handle classes. You must also adhere to these restrictions:

- If the parent function declares a persistent variable, it must assign the persistent variable before it calls a nested function that uses the persistent variable.
- A nested recursive function cannot refer to a variable that the parent function uses.
- If a nested function refers to a structure variable, you must define the structure by using `struct`.
- If a nested function uses a variable defined by the parent function, you cannot use `coder.varsize` with the variable in either the parent or the nested function.

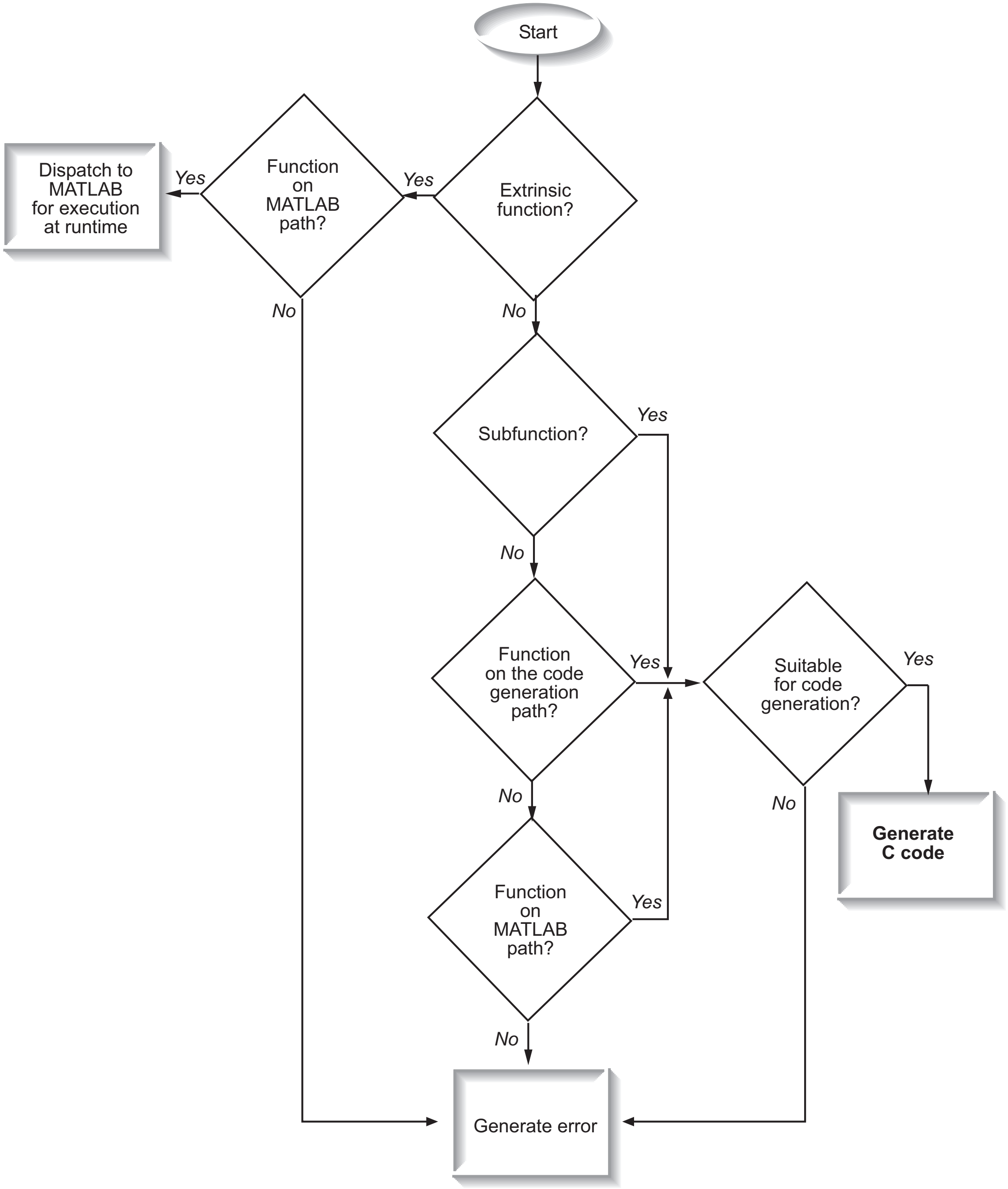
More About

- “MATLAB Classes Definition for Code Generation” on page 10-2
- “Handle Object Limitations for Code Generation” on page 10-22
- “Cell Array Limitations for Code Generation” on page 8-10
- “Code Generation for Recursive Functions” on page 13-18

Calling Functions for Code Generation

Resolution of Function Calls for Code Generation

From a MATLAB function, you can call local functions, supported toolbox functions, and other MATLAB functions. MATLAB resolves function names for code generation as follows:


Start
Extrinsic function?
Yes
Function on MATLAB path?
Yes
Dispatch to MATLAB for execution at runtime
No
No
Subfunction?
Yes
No
Function on the code generation path?
Yes
No
Suitable for code generation?
Yes
No
Generate C code
Function on MATLAB path?
Yes
No
Generate error

Key Points About Resolving Function Calls

The diagram illustrates key points about how MATLAB resolves function calls for code generation:

- Searches two paths, the code generation path and the MATLAB path

  See "Compile Path Search Order" on page 13-4.
- Attempts to compile functions unless the code generator determines that it should not compile them or you explicitly declare them to be extrinsic.

  If a MATLAB function is not supported for code generation, you can declare it to be extrinsic by using the construct `coder.extrinsic`, as described in "Declaring MATLAB Functions as Extrinsic Functions" on page 13-10. During simulation, the code generator produces code for the call to an extrinsic function, but does not generate the internal code for the function. Therefore, simulation can run only on platforms where MATLAB software is installed. During standalone code generation, the code generator attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that the output does not change, code generation proceeds, but the extrinsic function is excluded from the generated code. Otherwise, compilation errors occur.

  The code generator detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using the `coder.extrinsic` function.
- Resolves file type based on precedence rules described in "Resolution of File Types on Code Generation Path" on page 13-6

Compile Path Search Order

During code generation, function calls are resolved on two paths:

1. Code generation path

   MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.
2. MATLAB path

   If the function is not on the code generation path, MATLAB searches this path.

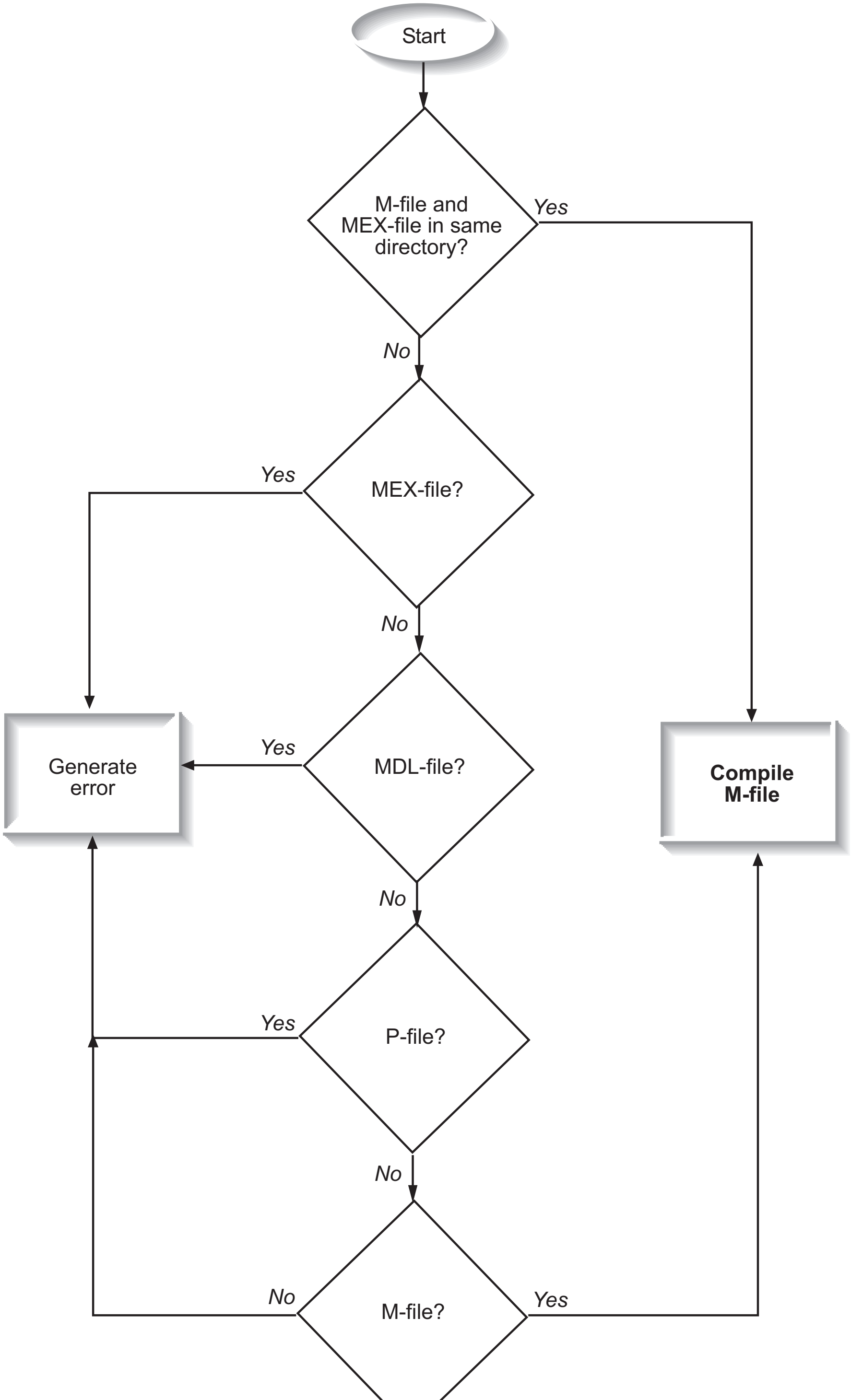
MATLAB applies the same dispatcher rules when searching each path (see "Function Precedence Order" (MATLAB)).

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

Resolution of File Types on Code Generation Path

MATLAB uses the following precedence rules for code generation:


Start
M-file and MEX-file in same directory?
Yes
No
MEX-file?
Yes
No
Generate error
MDL-file?
Yes
No
Compile M-file
P-file?
Yes
No
M-file?
No
Yes

Compilation Directive %#codegen

Add the `%#codegen` directive (or pragma) to your function after the function signature to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would result in errors during code generation.

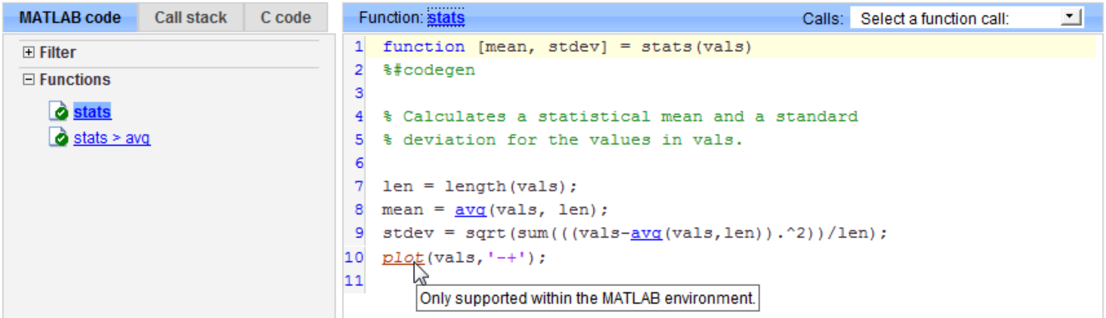
```
function y = my_fcn(x) %#codegen

....
```

Note: The `%#codegen` directive is not necessary for MATLAB Function blocks. Code inside a MATLAB Function block is always intended for code generation. The `%#codegen` directive, or the absence of it, does not change the error checking behavior.

Extrinsic Functions

The code generator attempts to generate code for functions, even if they are not supported for C code generation. The software detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. The software treats these functions like extrinsic functions but you do not have to declare them extrinsic using `coder.extrinsic`. During simulation, the code generator produces code for these functions, but does not generate their internal code. During standalone code generation, the code generator attempts to determine whether the visualization function affects the output of the function in which it is called. Provided that the output does not change, the code generator proceeds with code generation, but excludes the visualization function from the generated code. Otherwise, compilation errors occur.

For example, you might want to call `plot` to visualize your results in the MATLAB environment. If you generate a MEX function from a function that calls `plot` and then run the generated MEX function, the code generator dispatches calls to the `plot` function to MATLAB. If you generate a library or executable, the generated code does not contain calls to the `plot` function. The code generation report highlights calls from your MATLAB code to extrinsic functions so that it is easy to determine which functions are supported only in the MATLAB environment.



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

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

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

```

Only supported within the MATLAB environment.

For unsupported functions other than common visualization functions, you must declare the functions to be extrinsic (see “Resolution of Function Calls for Code Generation” on page 13-2). Extrinsic functions are not compiled, but instead executed in MATLAB during simulation (see “Resolution of Extrinsic Functions During Simulation” on page 13-14).

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

- Use the `coder.extrinsic` construct in main functions or local functions (see “Declaring MATLAB Functions as Extrinsic Functions” on page 13-10).
- Call the function indirectly using `feval` (see “Calling MATLAB Functions Using feval” on page 13-14).

Declaring MATLAB Functions as Extrinsic Functions

To declare a MATLAB function to be extrinsic, add the `coder.extrinsic` construct at the top of the main function or a local function:

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

Declaring Extrinsic Functions

The following code declares the MATLAB `patch` function extrinsic in the local function `create_plot`. You do not have to declare `axis` as extrinsic because `axis` is one of the common visualization functions that the code generator automatically treats as extrinsic.

```
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 as extrinsic

coder.extrinsic('patch');

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

The code generator does not produce code for `patch` and `axis`, but instead dispatches them to MATLAB for execution.

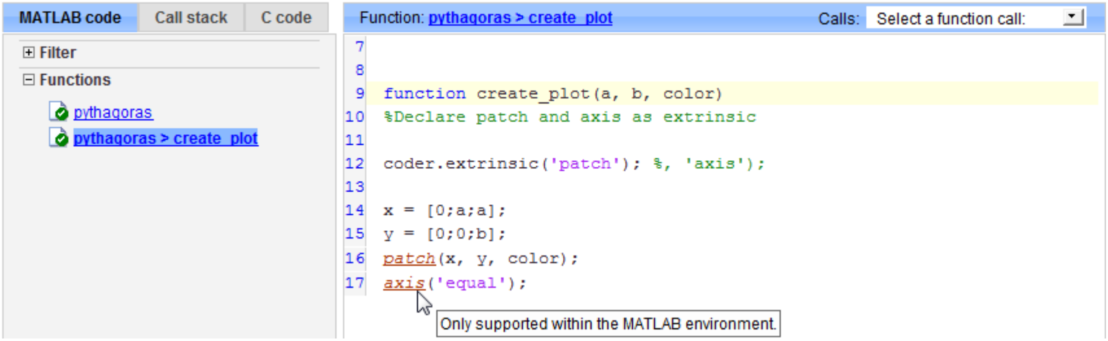
To test the function, follow these steps:

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

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

2 Click the link to the code generation report and then, in the report, view the MATLAB code for `create_plot`.

The report highlights the `patch` and `axis` functions to indicate that they are supported only within the MATLAB environment.



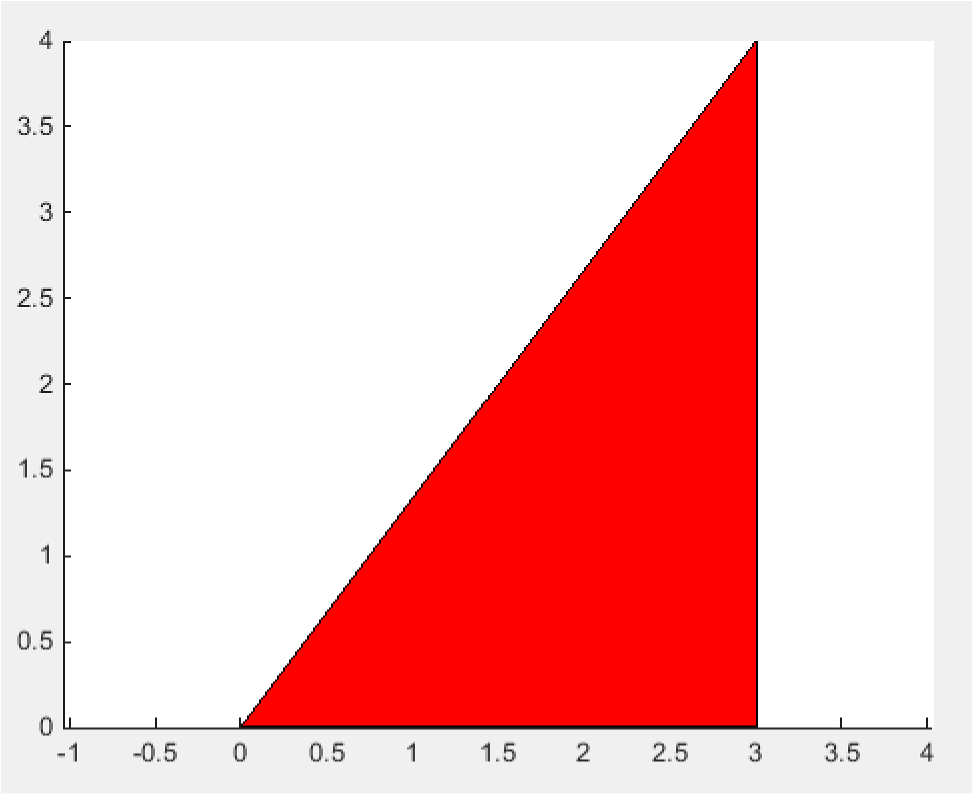
```


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');
```

Only supported within the MATLAB environment.

3 Run the MEX function by executing this command:

```
pythagoras_mex(3, 4, [1.0 0.0 0.0]);
```

MATLAB displays a plot of the right triangle as a red patch object:



When to Use the coder.extrinsic Construct

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output during simulation, without generating unnecessary code (see “Resolution of Extrinsic Functions During Simulation” on page 13-14).
- Make your code self-documenting and easier to debug. You can scan the source code for `coder.extrinsic` statements to isolate calls to MATLAB functions, which can potentially create and propagate `mxArrays` (see “Working with mxArrays” on page 13-15).

- Save typing. With one `coder.extrinsic` statement, each subsequent function call is extrinsic, as long as the call and the statement are in the same scope (see “Scope of Extrinsic Function Declarations” on page 13-13).
- Declare the MATLAB function(s) extrinsic throughout the calling function scope (see “Scope of Extrinsic Function Declarations” on page 13-13). To narrow the scope, use `feval` (see “Calling MATLAB Functions Using feval” on page 13-14).

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` as treated as extrinsic every time they are called in the main function `foo`. There are two ways to narrow the scope of an extrinsic declaration inside the main function:

- Declare the MATLAB function extrinsic in a local function, as in this example:

  ```
  function y = foo %#codegen
  coder.extrinsic('rat');
  [N D] = rat(pi);
  y = 0;
  y = mymin(N, D);

  function y = mymin(a,b)
  coder.extrinsic('min');
  y = min(a,b);
  ```

  Here, the function `rat` is extrinsic every time it is called inside the main function `foo`, but the function `min` is extrinsic only when called inside the local function `mymin`.

- Call the MATLAB function using feval, as described in “Calling MATLAB Functions Using feval” on page 13-14.

Calling MATLAB Functions Using feval

The function feval is automatically interpreted as an extrinsic function during code generation. Therefore, you can use feval to conveniently call functions that you want to execute in the MATLAB environment, rather than compiled to generated code.

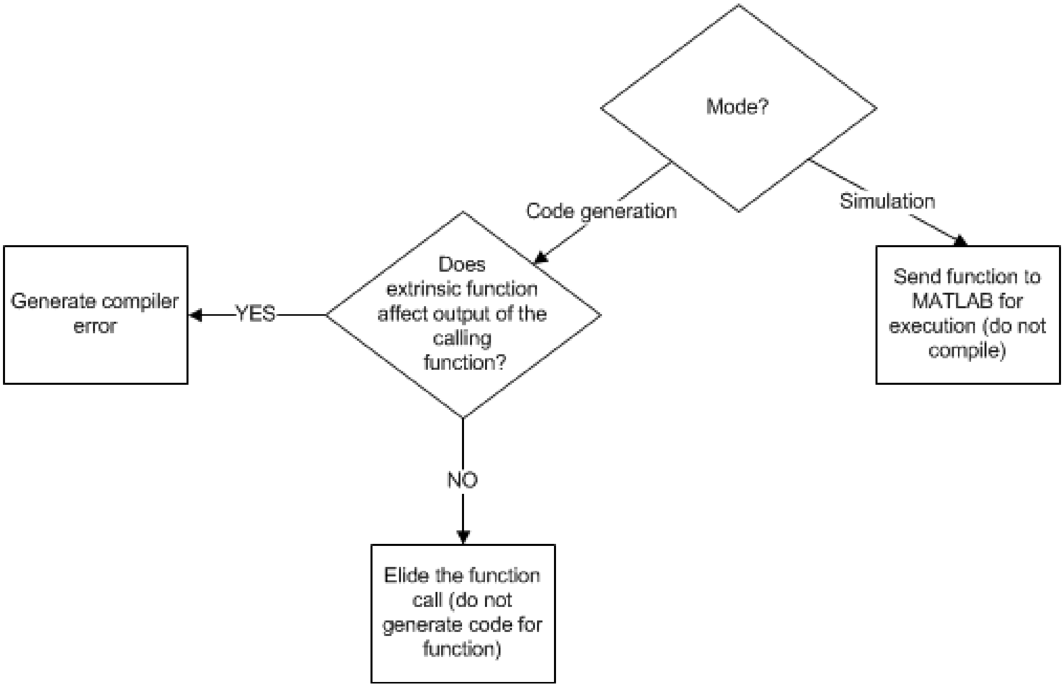
Consider the following example:

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min', N, D);
```

Because feval is extrinsic, the statement feval('min', N, D) is evaluated by MATLAB — not compiled — which has the same result as declaring the function min extrinsic for just this one call. By contrast, the function rat is extrinsic throughout the function foo.

Resolution of Extrinsic Functions During Simulation

The code generator resolves calls to extrinsic functions — functions that do not support code generation — as follows:


Mode?
Code generation
Simulation
Does extrinsic function affect output of the calling function?
YES
Generate compiler error
Send function to MATLAB for execution (do not compile)
NO
Elide the function call (do not generate code for function)

During simulation, the code generator produces code for the call to an extrinsic function, but does not generate the internal code for the function. Therefore, you can run the simulation only on platforms where you install MATLAB software.

During code generation, the code generator attempts to determine whether the extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable (see “Working with mxArrays” on page 13-15). Provided that the output does not change, code generation proceeds, but the extrinsic function is excluded from the generated code. Otherwise, the code generator issues a compiler error.

Working with mxArrays

The output of an extrinsic function is an `mxArray` — also called a MATLAB array. The only valid operations for `mxArrays` are:

- Storing `mxArrays` in variables

- Passing mxArrays to functions and returning them from functions
- Converting mxArrays to known types at run time

To use mxArrays returned by extrinsic functions in other operations, you must first convert them to known types, as described in “Converting mxArrays to Known Types” on page 13-16.

Converting mxArrays to Known Types

To convert an mxArray to a known type, assign the mxArray to a variable whose type is defined. At run time, the mxArray is converted to the type of the variable assigned to it. However, if the data in the mxArray is not consistent with the type of the variable, you get a run-time error.

For example, consider this code:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = min(N, D);
```

Here, the top-level function foo calls the extrinsic MATLAB function rat, which returns two mxArrays representing the numerator N and denominator D of the rational fraction approximation of pi. Although you can pass these mxArrays to another MATLAB function — in this case, min — you cannot assign the mxArray returned by min to the output y.

If you run this function foo in a MATLAB Function block in a Simulink model, the code generates the following error during simulation:

```
Function output 'y' cannot be of MATLAB type.
```

To fix this problem, define y to be the type and size of the value that you expect min to return — in this case, a scalar double — as follows:

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```

Restrictions on Extrinsic Functions for Code Generation

The full MATLAB run-time environment is not supported during code generation. Therefore, the following restrictions apply when calling MATLAB functions extrinsically:

- MATLAB functions that inspect the caller, or read or write to the caller workspace do not work during code generation. Such functions include:
  - `dbstack`
  - `evalin`
  - `assignin`
  - `save`
- The MATLAB debugger cannot inspect variables defined in extrinsic functions.
- Functions in generated code can produce unpredictable results if your extrinsic function performs the following actions at run time:
  - Change folders
  - Change the MATLAB path
  - Delete or add MATLAB files
  - Change warning states
  - Change MATLAB preferences
  - Change Simulink parameters

Limit on Function Arguments

You can call functions with up to 64 inputs and 64 outputs.

Code Generation for Recursive Functions

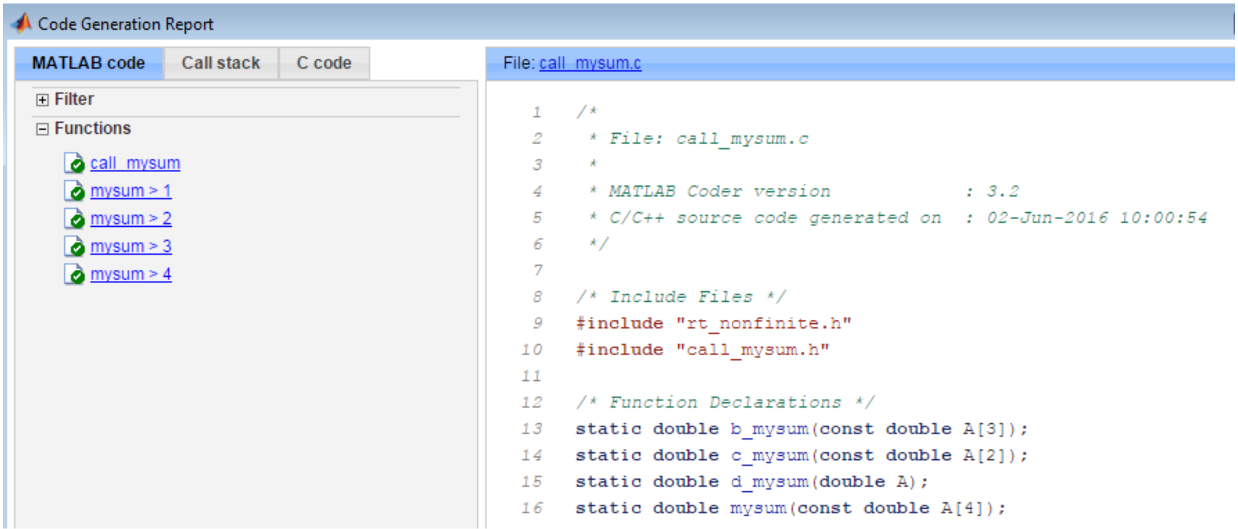
To generate code for recursive MATLAB functions, the code generator uses compile-time recursion or run-time recursion. You can influence whether the code generator uses compile-time or run-time recursion by modifying your MATLAB code. See “Force Code Generator to Use Run-Time Recursion” on page 13-22.

You can disallow recursion or disable run-time recursion by modifying configuration parameters.

When you use recursive functions in MATLAB code that is intended for code generation, you must adhere to certain restrictions. See “Recursive Function Limitations for Code Generation” on page 13-21.

Compile-Time Recursion

With compile-time recursion, the code generator creates multiple versions of a recursive function in the generated code. The inputs to each version have values or sizes that are customized for that version. These versions are known as *function specializations*. You can tell that the code generator used compile-time recursion by looking at the code generation report or the generated C code. Here is an example of compile-time recursion in the report.



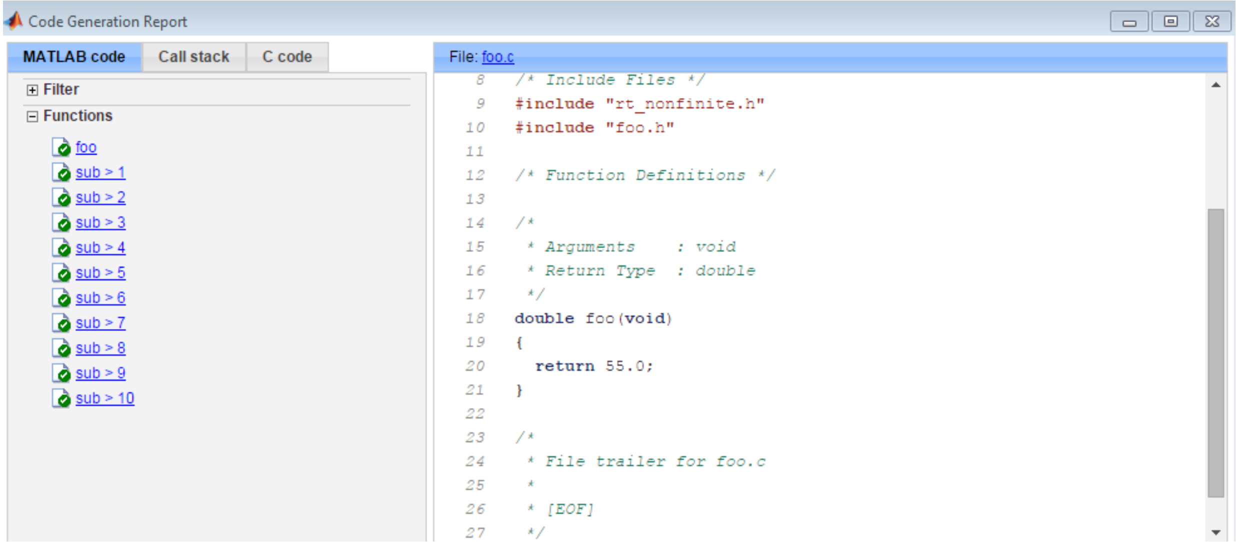
File: call_mysum.c

```
/*
 * File: call_mysum.c
 *
 * MATLAB Coder version            : 3.2
 * C/C++ source code generated on  : 02-Jun-2016 10:00:54
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "call_mysum.h"

/* Function Declarations */
static double b_mysum(const double A[3]);
static double c_mysum(const double A[2]);
static double d_mysum(double A);
static double mysum(const double A[4]);
```

Sometimes, the function specializations do not appear in the C/C++ code because of optimizations. For example, consider this function:

```
function y = foo()
%#codegen
    x = 10;
    y = sub(x);
end

function y = sub(x)
coder.inline('never');
if x > 1
    y = x + sub(x-1);
else
    y = x;
end
end
```

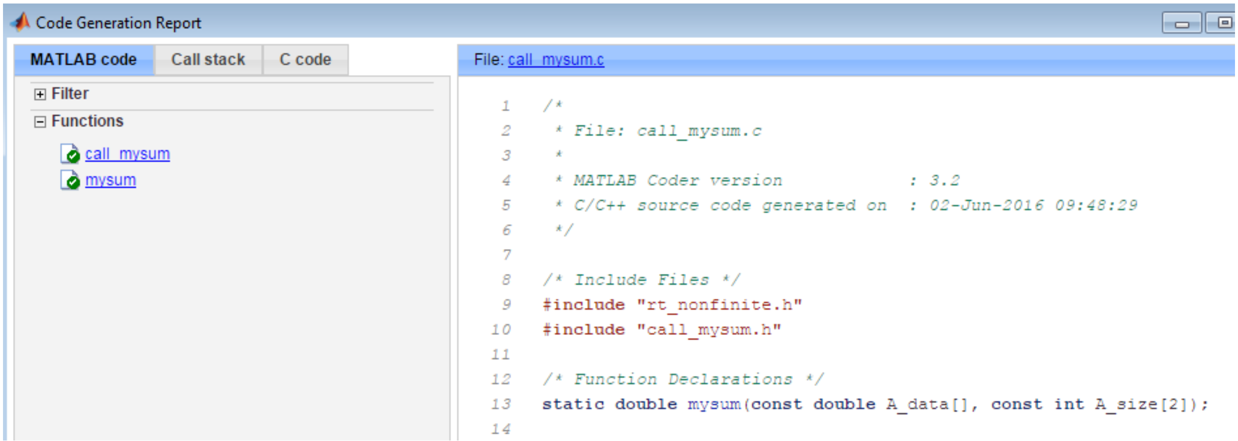
In the code generation report, on the **MATLAB code** tab, you see the function specializations for MATLAB function sub. However, the C code does not contain the specializations. It contains one function that returns the value 55.



```
/* Include Files */
#include "rt_nonfinite.h"
#include "foo.h"

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : double
 */
double foo(void)
{
  return 55.0;
}

/*
 * File trailer for foo.c
 *
 * [EOF]
 */
```

Run-Time Recursion

With run-time recursion, the code generator produces a recursive function in the generated code. You can tell that the code generator used run-time recursion by looking at the code generation report or the generated C code. Here is an example of run-time recursion in the report.



File: call_mysum.c

```
/*
 * File: call_mysum.c
 *
 * MATLAB Coder version            : 3.2
 * C/C++ source code generated on  : 02-Jun-2016 09:48:29
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "call_mysum.h"

/* Function Declarations */
static double mysum(const double A_data[], const int A_size[2]);

```

Disallow Recursion

- In a code generation configuration object, set the `CompileTimeRecursionLimit` configuration parameter to 0.
- In the MATLAB Coder app, set the value of the **Compile-time recursion limit** setting to 0.

Disable Run-Time Recursion

Some coding standards, such as MISRA®, do not allow recursion. To increase the likelihood of generating code that is compliant with MISRA C® , disable run-time recursion.

- In a code generation configuration object, set `EnableRuntimeRecursion` to `false`.
- In the MATLAB Coder app, set **Enable run-time recursion** to `No`.

If your code requires run-time recursion and run-time recursion is disabled, you must rewrite your code so that it uses compile-time recursion or does not use recursion.

Recursive Function Limitations for Code Generation

When you use recursion in MATLAB code that is intended for code generation, follow these restrictions:

- Assign all outputs of a run-time recursive function before the first recursive call in the function.
- Assign all elements of cell array outputs of a run-time recursive function.
- Inputs and outputs of run-time recursive functions cannot be classes.
- The maximum stack usage on page 28-17 setting is ignored for run-time recursion.

More About

- "Force Code Generator to Use Run-Time Recursion" on page 13-22
- "Output Variable Must Be Assigned Before Run-Time Recursive Call" on page 30-4
- "Compile-Time Recursion Limit Reached" on page 30-7
- "Configure Build Settings" on page 20-26
- "Code Generation Reports" on page 21-9

Force Code Generator to Use Run-Time Recursion

When your MATLAB code includes recursive function calls, the code generator uses compile-time or run-time recursion. With compile-time recursion, the code generator creates multiple versions of the recursive function in the generated code. These versions are known as function specializations. With run-time recursion, the code generator produces a recursive function. If compile-time recursion results in too many function specializations or if you prefer run-time recursion, you can try to force the code generator to use run-time recursion. Try one of these approaches:

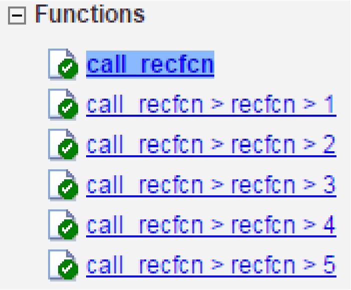
- "Treat the Input to the Recursive Function as a Nonconstant" on page 13-22
- "Make the Input to the Recursive Function Variable-Size" on page 13-23
- "Assign Output Variable Before the Recursive Call" on page 13-24

Treat the Input to the Recursive Function as a Nonconstant

Consider this function:

```
function y = call_recfcn(n)
A = ones(1,n);
x = 5;
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

`call_recfcn` calls `recfcn` with the value 5 for the second argument. `recfcn` calls itself recursively until `x` is 1. For each `recfcn` call, the input argument `x` has a different value. The code generator produces five specializations of `recfcn`, one for each call. After you generate code, you can see the specializations in the code generation report.



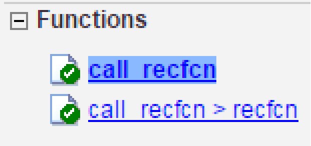
⊟ Functions

- call_recfcn
- call_recfcn > recfcn > 1
- call_recfcn > recfcn > 2
- call_recfcn > recfcn > 3
- call_recfcn > recfcn > 4
- call_recfcn > recfcn > 5

To force run-time recursion, in `call_recfcn`, in the call to `recfcn`, instruct the code generator to treat the value of the input argument `x` as a nonconstant value by using `coder.ignoreConst`.

```
function y = call_recfcn(n)
A = ones(1,n);
x = coder.ignoreConst(5);
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

After you generate code, in the code generation report., you see only one specialization.



⊟ Functions

- call_recfcn
- call_recfcn > recfcn

Make the Input to the Recursive Function Variable-Size

Consider this code:

```
function z = call_mysum(A)
%#codegen
```

```
z = mysum(A);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

If the input to `mysum` is fixed-size, the code generator uses compile-time recursion. To force the code generator to use run-time conversion, make the input to `mysum` variable-size by using `coder.varsize`.

```
function z = call_mysum(A)
%#codegen
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Assign Output Variable Before the Recursive Call

The code generator uses compile-time recursion for this code:

```
function y = callrecursive(n)
x = 10;
y = myrecursive(x,n);
end

function y = myrecursive(x,n)
coder.inline('never')
if x > 1
```

```
    y = n + myrecursive(x-1,n-1);

else
    y = n;
end
end
```

To force the code generator to use run-time recursion, modify `myrecursive` so that the output `y` is assigned before the recursive call. Place the assignment `y = n` in the `if` block and the recursive call in the `else` block.

```
function y = callrecursive(n)
x = 10;
y = myrecursive(x,n);
end

function y = myrecursive(x,n)
coder.inline('never')
if x == 1
    y = n;
else
    y = n + myrecursive(x-1,n-1);
end
end
```

See Also

`coder.ignoreConst`

More About

- “Code Generation for Recursive Functions” on page 13-18
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 30-4
- “Compile-Time Recursion Limit Reached” on page 30-7

Fixed-Point Conversion

Detect Dead and Constant-Folded Code

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

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

Coverage Bar Color	Indicates
Green	One of the following situations: • The entry-point function executes multiple times and the code executes more than one time. • The entry-point function executes one time and the code executes one time. Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

What Is Dead Code?

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

- Defensive code containing intended corner cases that are not reached
- Human error in the code, resulting in code that cannot be reached by any execution path
- Inadequate test bench range

- Constant folding

Detect Dead Code

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

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

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

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

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

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

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

myFunction(u,v);
```

3 From the apps gallery, open the MATLAB Coder app.

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

5 On the **Select Source Files** page, browse to the `myFunction` file, and click **Open**.

6 Click **Next**. On the **Define Input Types** page, browse to select the test file that you created, `myFunction_tb`. Click **Autodefine Input Types**.

7 Click **Next**. On the **Check for Run-Time Issues** page, click **Check for Issues**.

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

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

The color-coded bar on the left side of the edit window indicates whether the code executes. The code in the first condition of the if-statement does not execute during simulation because *u* is never greater than *v*. The `bar` function never executes because the if-statement never executes. These parts of the algorithm are marked with a red bar, indicating that they are dead code.

9 To apply the proposed data types to the function, click **Convert** .

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

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

10 Close the MATLAB Coder app.

Fix Dead Code

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

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

myFunction(u,v);
```

2 Reopen the MATLAB Coder app.

3 Using the same function and the edited test file, go through the conversion process again.

4 After you click **Analyze**, this time the code coverage bar shows that all parts of the algorithm execute with the new test file input ranges.

To finish the conversion process and convert the function to fixed point, click **Convert**.

Convert MATLAB Code to Fixed-Point C Code

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

1. Open the MATLAB Coder app.
2. On the **Select Source Files** page, add the entry-point function from which you want to generate code.
3. Set **Numeric Conversion** to `Convert to fixed point`.
4. Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. If the app does not find issues, it opens the **Define Input Types** page.
5. On the **Define Input Types** page, specify a test file that the app can use to define the input types.
6. Click **Next** to go to the **Check for Run-Time Issues** step.
7. On the **Check for Run-Time Issues** page, specify a test file that calls your entry-point function. Alternatively, at the prompt, enter code that calls your entry-point function. The app generates instrumented MEX. It runs the test file or code that you specified, replacing calls to your entry-point function with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. Click a message to highlight the problematic code in a window where you can edit the code.
8. Click **Next** to go to the **Convert to Fixed Point** step.
9. Propose data types based on simulation range data, derived (also known as static) range data, or both. See “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-7 and “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-21.
10. To convert the floating-point MATLAB code to fixed-point MATLAB code, click **Convert**. During fixed-point conversion, the app validates the build using the proposed fixed-point data types. See “Validating Types” on page 14-102.
11. Verify the behavior of the fixed-point MATLAB code. See “Testing Numerics” on page 14-102.
12. Click **Next** to go to the **Generate Code** step.
13. In the **Generate** dialog box, set **Build source** to `Fixed-Point`. Set the **Build type** to build a static or dynamic library, or executable. Set **Language** to C. Click **Generate**.

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

Related Examples

- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-7
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-21

Propose Fixed-Point Data Types Based on Simulation Ranges

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

Prerequisites

This example requires the following products:

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

  You can use `mex -setup` to change the default compiler. See "Change Default Compiler" (MATLAB).

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\ex_2ndOrder_filter`.
2. Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

Type	Name	Description
Function code	`ex_2ndOrder_filter.m`	Entry-point MATLAB function
Test file	`ex_2ndOrder_filter_test.m`	MATLAB script that tests `ex_2ndOrder_filter.m`

The ex_2ndOrder_filter Function

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
```

```
    end
    % [b,a] = butter(2, 0.25)
    b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
    a = [1, -0.942809041582063,  0.3333333333333333];


    y = zeros(size(x));
    for i = 1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

The ex_2ndOrder_filter_test Script

The test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```
% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256;                   % Number of points
t = linspace(0,1,N);       % Time vector from 0 to 1 second
f1 = N/2;                  % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N);        % Step
x_impulse = zeros(1,N);    % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
  y(i,:) = ex_2ndOrder_filter(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'}
clf
for i = 1:size(x,1)
  subplot(size(x,1),1,i)
  plot(t,x(i,:),t,y(i,:))
  title(titles{i})
  legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Open the MATLAB Coder App

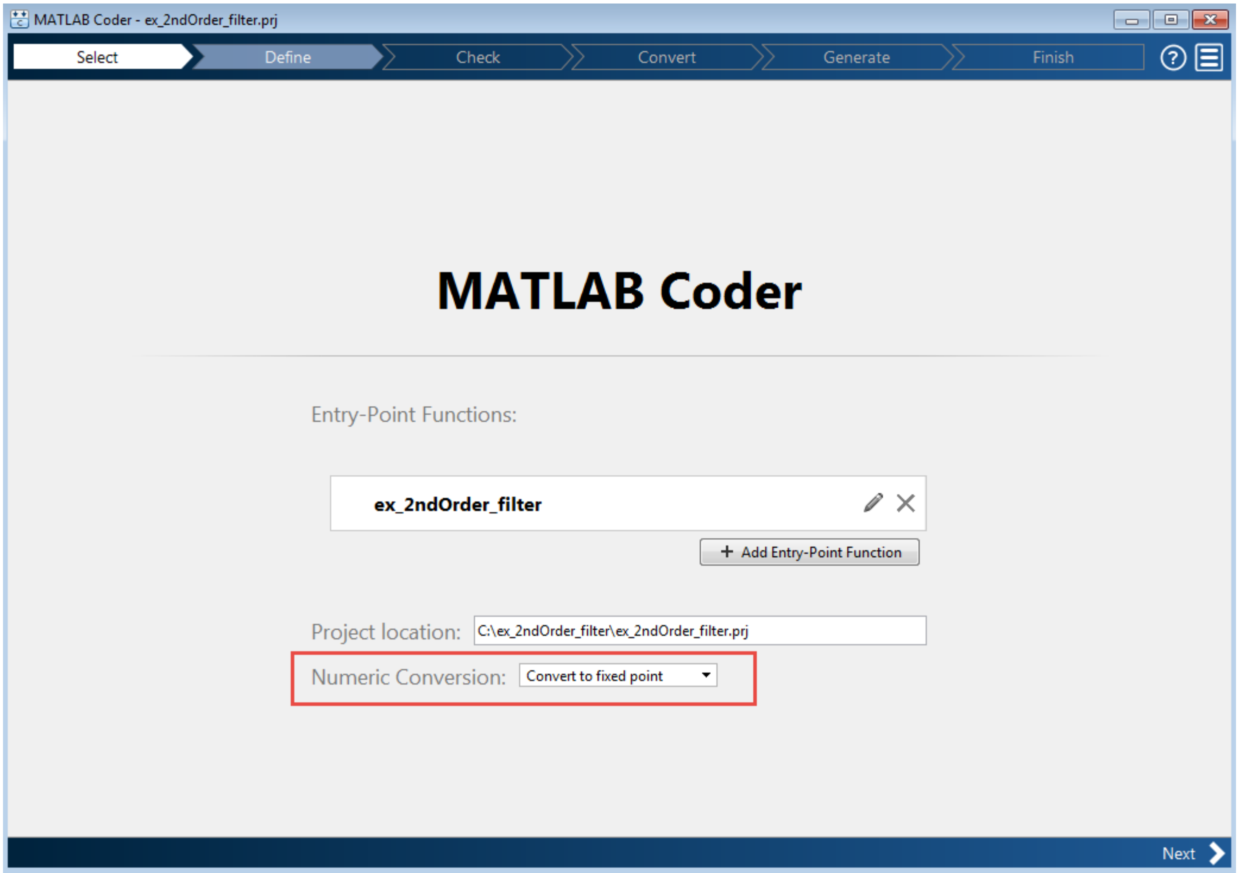
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Select Source Files

To add the entry-point function `ex_2ndOrder_filter` to the project, browse to the file `ex_2ndOrder_filter.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `ex_2ndOrder_filter.prj`.

Enable Fixed-Point Conversion

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



Entry-Point Functions:

ex_2ndOrder_filter

+ Add Entry-Point Function

Project location: C:\ex_2ndOrder_filter\ex_2ndOrder_filter.prj

Numeric Conversion: Convert to fixed point

2 Click **Next** to go to the **Define Input Types** step.

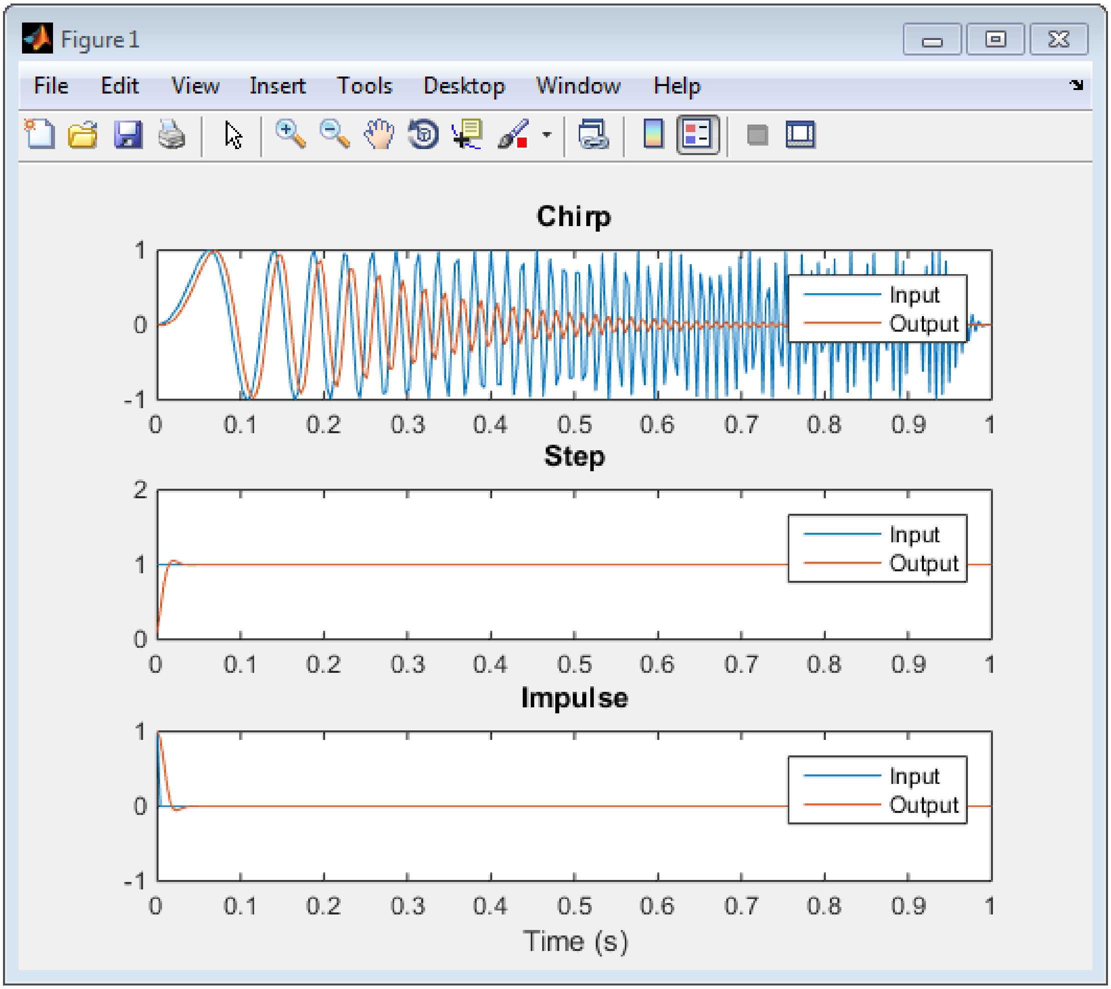
The app screens `ex_2ndOrder_filter.m` for code violations and code generation readiness issues. The app does not find issues in `ex_2ndOrder_filter.m`.

Define Input Types

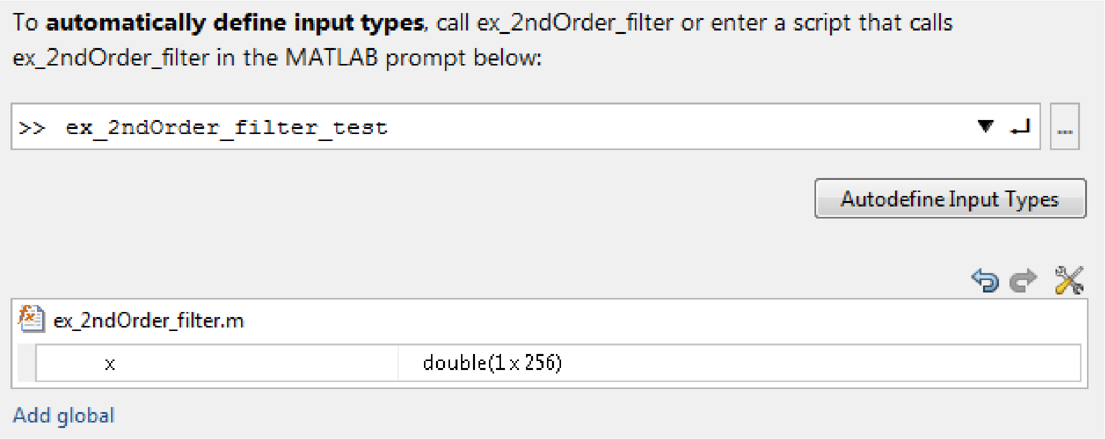
1 On the **Define Input Types** page, to add `ex_2ndOrder_filter_test` as a test file, browse to `ex_2ndOrder_filter_test`, and then click **Open**.

2 Click **Autodefine Input Types**.

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



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



To **automatically define input types**, call ex_2ndOrder_filter or enter a script that calls ex_2ndOrder_filter in the MATLAB prompt below:

```
>> ex_2ndOrder_filter_test
```

ex_2ndOrder_filter.m	
x	double(1 x 256)

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

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `ex_2ndOrder_filter_test` replacing calls to `ex_2ndOrder_filter` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in a window where you can edit the code.

1 On the **Check for Run-Time Issues** page, the app populates the test file field with `ex_2ndOrder_filter_test`, the test file that you used to define the input types.

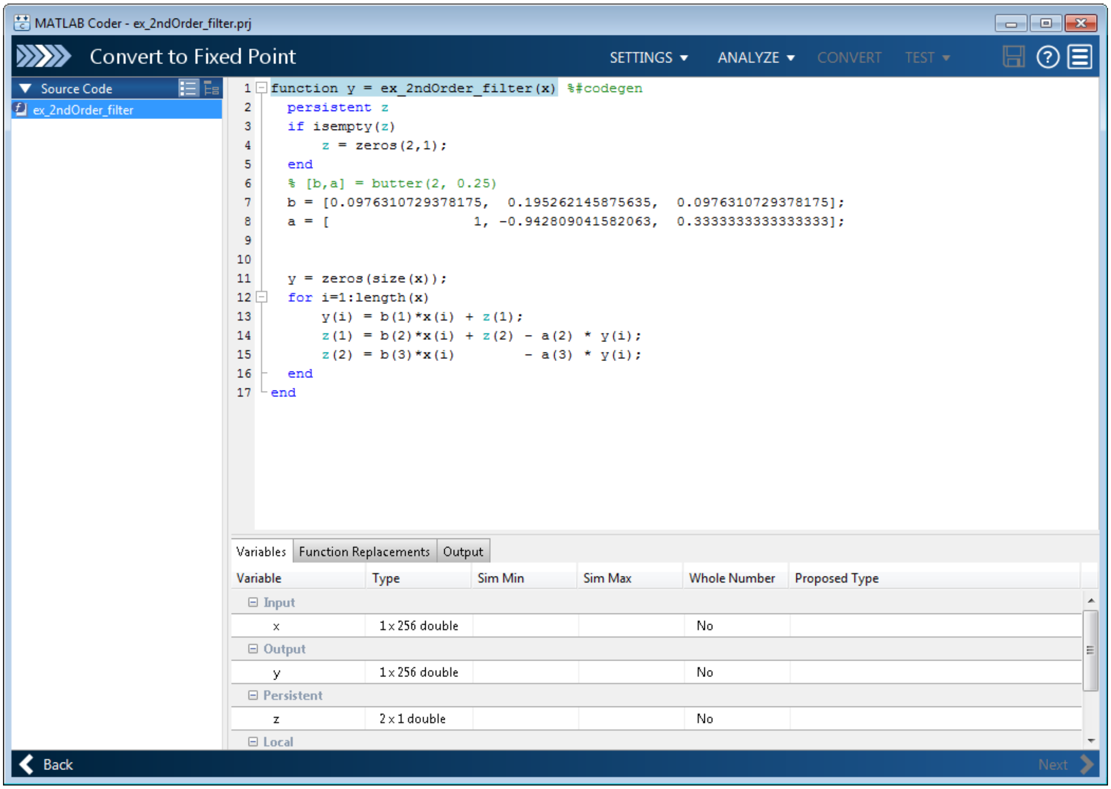
2 Click **Check for Issues**.

   The app does not detect issues.

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

Convert to Fixed Point

1 The app displays compiled information—type, size, and complexity—for variables in your code. See “View and Modify Variable Information” on page 14-79.



```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                  1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

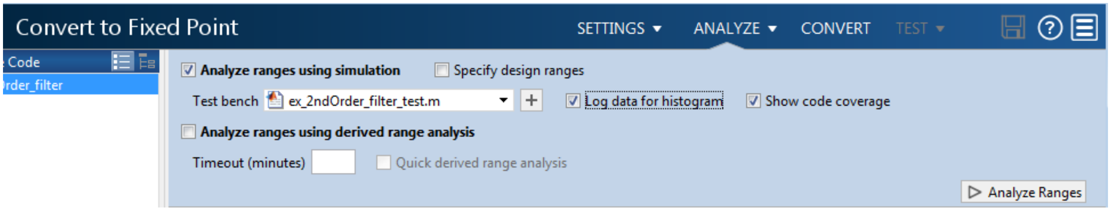
Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
x	1 x 256 double			No	
⊟ Output					
y	1 x 256 double			No	
⊟ Persistent					
z	2 x 1 double			No	
⊟ Local					

On the **Function Replacements** tab, the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 14-88.

2 Click the **Analyze** arrow ▼. Verify that **Analyze ranges using simulation** is selected and that the test bench file is `ex_2ndOrder_filter_test`. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the app merges the simulation results.

3 Select **Log data for histogram**.

By default, the **Show code coverage** option is selected. This option provides code coverage information that helps you verify that your test file is testing your algorithm over the intended operating range.



☑ Analyze ranges using simulation □ Specify design ranges

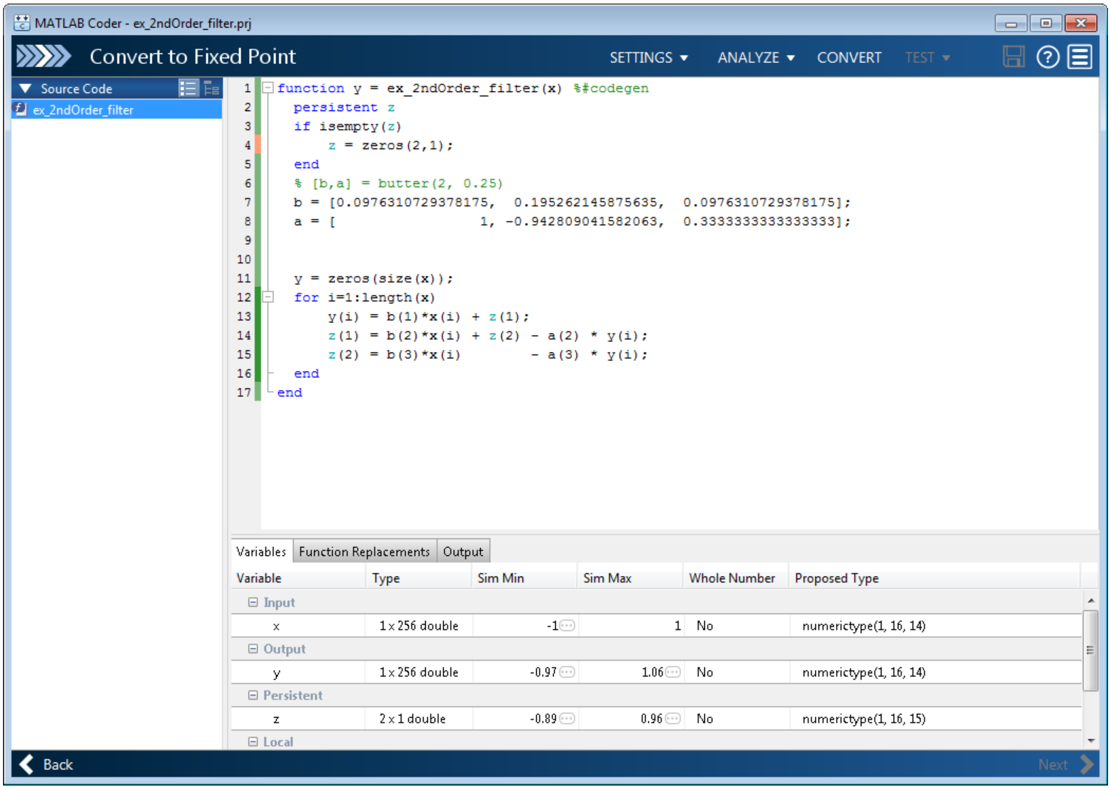
Test bench ex_2ndOrder_filter_test.m ☑ Log data for histogram ☑ Show code coverage

□ Analyze ranges using derived range analysis

Timeout (minutes) □ Quick derived range analysis

4 Click **Analyze**.

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



```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                 1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

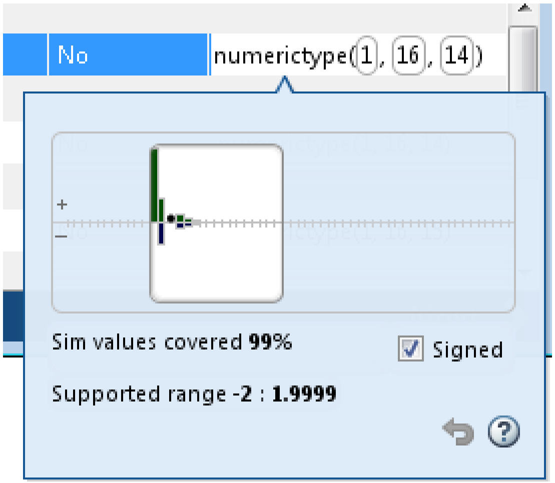
Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
x	1 x 256 double	-1 ⋯	1	No	numerictype(1, 16, 14)
⊟ Output					
y	1 x 256 double	-0.97 ⋯	1.06 ⋯	No	numerictype(1, 16, 14)
⊟ Persistent					
z	2 x 1 double	-0.89 ⋯	0.96 ⋯	No	numerictype(1, 16, 15)
⊟ Local					

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

The app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The app enables the **Convert** option.

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

5 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.



Sim values covered 99%

Supported range -2 : 1.9999

To modify the proposed data types, either enter the required type into the **Proposed Type** field or use the histogram controls. For more information about the histogram, see “Log Data for Histogram” on page 14-99.

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

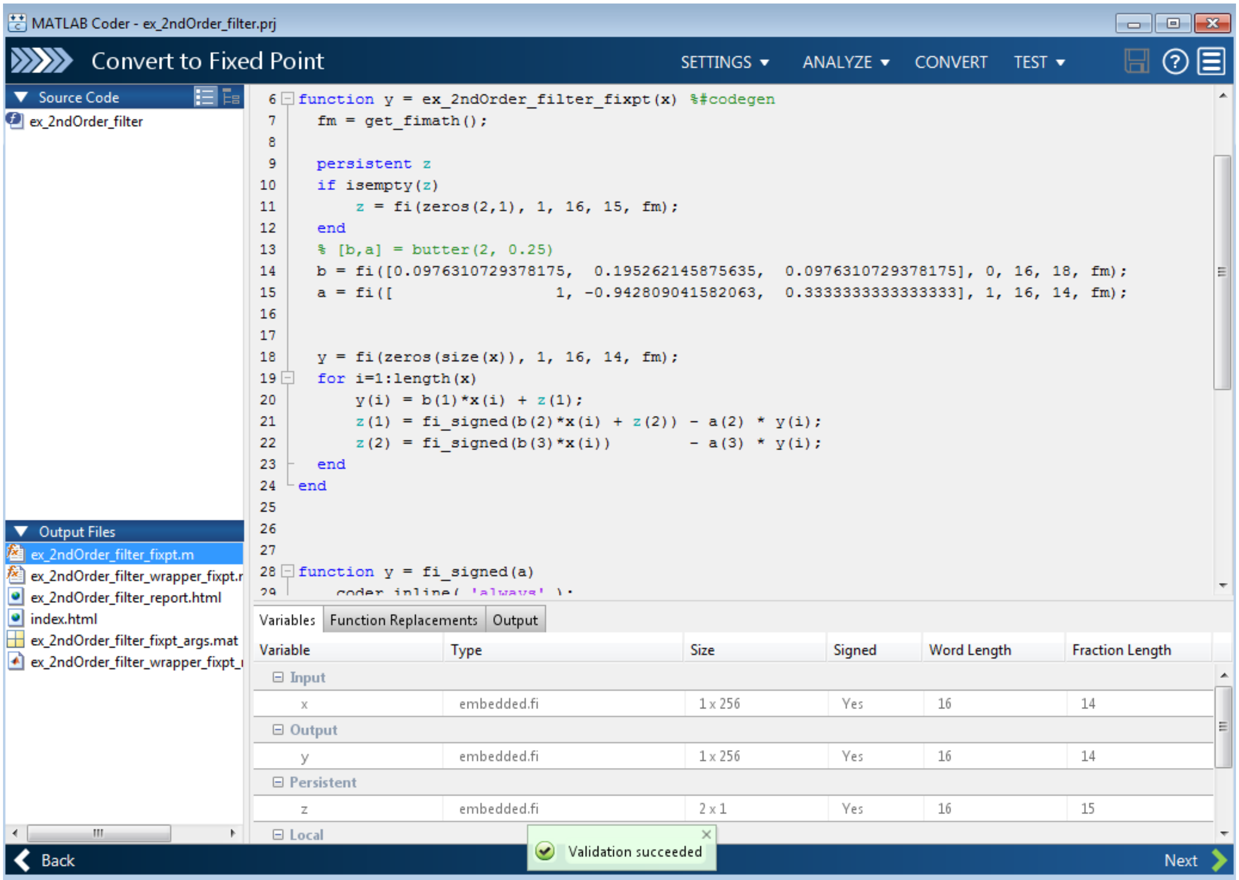
During the fixed-point conversion process, the software validates the proposed types and generates the following files in the `codegen\ex_2ndOrder_filter\fixpt` folder in your local working folder.

- `ex_2ndOrder_filter_fixpt.m` — the fixed-point version of `ex_2ndOrder_filter.m`.
- `ex_2ndOrder_filter_wrapper_fixpt.m` — this file converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during conversion. These fixed-point values are fed into the converted fixed-point design, `ex_2ndOrder_filter_fixpt.m`.
- `ex_2ndOrder_filter_fixpt_report.html` — this report shows the generated fixed-point code and the fixed-point instrumentation results.
- `ex_2ndOrder_filter_report.html` — this report shows the original algorithm and the fixed-point instrumentation results.

- `ex_2ndOrder_filter_fixpt_args.mat` — MAT-file containing a structure for the input arguments, a structure for the output arguments and the name of the fixed-point file.

If errors or warnings occur during validation, you see them on the **Type Validation Output** tab. See “Validating Types” on page 14-102.

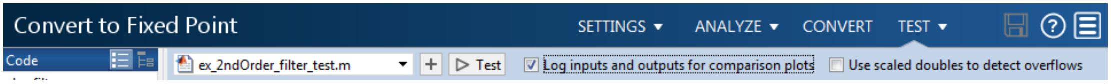
7 In the **Output Files** list, select `ex_2ndOrder_filter_fixpt.m`. The app displays the generated fixed-point code.


MATLAB Coder - ex_2ndOrder_filter.prj
Convert to Fixed Point
SETTINGS ▼
ANALYZE ▼
CONVERT
TEST ▼
Source Code
ex_2ndOrder_filter
Output Files
ex_2ndOrder_filter_fixpt.m
ex_2ndOrder_filter_wrapper_fixpt.r
ex_2ndOrder_filter_report.html
index.html
ex_2ndOrder_filter_fixpt_args.mat
ex_2ndOrder_filter_wrapper_fixpt_i
Variables
Function Replacements
Output
Validation succeeded
Back
Next

```
function y = ex_2ndOrder_filter_fixpt(x) %#codegen
  fm = get_fimath();

  persistent z
  if isempty(z)
      z = fi(zeros(2,1), 1, 16, 15, fm);
  end
  % [b,a] = butter(2, 0.25)
  b = fi([0.097631072937817,  0.195262145875635,  0.097631072937817], 0, 16, 18, fm);
  a = fi([                 1, -0.942809041582063,  0.333333333333333], 1, 16, 14, fm);


  y = fi(zeros(size(x)), 1, 16, 14, fm);
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
      z(2) = fi_signed(b(3)*x(i))        - a(3) * y(i);
  end
end



function y = fi_signed(a)
    coder.inline( 'always' );
```

Variable	Type	Size	Signed	Word Length	Fraction Length
⊟ Input					
x	embedded.fi	1 x 256	Yes	16	14
⊟ Output					
y	embedded.fi	1 x 256	Yes	16	14
⊟ Persistent					
z	embedded.fi	2 x 1	Yes	16	15
⊟ Local					

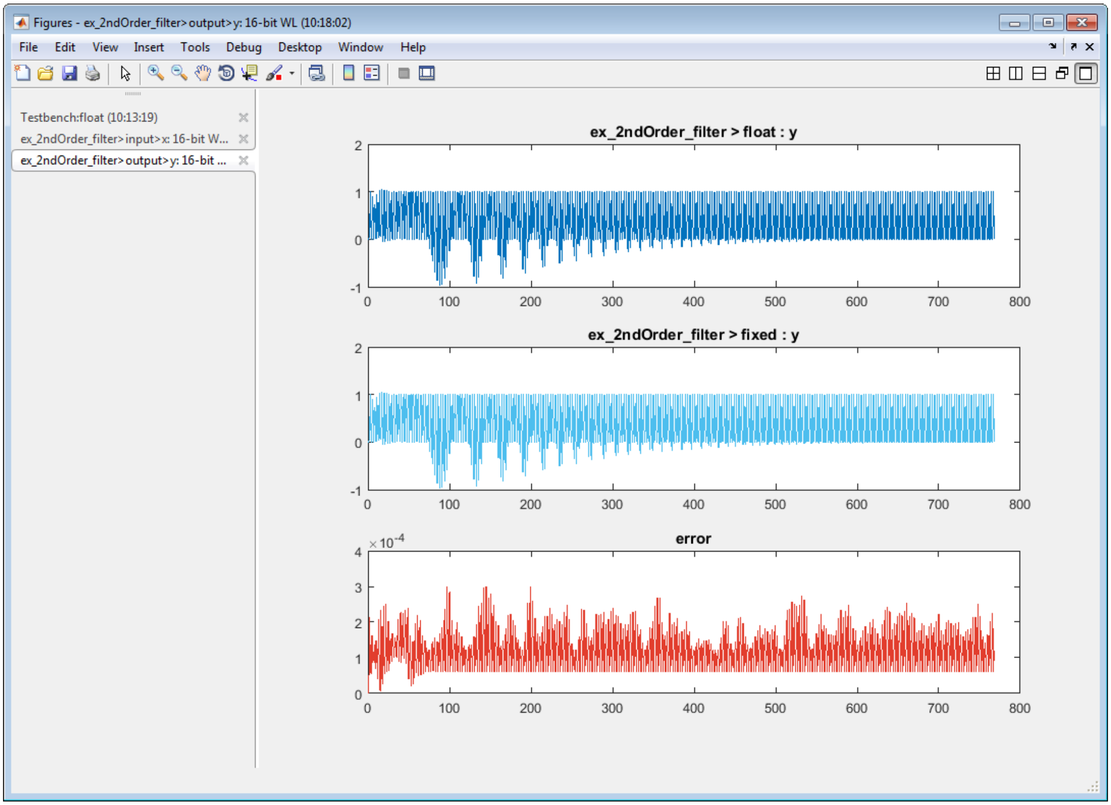
8 Click the **Test** arrow ▼. Select **Log inputs and outputs for comparison plots**, and then click **Test**.



☑ Log inputs and outputs for comparison plots

□ Use scaled doubles to detect overflows

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


Figures - ex_2ndOrder_filter>output>y: 16-bit WL (10:18:02)

Testbench:float (10:13:19)

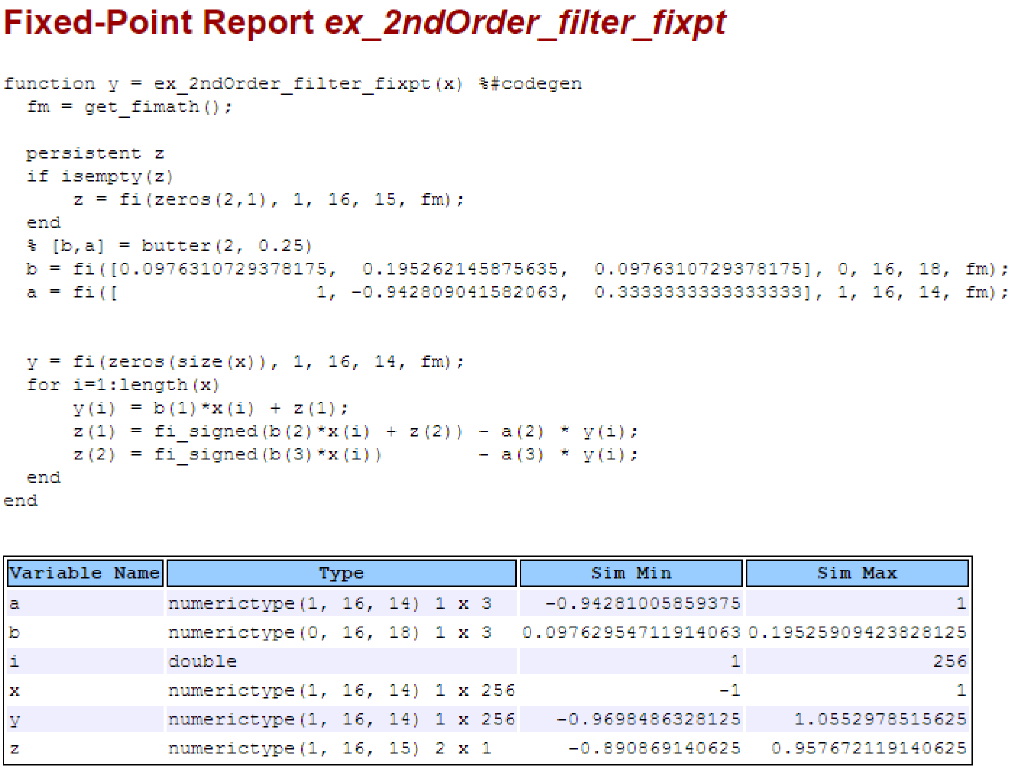
ex_2ndOrder_filter>input>x: 16-bit W...

ex_2ndOrder_filter>output>y: 16-bit ...

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

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see "Testing Numerics" on page 14-102.

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



Fixed-Point Report *ex_2ndOrder_filter_fixpt*

```
function y = ex_2ndOrder_filter_fixpt(x) %#codegen
  fm = get_fimath();

  persistent z
  if isempty(z)
      z = fi(zeros(2,1), 1, 16, 15, fm);
  end
  % [b,a] = butter(2, 0.25)
  b = fi([0.0976310729378175,  0.195262145875635,  0.0976310729378175], 0, 16, 18, fm);
  a = fi([                 1, -0.942809041582063,  0.3333333333333333], 1, 16, 14, fm);


  y = fi(zeros(size(x)), 1, 16, 14, fm);
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
      z(2) = fi_signed(b(3)*x(i))        - a(3) * y(i);
  end
end
```

Variable Name	Type	Sim Min	Sim Max
a	numerictype(1, 16, 14) 1 x 3	-0.94281005859375	1
b	numerictype(0, 16, 18) 1 x 3	0.09762954711914063	0.19525909423828125
i	double	1	256
x	numerictype(1, 16, 14) 1 x 256	-1	1
y	numerictype(1, 16, 14) 1 x 256	-0.9698486328125	1.0552978515625
z	numerictype(1, 16, 15) 2 x 1	-0.890869140625	0.957672119140625

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

Generate Fixed-Point C Code

1. In the **Generate** dialog box, set **Build source** to `Fixed-Point` and **Build type** to `Static Library`.
2. Set **Language** to **C**.
3. Click **Generate** to generate a library using the default project settings.

   MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/ex_2ndOrder_filter`.
4. The app displays the generated code for `ex_2ndOrder_filter.c`. In the generated C code, variables are assigned fixed-point data types.
5. Click **Next** to go to the **Finish Workflow** page.

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

Propose Fixed-Point Data Types Based on Derived Ranges

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

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	`dti.m`	Entry-point MATLAB function
Test file	`dti_test.m`	MATLAB script that tests `dti.m`

The dti Function

The `dti` function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
```

```
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
 y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
 y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;
```

The dti_test Function

The test script runs the `dti` function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10);

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;

    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);

end

figure('Name', [mfilename, '_plot']);
subplot(2,1,1)
plot(1:len,x_in)
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

subplot(2,1,2)
plot(1:len,y_out)
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (DTI)')

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

Open the MATLAB Coder App

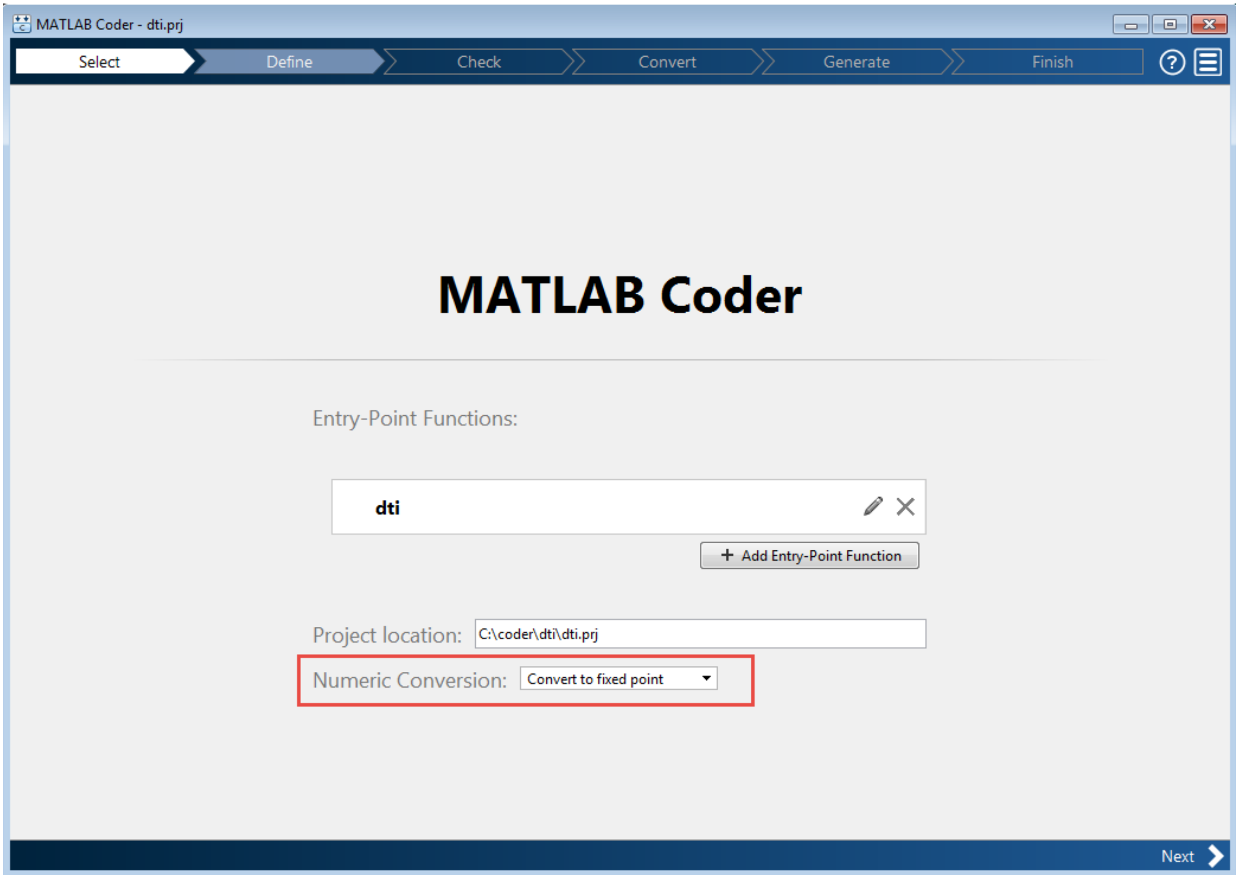
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Select Source Files

To add the entry-point function `dti` to the project, browse to the file `dti.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `dti.prj`.

Enable Fixed-Point Conversion

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



Entry-Point Functions:

dti

Project location: C:\coder\dti\dti.prj

Numeric Conversion: Convert to fixed point

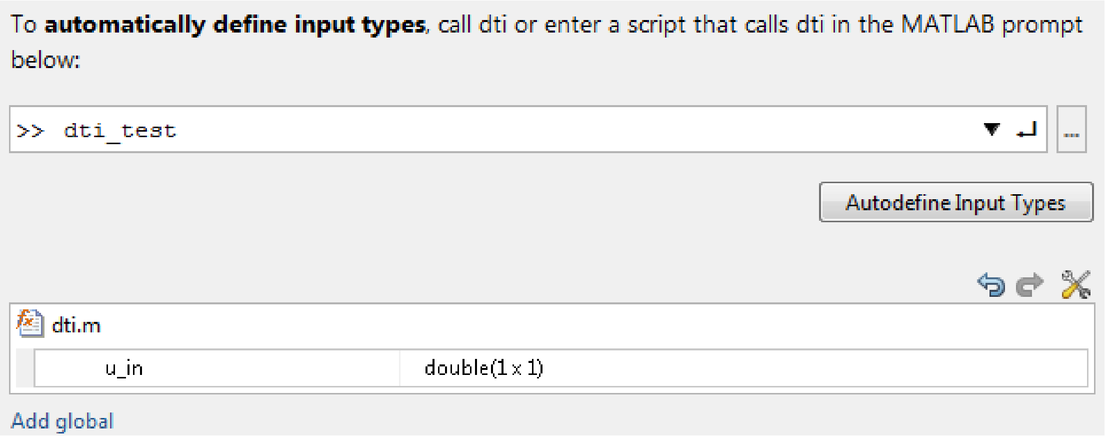
2 Click **Next** to go to the **Define Input Types** step.

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

Define Input Types

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

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



To **automatically define input types**, call dti or enter a script that calls dti in the MATLAB prompt below:

```
>> dti_test
```

dti.m

u_in	double(1 x 1)

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

Check for Run-Time Issues

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

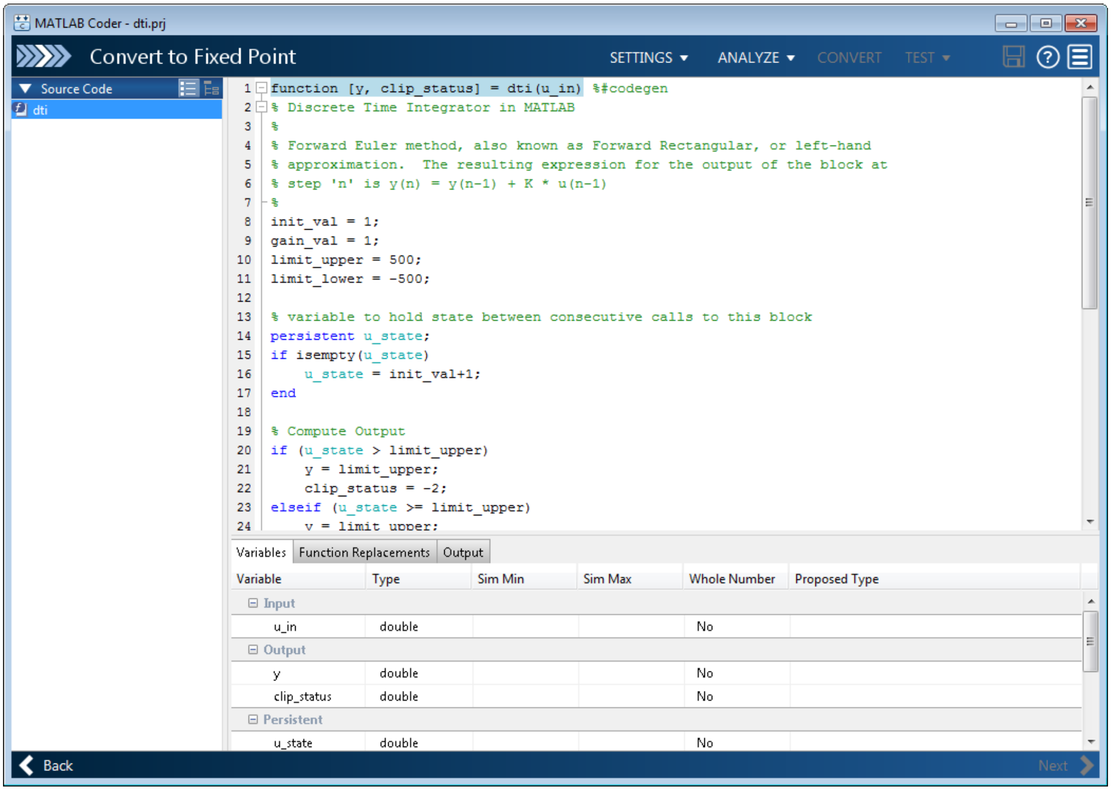
1. On the **Check for Run-Time Issues** page, the app populates the test file field with `dti_test`, the test file that you used to define the input types.
2. Click **Check for Issues**.

   The app does not detect issues.

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

Convert to Fixed Point

1 The app displays compiled information—type, size, and complexity—for variables in your code. For more information, see “View and Modify Variable Information” on page 14-79.



```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
u_in	double			No	
⊟ Output					
y	double			No	
clip_status	double			No	
⊟ Persistent					
u_state	double			No	

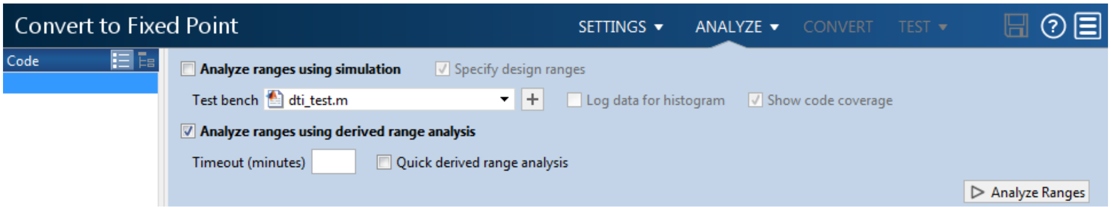
If functions are not supported for fixed-point conversion, the app displays them on the **Function Replacements** tab.

2 Click the **Analyze** arrow ▼.

a Select **Analyze ranges using derived range analysis**.

b Clear the **Analyze ranges using simulation** check box.

Design ranges are required to use derived range analysis.



□ Analyze ranges using simulation ☑ Specify design ranges

Test bench dti_test.m □ Log data for histogram ☑ Show code coverage

☑ Analyze ranges using derived range analysis

Timeout (minutes) □ Quick derived range analysis

3 On the **Convert to Fixed Point** page, on the **Variables** tab, for input `u_in`, select **Static Min** and set it to `-1`. Set **Static Max** to `1`.

To compute derived range information, at a minimum you must specify static minimum and maximum values or proposed data types for all input variables.

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

4 Click **Analyze**.

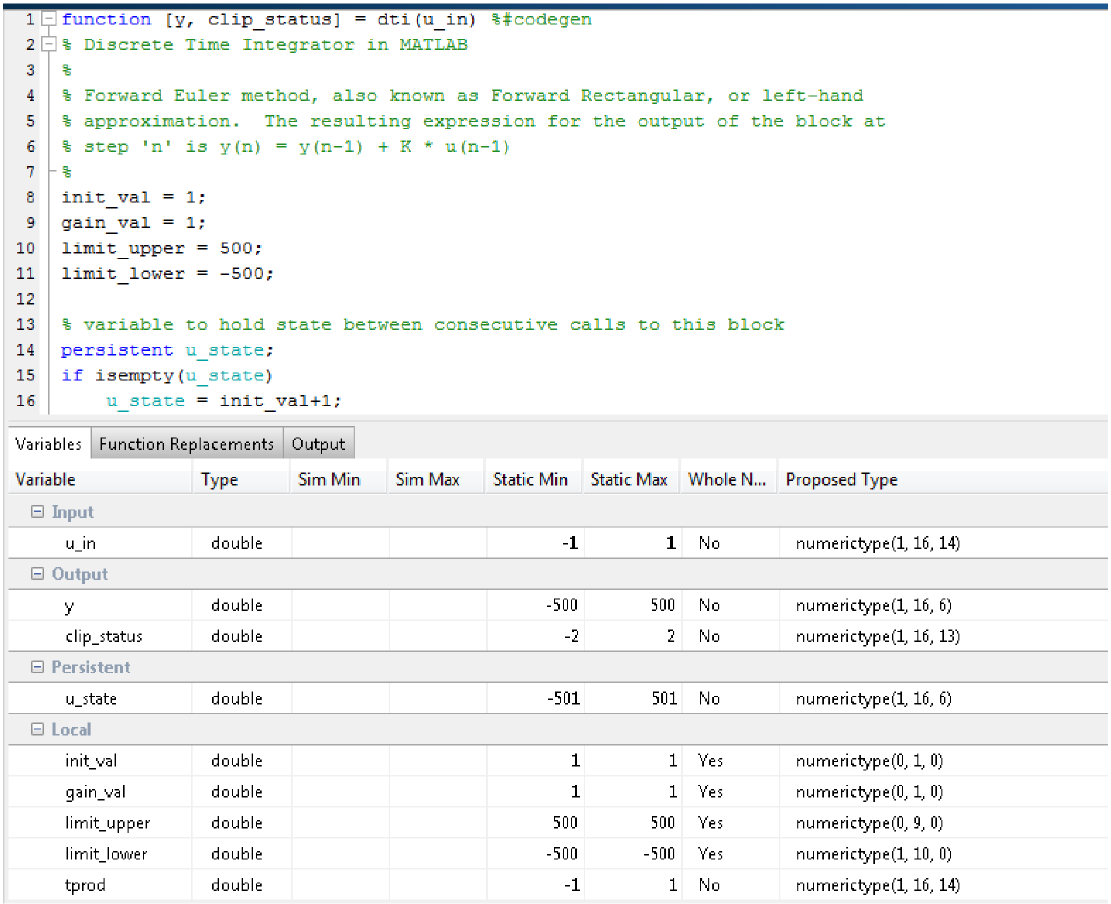
Range analysis computes the derived ranges and displays them in the **Variables** tab. Using these derived ranges, the analysis proposes fixed-point types for each variable based on the default type proposal settings. The app displays them in the **Proposed Type** column.

In the `dti` function, the `clip_status` output has a minimum value of `-2` and a maximum of `2`.

```
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
 y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
 y = limit_lower;
```

```
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```

When you derive ranges, the app analyzes the function and computes these minimum and maximum values for `clip_status`.



```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = init_val+1;
```

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

The app provides a **Quick derived range analysis** option and the option to specify a timeout in case the analysis takes a long time. See “Computing Derived Ranges” on page 14-89.

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

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

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

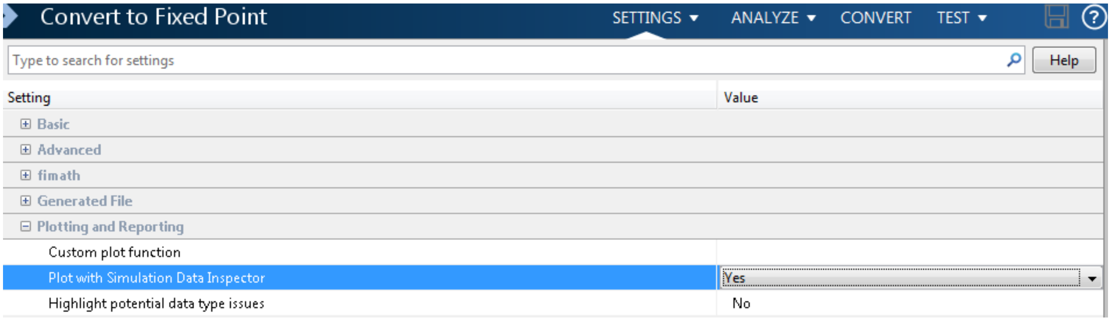
   If errors or warnings occur during validation, they show on the **Output** tab. See “Validating Types” on page 14-102.

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

7. Use the Simulation Data Inspector to plot the floating-point and fixed-point results.

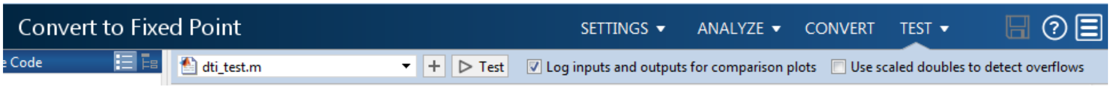
   a. Click the **Settings** arrow ▼.

   b. Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to `Yes`.



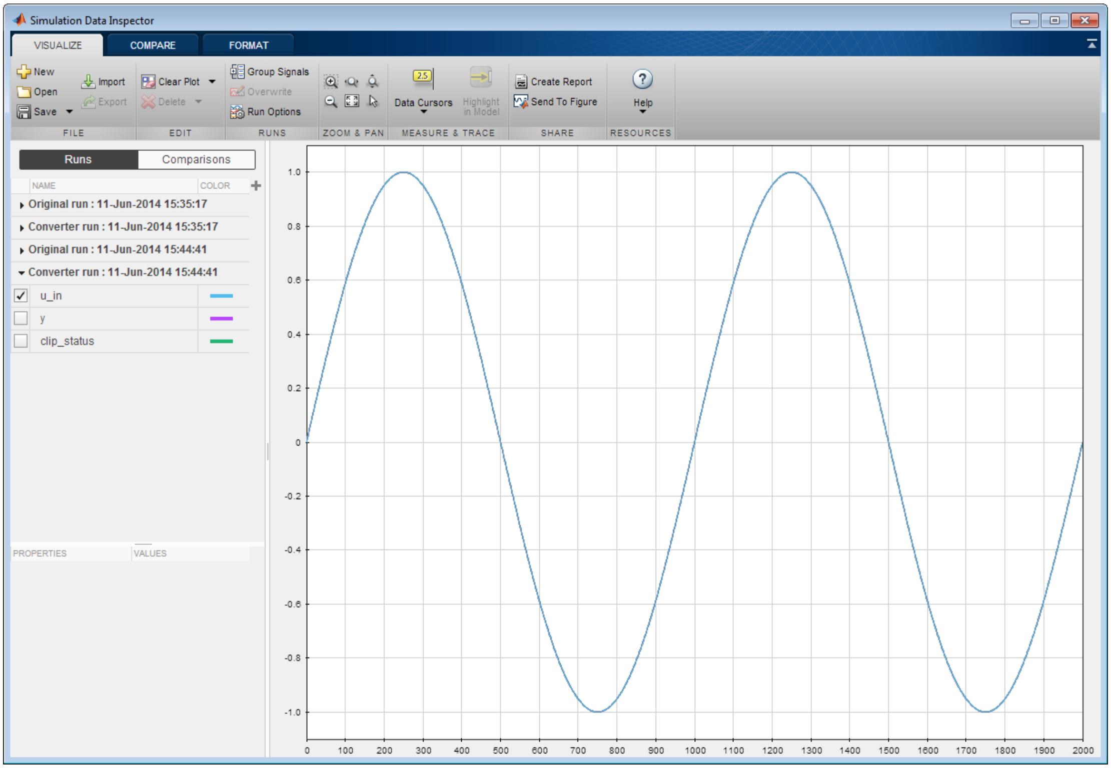
Setting	Value
⊞ Basic	
⊞ Advanced	
⊞ fimath	
⊞ Generated File	
⊟ Plotting and Reporting	
Custom plot function	
Plot with Simulation Data Inspector	Yes
Highlight potential data type issues	No

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



☑ Log inputs and outputs for comparison plots □ Use scaled doubles to detect overflows

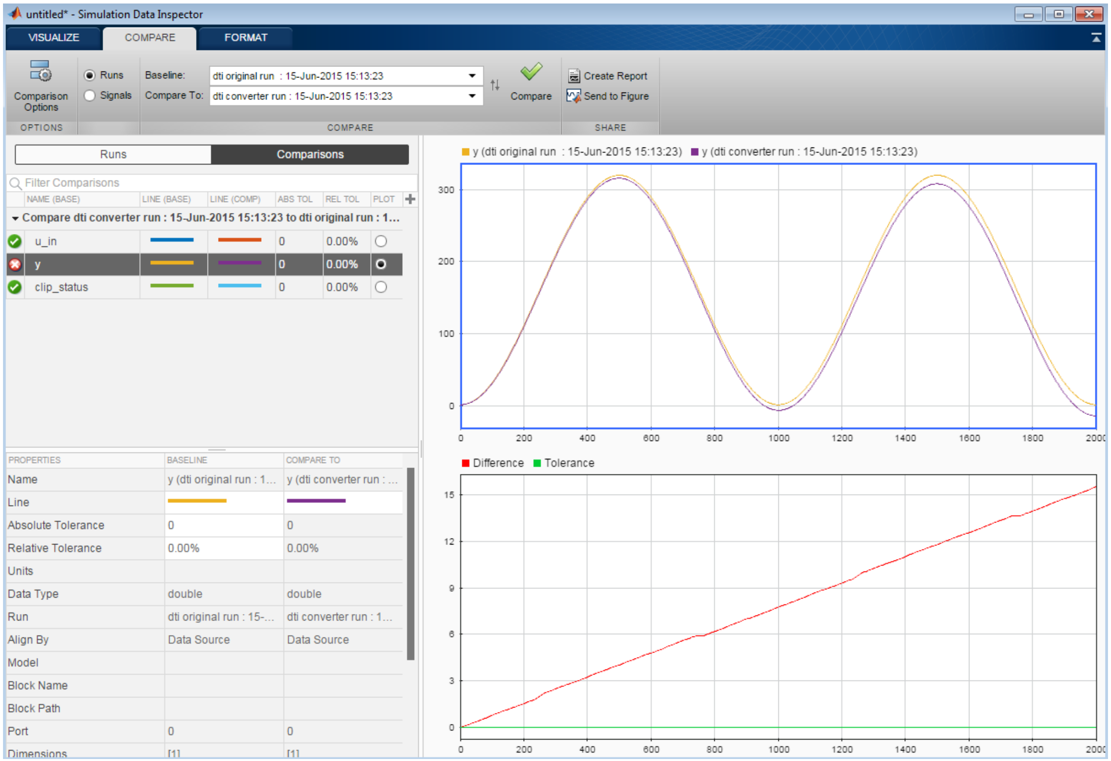
The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable `y`. Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.



NAME	COLOR
▸ Original run : 11-Jun-2014 15:35:17	
▸ Converter run : 11-Jun-2014 15:35:17	
▸ Original run : 11-Jun-2014 15:44:41	
▾ Converter run : 11-Jun-2014 15:44:41	
☑ u_in	—
□ y	—
□ clip_status	—

d You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-point values for the output `y`, on the **Compare** tab, select `y`. Select **Runs** and then click **Compare**.

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



● Runs ○ Signals

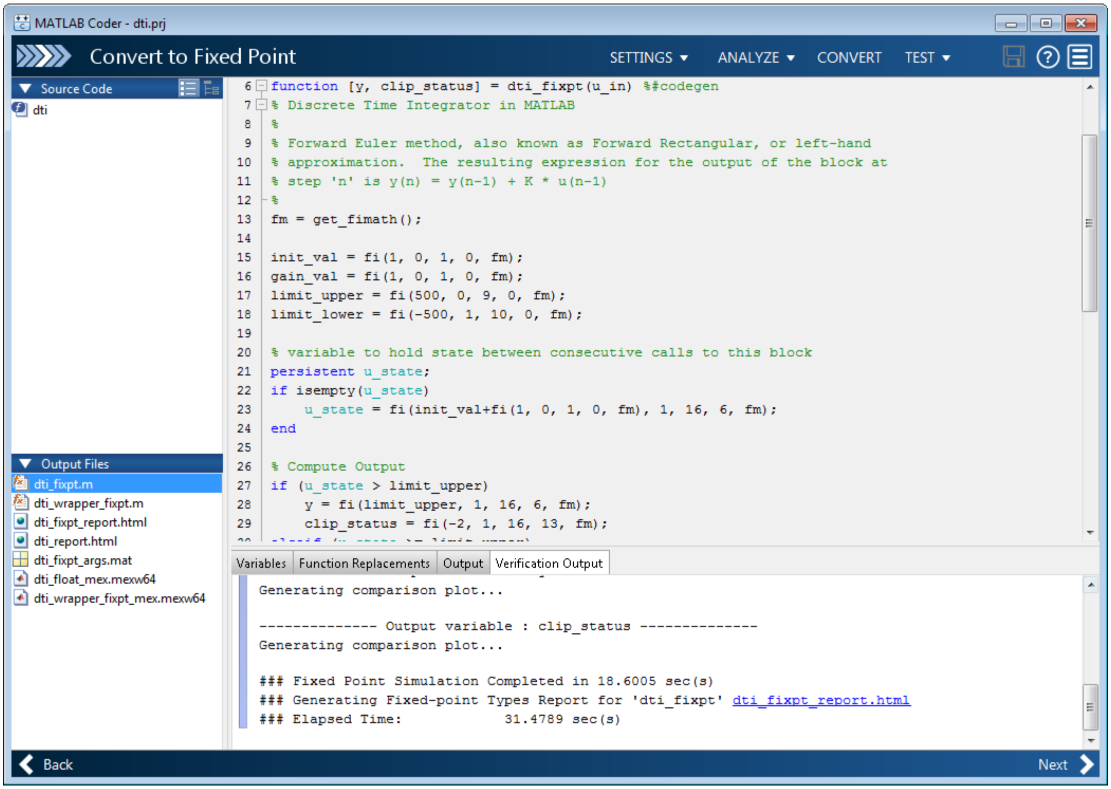
Baseline: dti original run : 15-Jun-2015 15:13:23

Compare To: dti converter run : 15-Jun-2015 15:13:23

	NAME (BASE)	LINE (BASE)	LINE (COMP)	ABS TOL	REL TOL	PLOT
	Compare dti converter run : 15-Jun-2015 15:13:23 to dti original run : 1...					
✓	u_in	━	━	0	0.00%	○
✗	y	━	━	0	0.00%	●
✓	clip_status	━	━	0	0.00%	○

PROPERTIES	BASELINE	COMPARE TO
Name	y (dti original run : 1...	y (dti converter run : ...
Line	━	━
Absolute Tolerance	0	0
Relative Tolerance	0.00%	0.00%
Units		
Data Type	double	double
Run	dti original run : 15-...	dti converter run : 1...
Align By	Data Source	Data Source
Model		
Block Name		
Block Path		
Port	0	0
Dimensions	[1]	[1]

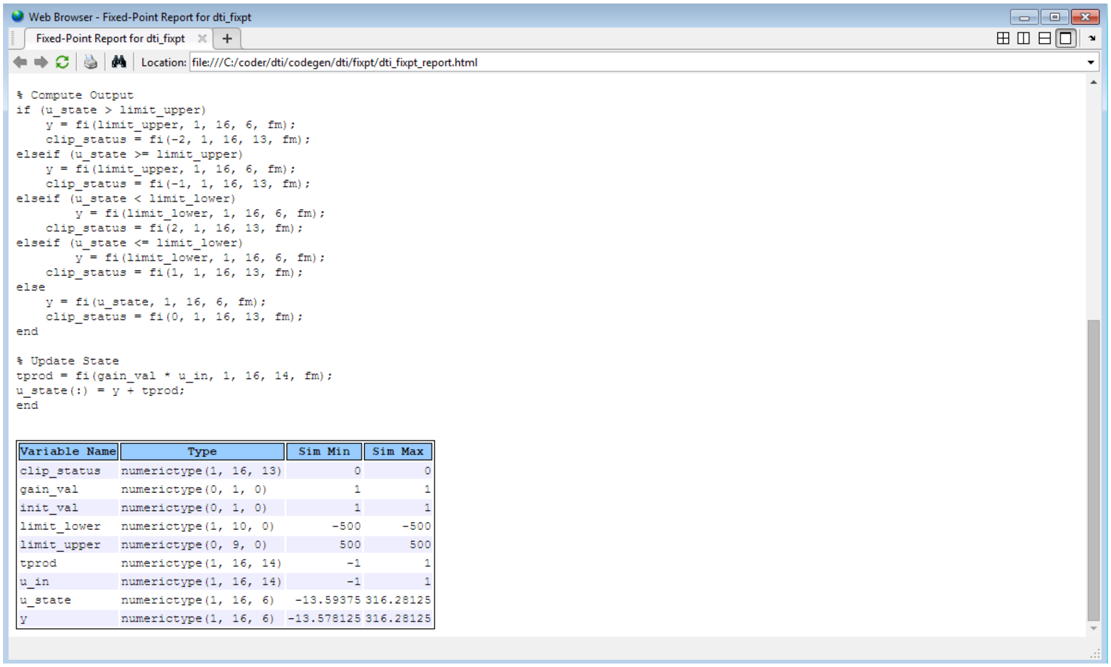
8 On the **Verification Output** tab, the app provides a link to a type proposal report.



```
function [y, clip_status] = dti_fixpt(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
fm = get_fimath();

init_val = fi(1, 0, 1, 0, fm);
gain_val = fi(1, 0, 1, 0, fm);
limit_upper = fi(500, 0, 9, 0, fm);
limit_lower = fi(-500, 1, 10, 0, fm);

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = fi(init_val+fi(1, 0, 1, 0, fm), 1, 16, 6, fm);
end

% Compute Output
if (u_state > limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-2, 1, 16, 13, fm);
```

```
Generating comparison plot...

-------------- Output variable : clip_status --------------
Generating comparison plot...

### Fixed Point Simulation Completed in 18.6005 sec(s)
### Generating Fixed-point Types Report for 'dti_fixpt' dti_fixpt_report.html
### Elapsed Time:            31.4789 sec(s)
```

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


Web Browser - Fixed-Point Report for dti_fixpt
Fixed-Point Report for dti_fixpt
Location: file:///C:/coder/dti/codegen/dti/fixpt/dti_fixpt_report.html

```
% Compute Output
if (u_state > limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-2, 1, 16, 13, fm);
elseif (u_state >= limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-1, 1, 16, 13, fm);
elseif (u_state < limit_lower)
        y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(2, 1, 16, 13, fm);
elseif (u_state <= limit_lower)
        y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(1, 1, 16, 13, fm);
else
    y = fi(u_state, 1, 16, 6, fm);
    clip_status = fi(0, 1, 16, 13, fm);
end

% Update State
tprod = fi(gain_val * u_in, 1, 16, 14, fm);
u_state(:) = y + tprod;
end
```

Variable Name	Type	Sim Min	Sim Max
clip_status	numerictype(1, 16, 13)	0	0
gain_val	numerictype(0, 1, 0)	1	1
init_val	numerictype(0, 1, 0)	1	1
limit_lower	numerictype(1, 10, 0)	-500	-500
limit_upper	numerictype(0, 9, 0)	500	500
tprod	numerictype(1, 16, 14)	-1	1
u_in	numerictype(1, 16, 14)	-1	1
u_state	numerictype(1, 16, 6)	-13.59375	316.28125
y	numerictype(1, 16, 6)	-13.578125	316.28125

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

Generate Fixed-Point C Code

1 In the **Generate** dialog box, set **Build source** to `Fixed-Point` and **Build type** to `Source Code`.

2 Set **Language** to **C**.

3 Click **Generate** to generate a library using the default project settings.

   MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/dti_fixpt`.

4 The app displays the generated code for `dti_fixpt.c`. In the generated C code, variables are assigned fixed-point data types.

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

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

Specify Type Proposal Options

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

The following options are available.

Basic Type Proposal Settings	Values	Description
Fixed-point type proposal mode	Propose fraction lengths for specified word length	Use the specified word length for data type proposals and propose the minimum fraction lengths to avoid overflows.
	Propose word lengths for specified fraction length (default)	Use the specified fraction length for data type proposals and propose the minimum word lengths to avoid overflows.
Default word length	16 (default)	Default word length to use when **Fixed-point type proposal mode** is set to `Propose fraction lengths for specified word lengths`
Default fraction length	4 (default)	Default fraction length to use when **Fixed-point type proposal mode** is set to `Propose word lengths for specified fraction lengths`

Advanced Type Proposal Settings	Values	Description
When proposing types **Note:** Manually-entered static ranges always take precedence over simulation ranges.	ignore simulation ranges	Propose data types based on derived ranges.
	ignore derived ranges	Propose data types based on simulation ranges.

Advanced Type Proposal Settings	Values	Description
	use all collected data (default)	Propose data types based on both simulation and derived ranges.
Propose target container types	Yes	Propose data type with the smallest word length that can represent the range and is suitable for C code generation (8,16,32, 64 …). For example, for a variable with range `[0..7]`, propose a word length of 8 rather than 3.
	No (default)	Propose data types with the minimum word length needed to represent the value.
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.

Advanced Type Proposal Settings	Values	Description
Safety margin for sim min/max (%)	0 (default)	Specify safety factor for simulation minimum and maximum values. The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of `55` specifies that you want a range at least 55 percent larger. A value of `-15` specifies that a range up to 15 percent smaller is acceptable.
Search paths	`''` (default)	Add paths to the list of paths to search for MATLAB files. Separate list items with a semicolon.

fimath Settings	Values	Description
Rounding method	Ceiling	Specify the `fimath` properties for the generated fixed-point data types. The default fixed-point math properties use the `Floor` rounding and `Wrap` overflow because they are the default actions in C. These settings generate the most efficient code but might cause problems with overflow. After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	

fimath Settings	Values	Description
	KeepLSB	bias, and then rerun the verification. For more information on `fimath` properties, see “fimath Object Properties” (Fixed-Point Designer).
	KeepMSB	
	SpecifyPrecision	

Generated File Settings	Value	Description
Generated fixed-point file name suffix	_fixpt (default)	Specify the suffix to add to the generated fixed-point file names. For example, by default, if you generate a static library for a project named `test`, the generated files are in the subfolder `codegen\lib\test_fixpt`. The generated static library is named `test.lib`, but the generated C code files use the suffix, for example, `test_fixpt.c`.

Plotting and Reporting Settings	Values	Description
Custom plot function	`''` (default)	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	Specify whether to use the Simulation Data Inspector for comparison plots.
	Yes	
Highlight potential data type issues	No (default)	Specify whether to highlight potential data types in the generated html report. If this option is turned on, the report highlights single-
	Yes	

Plotting and Reporting Settings	Values	Description
		precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.

Detect Overflows

This example shows how to detect overflows using the MATLAB Coder app. At the numerical testing stage in the conversion process, you choose to simulate the fixed-point code using scaled doubles. The app then reports which expressions in the generated code produce values that overflow the fixed-point data type.

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `overflow.m` and `overflow_test.m` files to your local working folder.

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

The overflow Function

```
function y = overflow(b,x,reset)
    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
```

```
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
end
```

The overflow_test Function

You use this test file to define input types for b, x, and reset, and, later, to verify the fixed-point version of the algorithm.

```
function overflow_test
    % The filter coefficients were computed using the FIR1 function from
    % Signal Processing Toolbox.
    %   b = fir1(11,0.25);
    b = [-0.004465461051254
         -0.004324228005260
         +0.012676739550326
         +0.074351188907780
         +0.172173206073645
         +0.249588554524763
         +0.249588554524763
         +0.172173206073645
         +0.074351188907780
         +0.012676739550326
         -0.004324228005260
         -0.004465461051254]';

    % Input signal
    nx = 256;
    t = linspace(0,10*pi,nx)';

    % Impulse
    x_impulse = zeros(nx,1); x_impulse(1) = 1;

    % Max Gain
    % The maximum gain of a filter will occur when the inputs line up with the
    % signs of the filter's impulse response.
    x_max_gain = sign(b)';
    x_max_gain = repmat(x_max_gain,ceil(nx/length(b)),1);
    x_max_gain = x_max_gain(1:nx);


    % Sums of sines
    f0=0.1; f1=2;
    x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);

    % Chirp
    f_chirp = 1/16;                  % Target frequency
    x_chirp = sin(pi*f_chirp*t.^2);  % Linear chirp

    x = [x_impulse, x_max_gain, x_sines, x_chirp];
    titles = {'Impulse', 'Max gain', 'Sum of sines', 'Chirp'};
    y = zeros(size(x));

    for i=1:size(x,2)
        reset = true;
        y(:,i) = overflow(b,x(:,i),reset);
    end

    test_plot(1,titles,t,x,y)

end
function test_plot(fig,titles,t,x,y1)
    figure(fig)
    clf
    sub_plot = 1;
    font_size = 10;
    for i=1:size(x,2)
        subplot(4,1,sub_plot)
        sub_plot = sub_plot+1;
        plot(t,x(:,i),'c',t,y1(:,i),'k')
        axis('tight')
        xlabel('t','FontSize',font_size);
        title(titles{i},'FontSize',font_size);
        ax = gca;
        ax.FontSize = 10;
    end
    figure(gcf)
end
```

Open the MATLAB Coder App

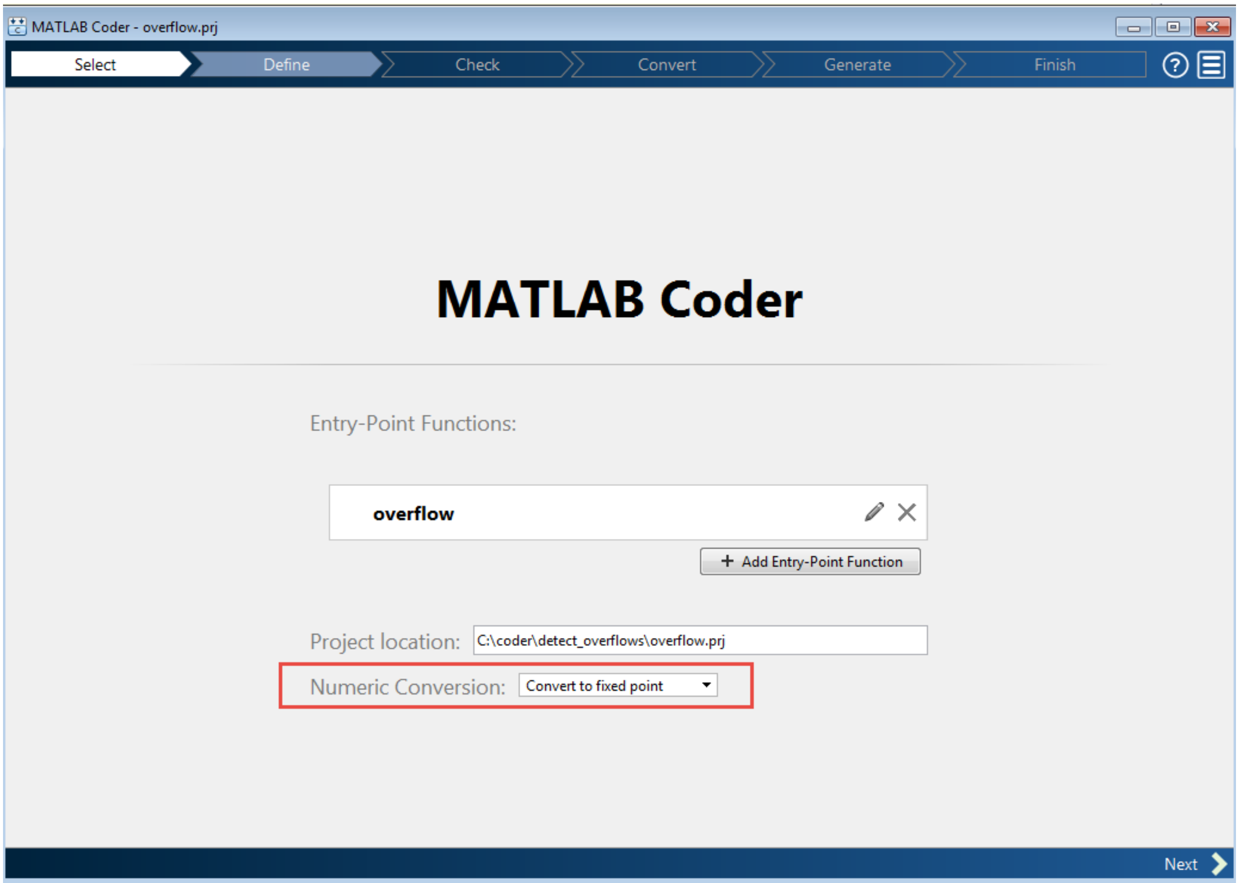
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Select Source Files

To add the entry-point function `overflow` to the project, browse to the file `overflow.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `overflow.prj`.

Enable Fixed-Point Conversion

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



Project location: C:\coder\detect_overflows\overflow.prj

Numeric Conversion: Convert to fixed point

2 Click **Next** to go to the **Define Input Types** step.

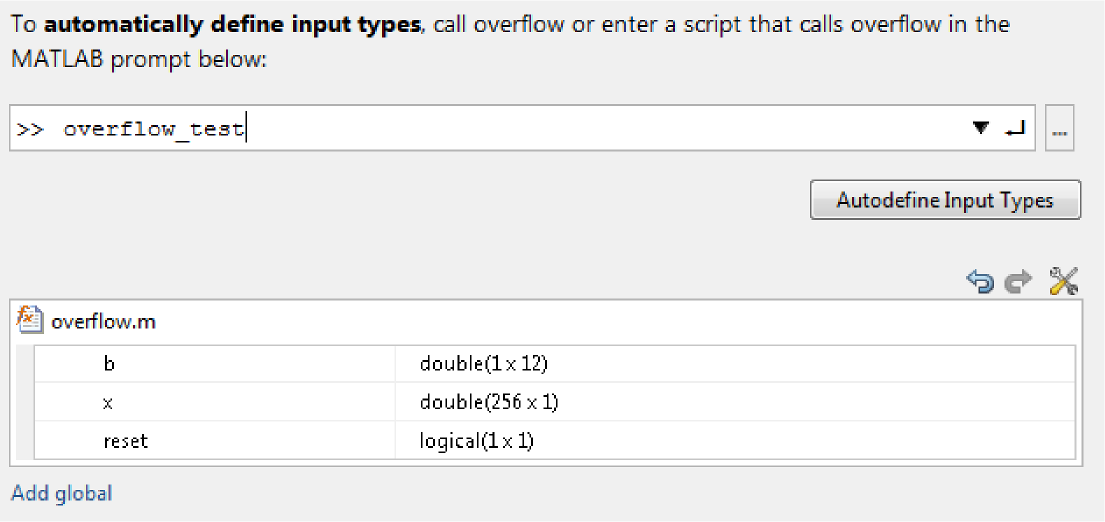
The app screens `overflow.m` for code violations and code generation readiness issues. The app does not find issues in `overflow.m`.

Define Input Types

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

2 Click **Autodefine Input Types**.

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



To **automatically define input types**, call overflow or enter a script that calls overflow in the MATLAB prompt below:

```
>> overflow_test
```

overflow.m

b	double(1 x 12)
x	double(256 x 1)
reset	logical(1 x 1)

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

Check for Run-Time Issues

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

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

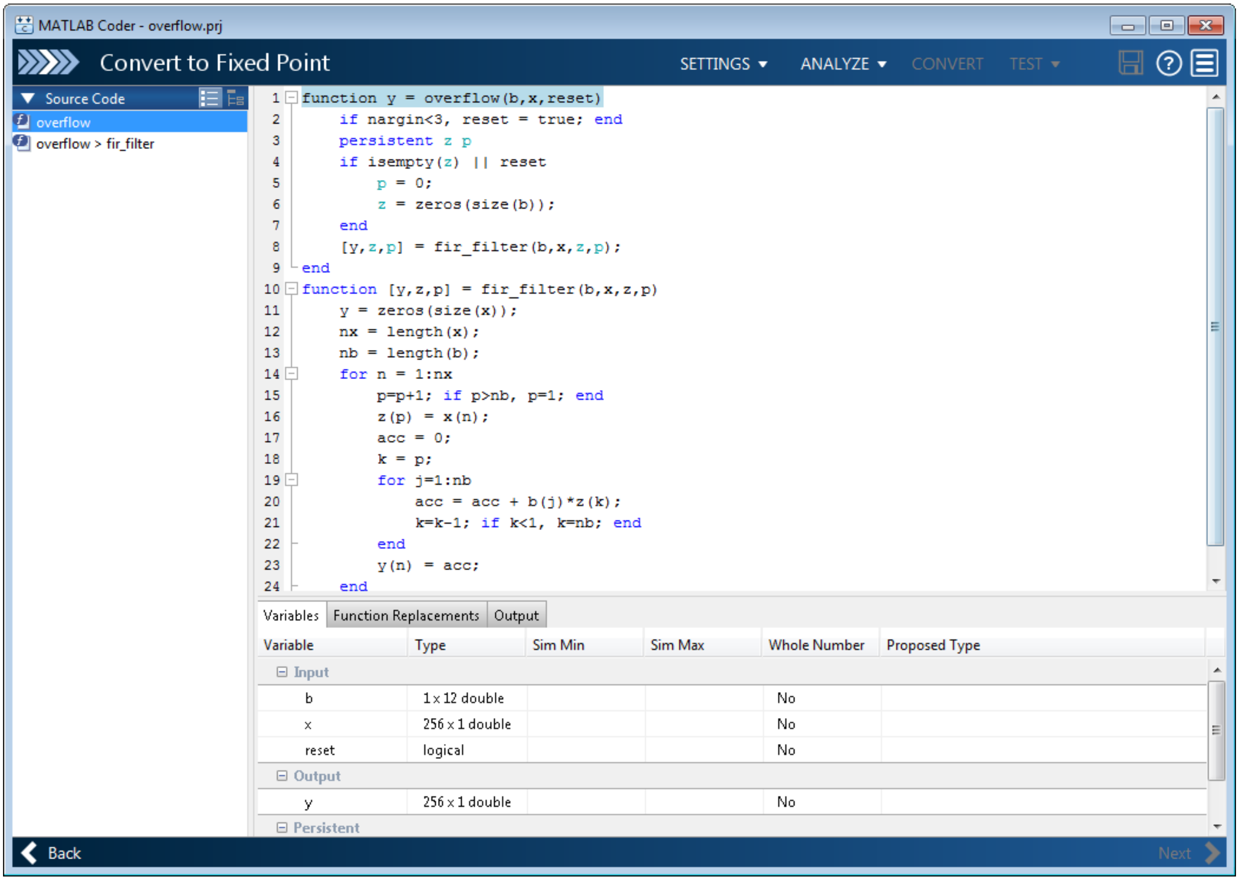
2 Click **Check for Issues**.

   The app does not detect issues.

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

Convert to Fixed Point

1 The app displays compiled information — type, size, and complexity — for variables in your code. For more information, see “View and Modify Variable Information” on page 14-79.

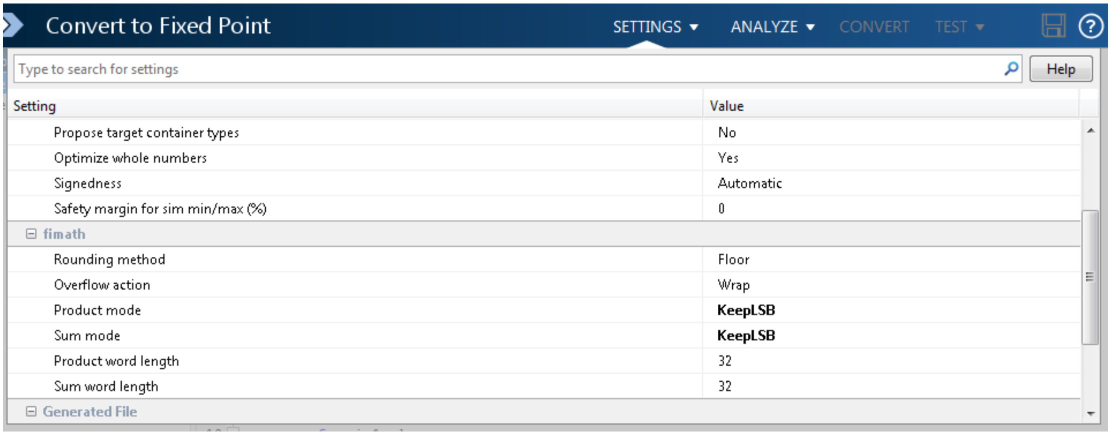


```
function y = overflow(b,x,reset)
    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
```

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

On the **Function Replacements** tab the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 14-88.

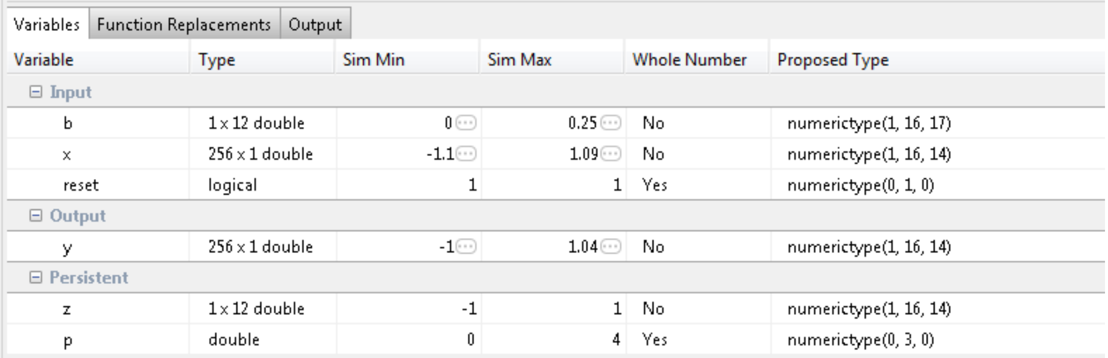
2 To view the fimath settings, click the **Settings** arrow ▾. Set the fimath **Product mode** and **Sum mode** to `KeepLSB`. These settings model the behavior of integer operations in the C language.



Setting	Value
Propose target container types	No
Optimize whole numbers	Yes
Signedness	Automatic
Safety margin for sim min/max (%)	0
⊟ fimath	
Rounding method	Floor
Overflow action	Wrap
Product mode	**KeepLSB**
Sum mode	**KeepLSB**
Product word length	32
Sum word length	32
⊟ Generated File	

3 Click **Analyze**.

The test file, `overflow_test`, runs. The app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.



Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
b	1 x 12 double	0	0.25	No	numerictype(1, 16, 17)
x	256 x 1 double	-1.1	1.09	No	numerictype(1, 16, 14)
reset	logical	1	1	Yes	numerictype(0, 1, 0)
⊟ Output					
y	256 x 1 double	-1	1.04	No	numerictype(1, 16, 14)
⊟ Persistent					
z	1 x 12 double	-1	1	No	numerictype(1, 16, 14)
p	double	0	4	Yes	numerictype(0, 3, 0)

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

The software validates the proposed types and generates a fixed-point version of the entry-point function.

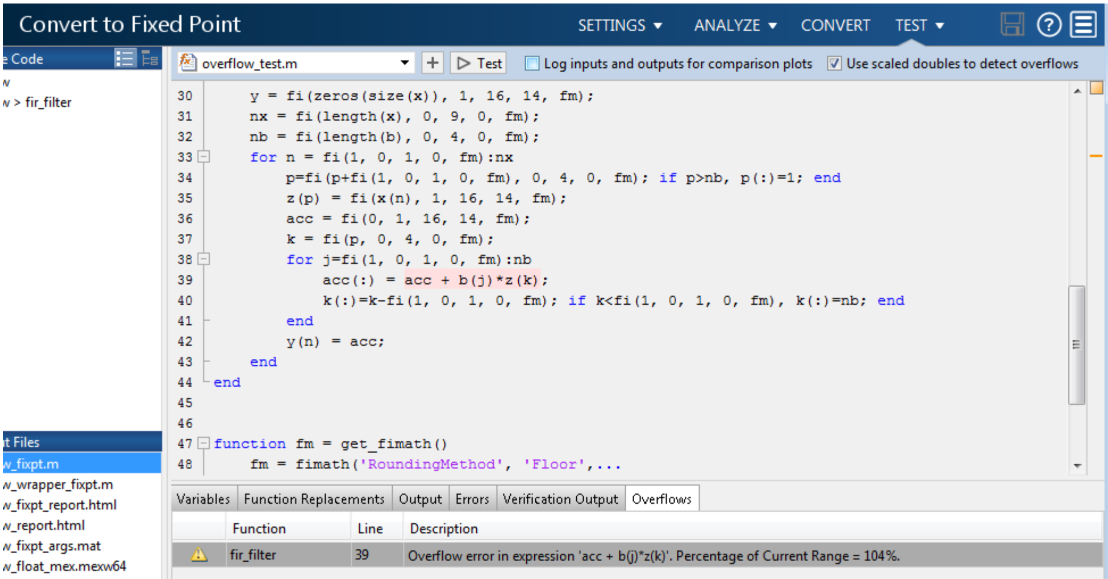
If errors and warnings occur during validation, the app displays them on the **Output** tab. See “Validating Types” on page 14-102.

Test Numerics and Check for Overflows

1 Click the **Test** arrow ▼. Verify that the test file is `overflow_test.m`. Select **Use scaled doubles to detect overflows**, and then click **Test**.

The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to detect overflows, it also runs the simulation using scaled double versions of the proposed fixed-point types. Scaled doubles store their data in double-precision floating-point, so they carry out arithmetic in full range. Because they retain their fixed-point settings, they can report when a computation goes out of the range of the fixed-point type.

The simulation runs. The app detects an overflow. The app reports the overflow on the **Overflow** tab. To highlight the expression that overflowed, click the overflow.



```
30      y = fi(zeros(size(x)), 1, 16, 14, fm);
31      nx = fi(length(x), 0, 9, 0, fm);
32      nb = fi(length(b), 0, 4, 0, fm);
33      for n = fi(1, 0, 1, 0, fm):nx
34          p=fi(p+fi(1, 0, 1, 0, fm), 0, 4, 0, fm); if p>nb, p(:)=1; end
35          z(p) = fi(x(n), 1, 16, 14, fm);
36          acc = fi(0, 1, 16, 14, fm);
37          k = fi(p, 0, 4, 0, fm);
38          for j=fi(1, 0, 1, 0, fm):nb
39              acc(:) = acc + b(j)*z(k);
40              k(:)=k-fi(1, 0, 1, 0, fm); if k<fi(1, 0, 1, 0, fm), k(:)=nb; end
41          end
42          y(n) = acc;
43      end
44  end
45
46
47  function fm = get_fimath()
48      fm = fimath('RoundingMethod', 'Floor',...
```

	Function	Line	Description
⚠	fir_filter	39	Overflow error in expression 'acc + b(j)*z(k)'. Percentage of Current Range = 104%.

2 Determine whether it was the sum or the multiplication that overflowed.

In the **fimath** settings, set **Product mode** to `FullPrecision`, and then repeat the conversion and test the fixed-point code again.

The overflow still occurs, indicating that it is the addition in the expression that is overflowing.

Replace the exp Function with a Lookup Table

This example shows how to replace the `exp` function with a lookup table approximation in fixed-point code generated using the MATLAB Coder app.

Prerequisites

To complete this example, you must install the following products:

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

  You can use `mex -setup` to change the default compiler. See "Change Default Compiler" (MATLAB).

Create Algorithm and Test Files

1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Open the MATLAB Coder App

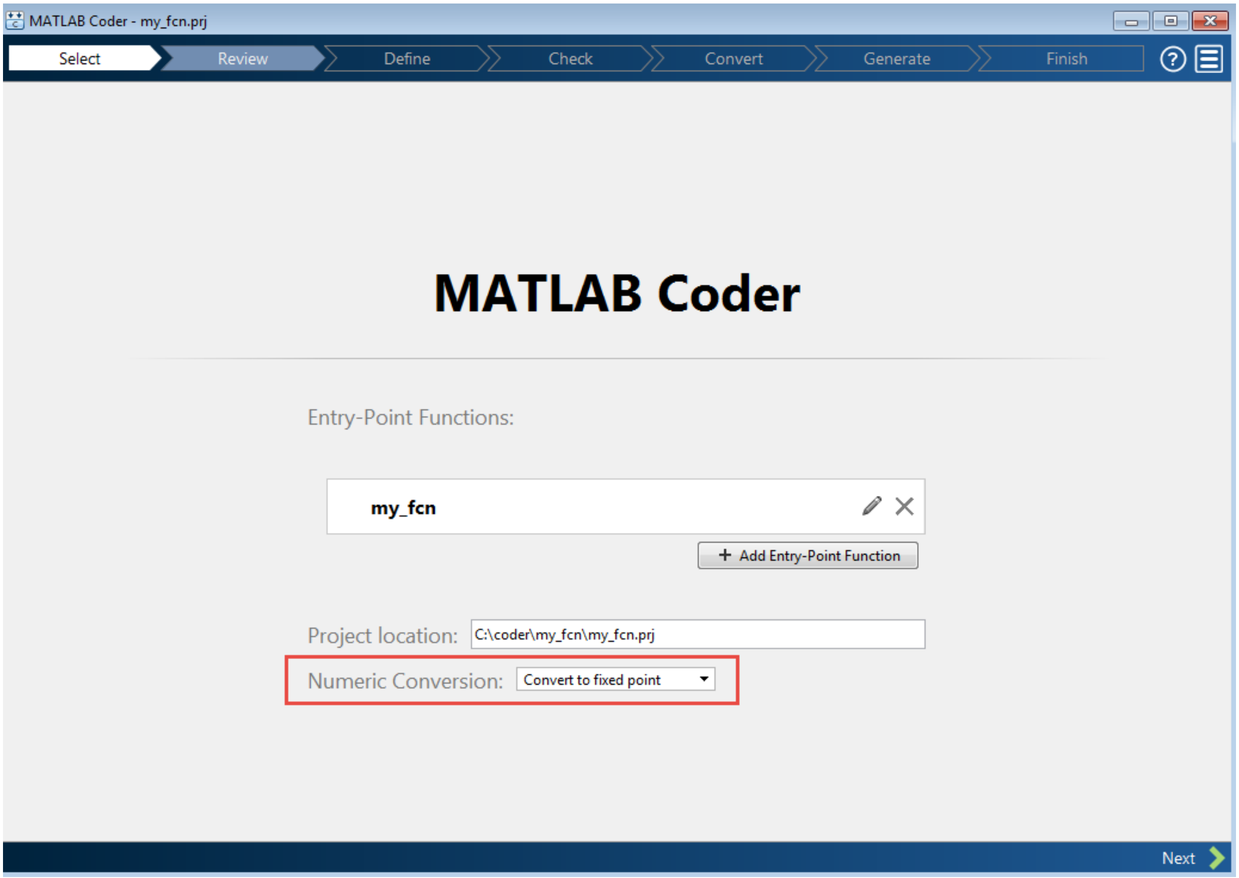
1 Navigate to the work folder that contains the file for this example.
2 On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Select Source Files

To add the entry-point function `my_fcn` to the project, browse to the file `my_fcn.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `my_fcn.prj`.

Enable Fixed-Point Conversion

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


MATLAB Coder - my_fcn.prj

MATLAB Coder

Entry-Point Functions:

my_fcn

Project location: C:\coder\my_fcn\my_fcn.prj

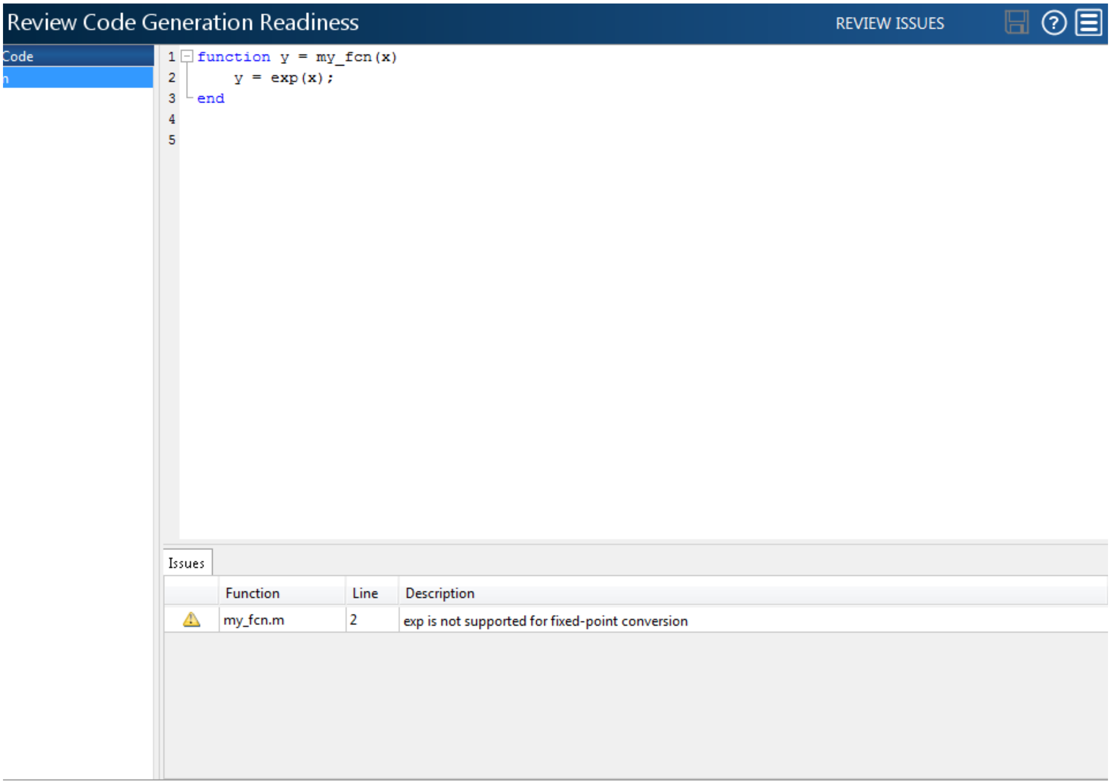
Numeric Conversion: Convert to fixed point

2 Click **Next** to go to the **Define Input Types** step.

The app screens `my_fcn.m` for code violations and code generation readiness issues. The app opens the **Review Code Generation Readiness** page.

Review Code Generation Readiness

1 Click **Review Issues**. The app indicates that the `exp` function is not supported for fixed-point conversion. In a later step, you specify a lookup table replacement for this function.



```
function y = my_fcn(x)
    y = exp(x);
end
```

	Function	Line	Description
⚠	my_fcn.m	2	exp is not supported for fixed-point conversion

2 Click **Next** to go to the **Define Input Types** step.

Define Input Types

1 Add `my_fcn_test` as a test file and then click **Autodefine Input Types**.

The test file runs. The app determines from the test file that `x` is a scalar double.

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

Check for Run-Time Issues

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

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

2 Click **Check for Issues**.

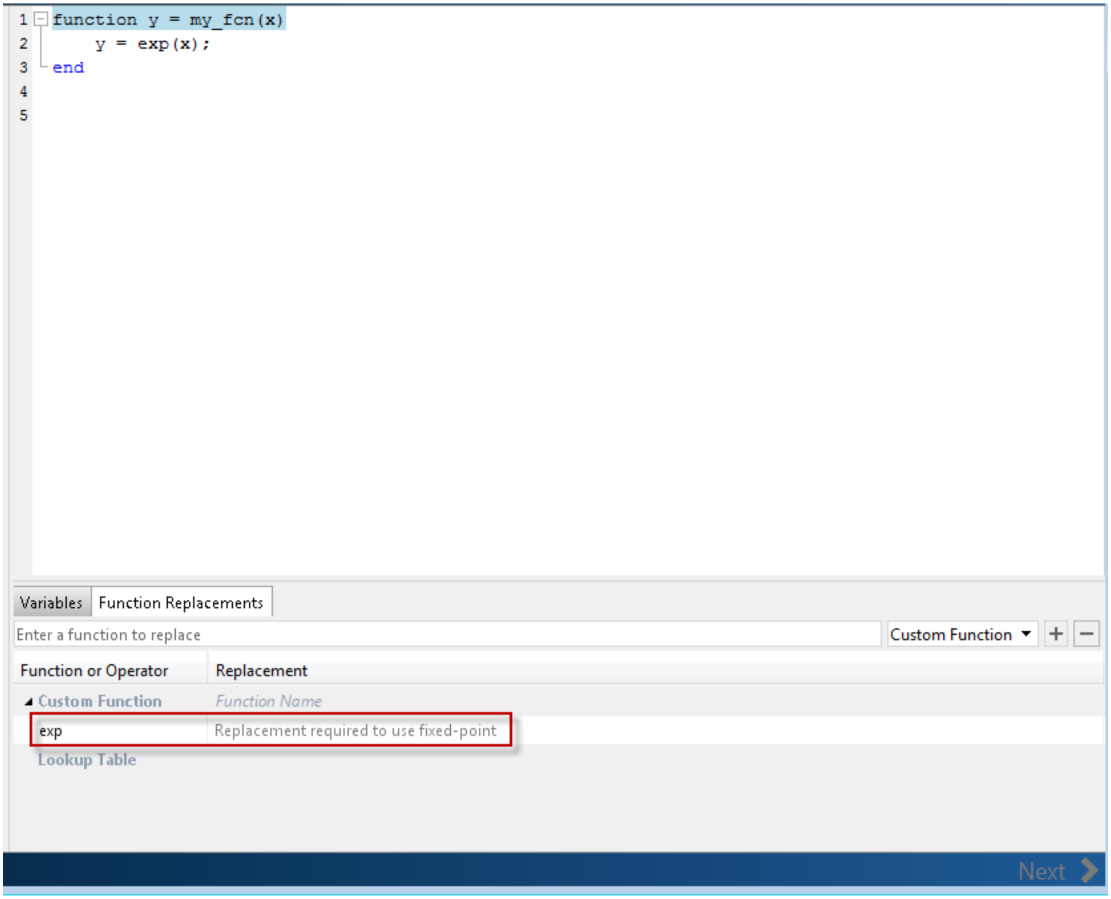
   The app does not detect issues.

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

Replace exp Function with Lookup Table

1 Select the **Function Replacements** tab.

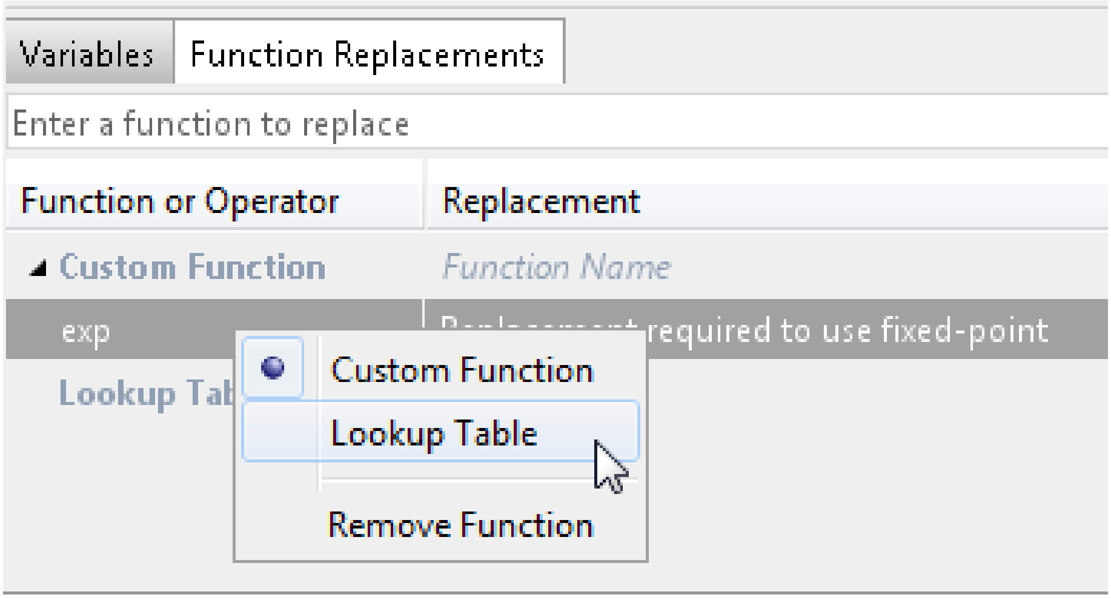
   The app indicates that you must replace the `exp` function.



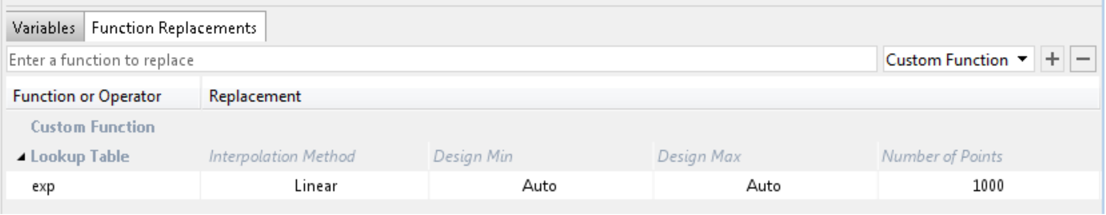
```
function y = my_fcn(x)
    y = exp(x);
end
```

Function or Operator	Replacement
Custom Function	Function Name
exp	Replacement required to use fixed-point

2 On the **Function Replacements** tab, right-click the `exp` function and select `Lookup Table`.

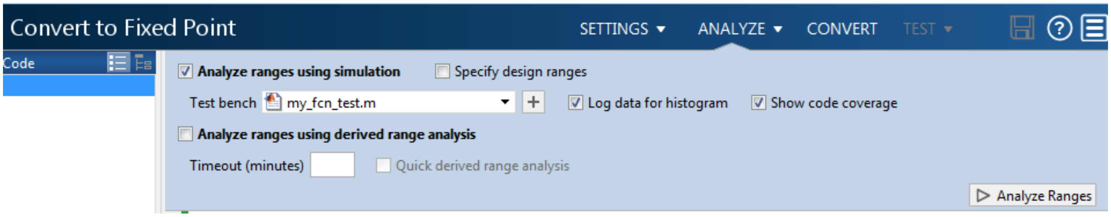


The app moves the `exp` function to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. **Design Min** and **Design Max** are set to `Auto` which means that the app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.



Function or Operator	Replacement			
Custom Function				
Lookup Table	Interpolation Method	Design Min	Design Max	Number of Points
exp	Linear	Auto	Auto	1000

3 Click the **Analyze** arrow ▼, select **Log data for histogram**, and verify that the test file is `my_fcn_test`.



☑ **Analyze ranges using simulation** □ Specify design ranges

Test bench my_fcn_test.m ☑ Log data for histogram ☑ Show code coverage

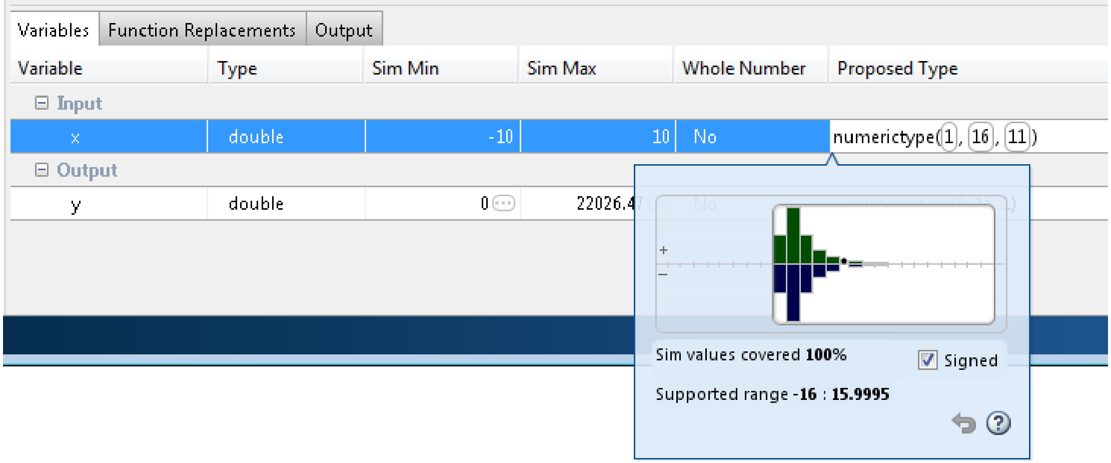
□ **Analyze ranges using derived range analysis**

Timeout (minutes) □ Quick derived range analysis

4 Click **Analyze**.

The simulation runs. On the **Variables** tab, the app displays simulation minimum and maximum ranges. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The app enables the **Convert** option.

5 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field. The histogram provides range information and the percentage of simulation range covered by the proposed data type.



Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
x	double	-10	10	No	numerictype(1, 16, 11)
⊟ Output					
y	double	0	22026.4		

Sim values covered **100%** ☑ Signed

Supported range **-16 : 15.9995**

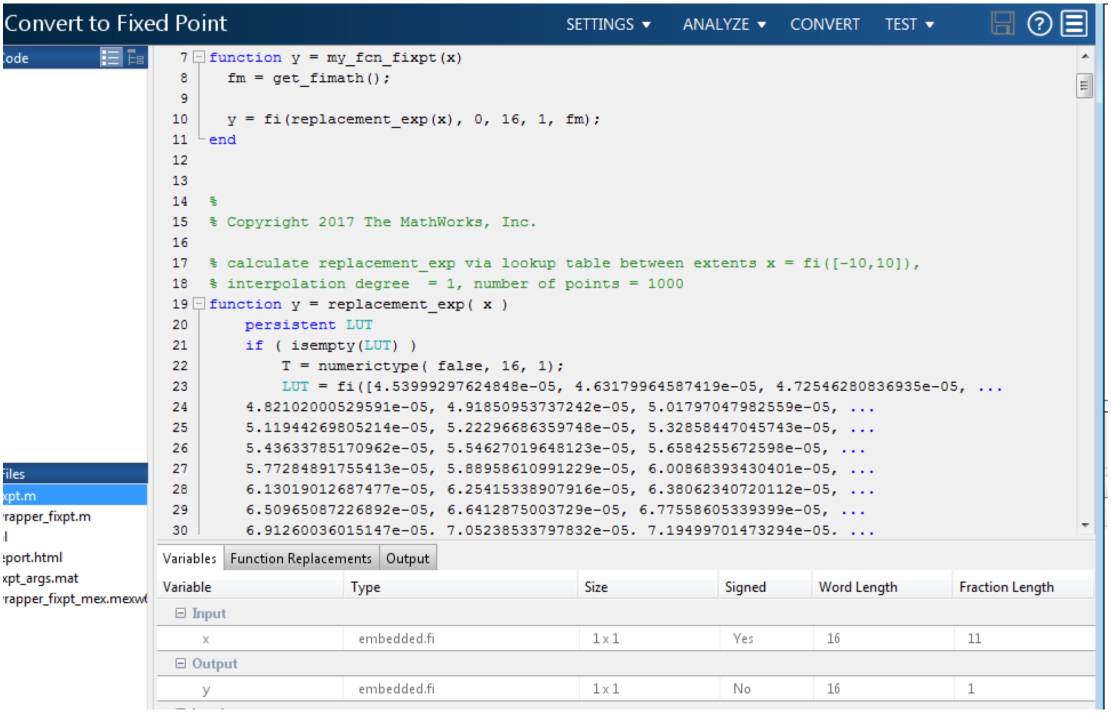
Convert to Fixed Point

1 Click **Convert**.

The app validates the proposed types, and generates a fixed-point version of the entry-point function, `my_fcn_fixpt.m`.

2 In the Output Files list, select `my_fcn_fixpt.m`.

The conversion process generates a lookup table approximation, `replacement_exp`, for the `exp` function.



```
function y = my_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_exp(x), 0, 16, 1, fm);
end


%
% Copyright 2017 The MathWorks, Inc.

% calculate replacement_exp via lookup table between extents x = fi([-10,10]),
% interpolation degree  = 1, number of points = 1000
function y = replacement_exp( x )
    persistent LUT
    if ( isempty(LUT) )
        T = numerictype( false, 16, 1);
        LUT = fi([4.53999297624848e-05, 4.63179964587419e-05, 4.72546280836935e-05, ...
    4.82102000529591e-05, 4.91850953737242e-05, 5.01797047982559e-05, ...
    5.11944269805214e-05, 5.22296686359748e-05, 5.32858447045743e-05, ...
    5.43633785170962e-05, 5.54627019648123e-05, 5.6584255672598e-05, ...
    5.77284891755413e-05, 5.88958610991229e-05, 6.00868393430401e-05, ...
    6.13019012687477e-05, 6.25415338907916e-05, 6.38062340720112e-05, ...
    6.50965087226892e-05, 6.6412875003729e-05, 6.77558605339399e-05, ...
    6.91260036015147e-05. 7.05238533797832e-05. 7.19499701473294e-05. ...
```

Variables | Function Replacements | Output

Variable	Type	Size	Signed	Word Length	Fraction Length
⊟ Input					
x	embedded.fi	1 x 1	Yes	16	11
⊟ Output					
y	embedded.fi	1 x 1	No	16	1

The generated fixed-point function, `my_fcn_fixpt.m`, calls this approximation instead of calling `exp`. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

```
function y = my_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_exp(x), 0, 16, 1, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table. Then, regenerate the code.

Replace a Custom Function with a Lookup Table

This example shows how to replace a custom function with a lookup table approximation function using the MATLAB Coder app.

Prerequisites

This example requires the following products:

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

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

Create Algorithm and Test Files

In a local, writable folder:

1. Create a MATLAB function, `custom_fcn.m` which is the function that you want to replace.

   ```
   function y = custom_fcn(x)
       y = 1./(1+exp(-x));
   end
   ```
2. Create a wrapper function, `call_custom_fcn.m`, that calls `custom_fcn.m`.

   ```
   function y = call_custom_fcn(x)
       y = custom_fcn(x);
   end
   ```
3. Create a test file, `custom_test.m`, that uses `call_custom_fcn`.

   ```
   close all
   clear all

   x = linspace(-10,10,1e3);
   for itr = 1e3:-1:1
      y(itr) = call_custom_fcn( x(itr) );
   ```

```
    end
    plot( x, y );
```

Open the MATLAB Coder App

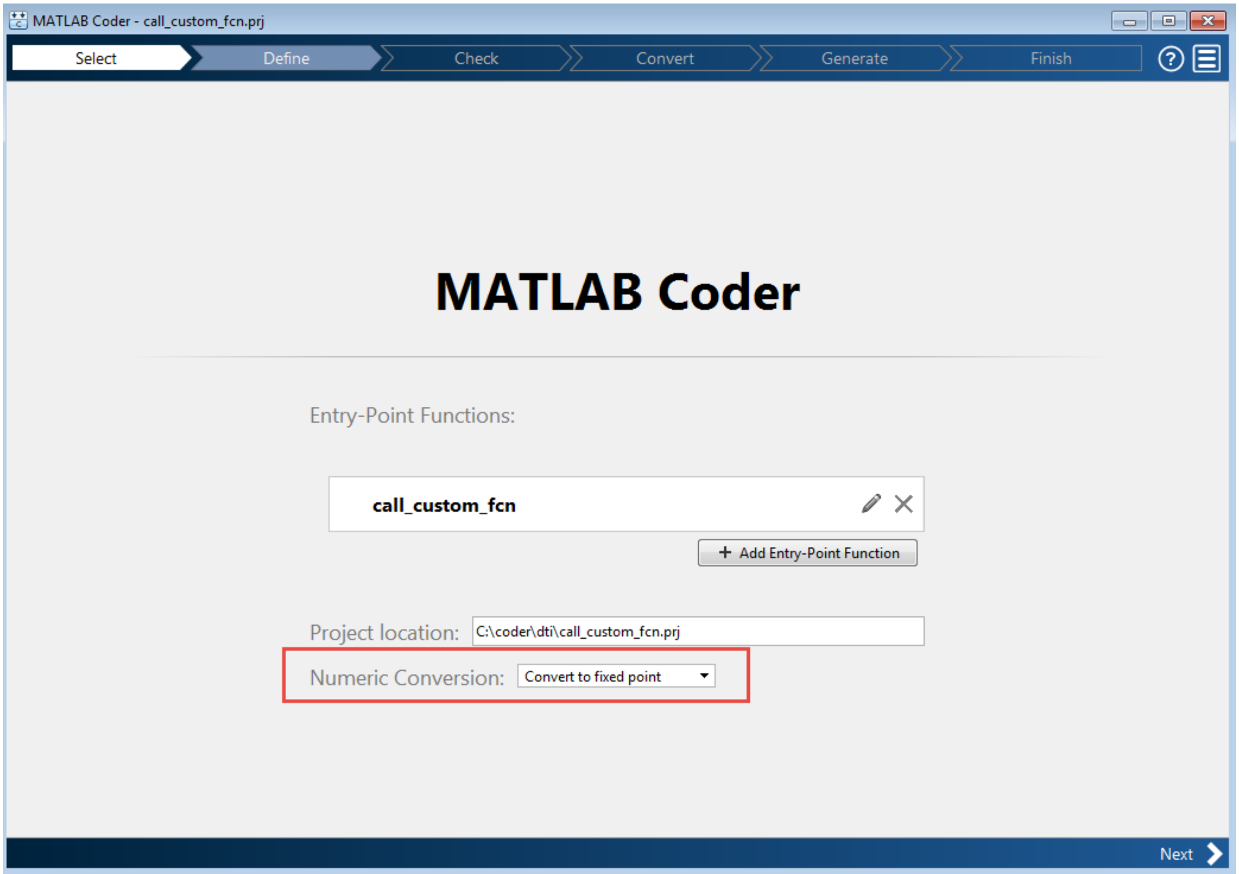
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Select Source Files

To add the entry-point function `call_custom_fcn` to the project, browse to the file `call_custom_fcn.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `call_custom_fcn.prj`.

Enable Fixed-Point Conversion

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


MATLAB Coder - call_custom_fcn.prj
Select
Define
Check
Convert
Generate
Finish
MATLAB Coder
+ Add Entry-Point Function
Next

Entry-Point Functions: call_custom_fcn

Project location: C:\coder\dti\call_custom_fcn.prj

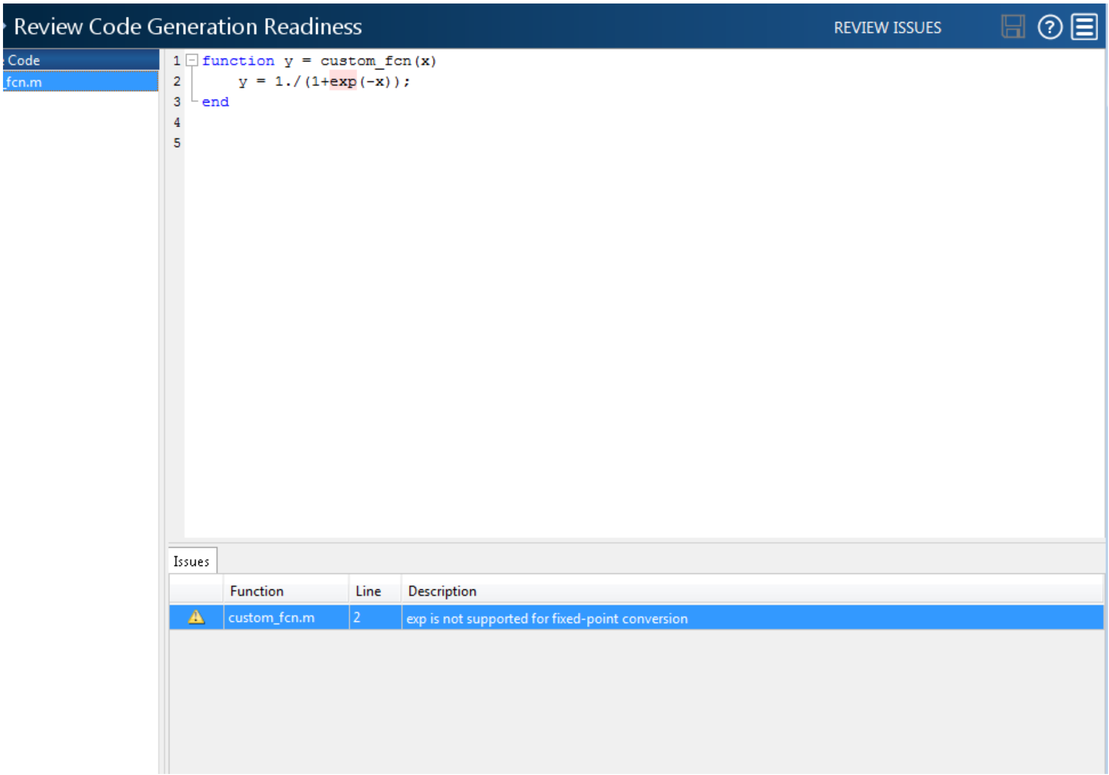
Numeric Conversion: Convert to fixed point

2 Click **Next** to go to the **Define Input Types** step.

The app screens `call_custom_fcn.m` for code violations and code generation issues. The app opens the **Review Code Generation Readiness** page.

Review Code Generation Readiness

1 Click **Review Issues**. The app indicates that the `exp` function is not supported for fixed-point conversion. You can ignore this warning because you are going to replace `custom_fcn`, which is the function that calls `exp`.



```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

	Function	Line	Description
⚠	custom_fcn.m	2	exp is not supported for fixed-point conversion

2 Click **Next** to go to the **Define Input Types** step.

Define Input Types

1 Add `custom_test` as a test file and then click **Autodefine Input Types**.

The test file runs. The app determines from the test file that `x` is a scalar double.

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

Check for Run-Time Issues

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

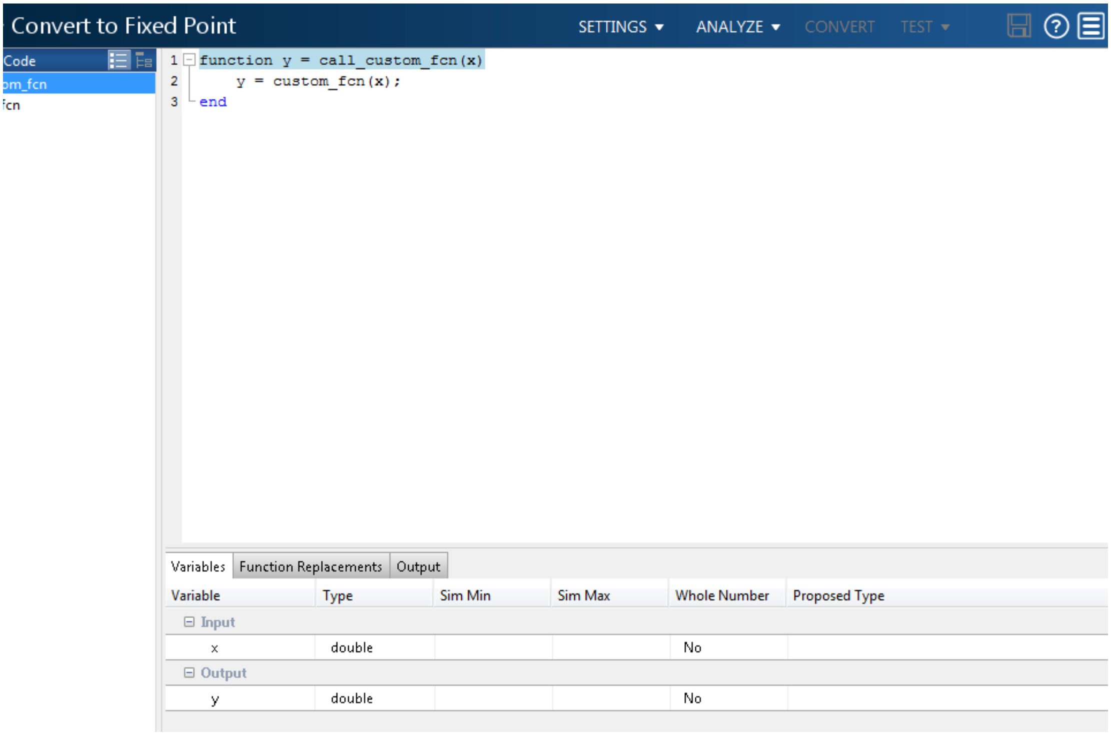
1 On the **Check for Run-Time Issues** page, the app populates the test file field with `custom_test`, the test file that you used to define the input types.
2 Click **Check for Issues**.

  The app does not detect issues.
3 Click **Next** to go to the **Convert to Fixed Point** step.

Replace custom_fcn with Lookup Table

1 Select the **Function Replacements** tab.

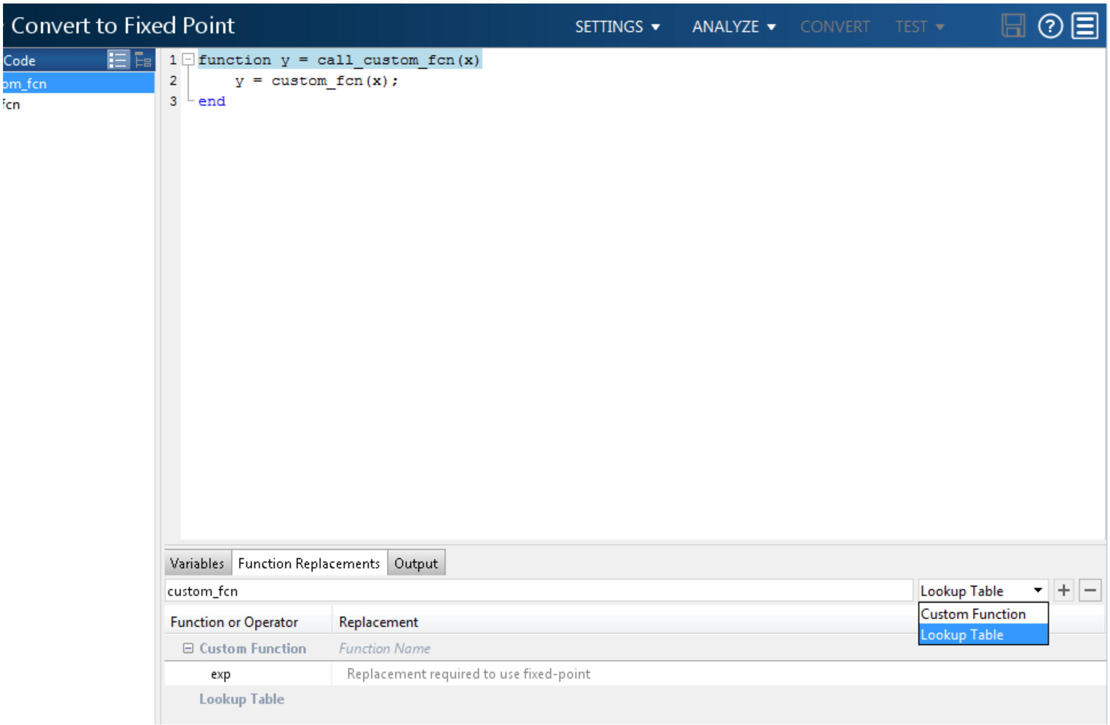
  The app indicates that you must replace the `exp` function.



```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
x	double			No	
⊟ Output					
y	double			No	

2 Enter the name of the function to replace, `custom_fcn`, select `Lookup Table`, and then click +.

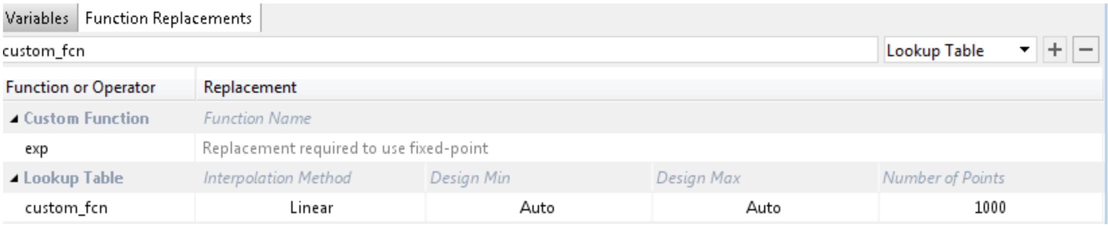


```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

custom_fcn

Function or Operator	Replacement
⊟ Custom Function	Function Name
exp	Replacement required to use fixed-point
Lookup Table	

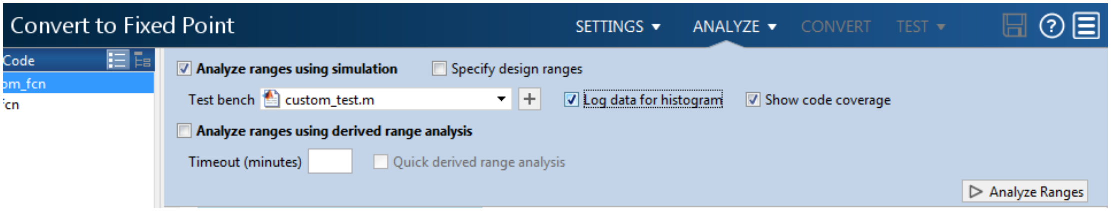
The app adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. The app sets **Design Min** and **Design Max** to `Auto` which means that app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.



custom_fcn

Function or Operator	Replacement			
◢ Custom Function	Function Name			
exp	Replacement required to use fixed-point			
◢ Lookup Table	Interpolation Method	Design Min	Design Max	Number of Points
custom_fcn	Linear	Auto	Auto	1000

3 Click the **Analyze** arrow ▼, select **Log data for histogram**, and verify that the test file is `call_custom_test`.



☑ Analyze ranges using simulation □ Specify design ranges

Test bench custom_test.m ☑ Log data for histogram ☑ Show code coverage

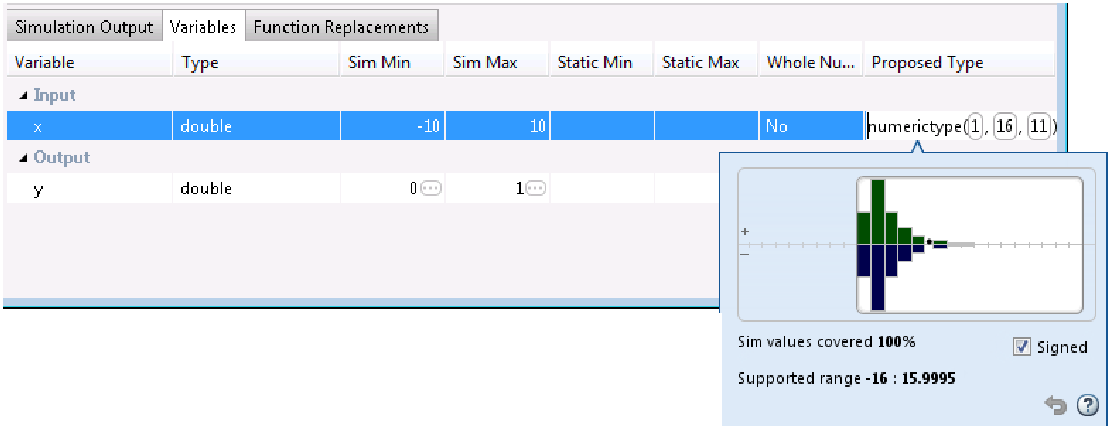
□ Analyze ranges using derived range analysis

Timeout (minutes) □ Quick derived range analysis

4 Click **Analyze**.

The simulation runs. The app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column. The **Convert** option is now enabled.

5 Examine the proposed types and verify that they cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field. The histogram provides range information and the percentage of simulation range covered by the proposed data type.



Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
x	double	-10	10			No	numerictype(1, 16, 11)
Output							
y	double	0	1				

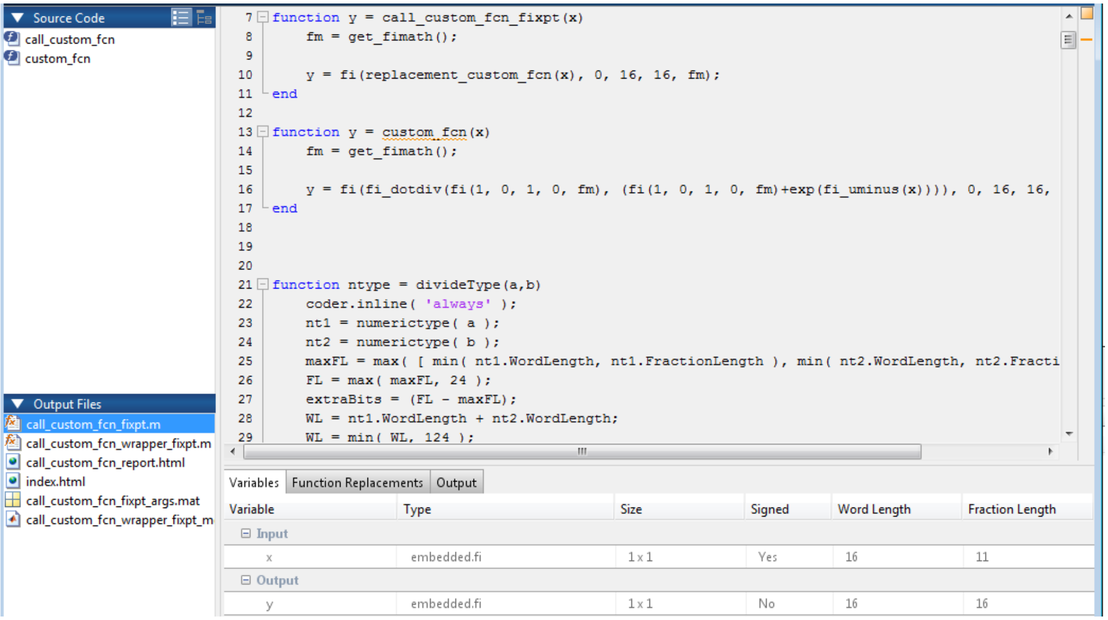
Sim values covered **100%** ☑ Signed

Supported range **-16 : 15.9995**

Convert to Fixed Point

1 Click **Convert**.

The app validates the proposed types and generates a fixed-point version of the entry-point function, `call_custom_fcn_fixpt.m`.


Source Code
call_custom_fcn
custom_fcn
Output Files
call_custom_fcn_fixpt.m
call_custom_fcn_wrapper_fixpt.m
call_custom_fcn_report.html
index.html
call_custom_fcn_fixpt_args.mat
call_custom_fcn_wrapper_fixpt_m
Variables
Function Replacements
Output

```
function y = call_custom_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_custom_fcn(x), 0, 16, 16, fm);
end

function y = custom_fcn(x)
    fm = get_fimath();

    y = fi(fi_dotdiv(fi(1, 0, 1, 0, fm), (fi(1, 0, 1, 0, fm)+exp(fi_uminus(x)))), 0, 16, 16,
end



function ntype = divideType(a,b)
    coder.inline( 'always' );
    nt1 = numerictype( a );
    nt2 = numerictype( b );
    maxFL = max( [ min( nt1.WordLength, nt1.FractionLength ), min( nt2.WordLength, nt2.Fracti
    FL = max( maxFL, 24 );
    extraBits = (FL - maxFL);
    WL = nt1.WordLength + nt2.WordLength;
    WL = min( WL, 124 );
```

Variable	Type	Size	Signed	Word Length	Fraction Length
⊟ Input					
x	embedded.fi	1 x 1	Yes	16	11
⊟ Output					
y	embedded.fi	1 x 1	No	16	16

2 In the Output Files list, select `call_custom_fcn_fixpt.m`.

The conversion process generates a lookup table approximation, `replacement_custom_fcn`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_custom_fcn(x), 0, 16, 16, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

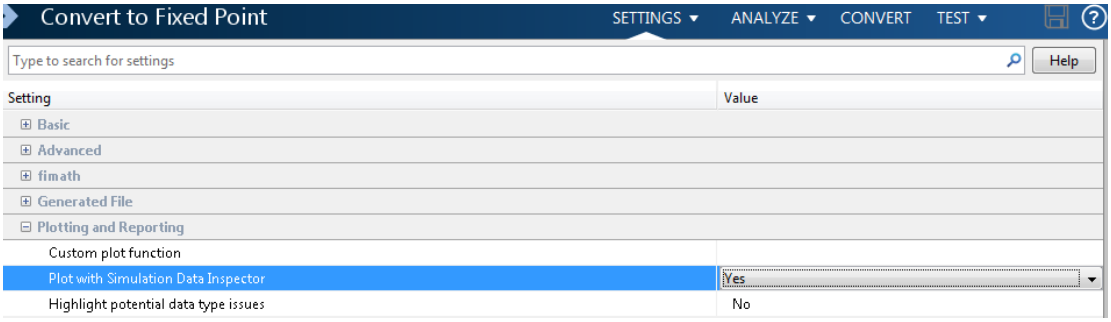
Enable Plotting Using the Simulation Data Inspector

You can use the Simulation Data Inspector with the MATLAB Coder app to inspect and compare floating-point and fixed-point logged input and output data.

1 On the **Convert to Fixed Point** page,

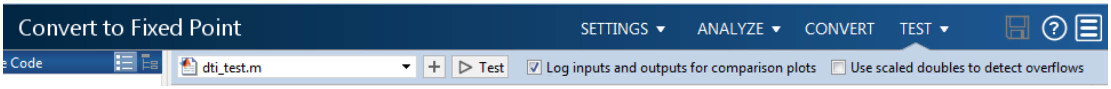
   Click the **Settings** arrow ▼.

2 Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to `Yes`.



Setting	Value
⊞ Basic	
⊞ Advanced	
⊞ fimath	
⊞ Generated File	
⊟ Plotting and Reporting	
Custom plot function	
Plot with Simulation Data Inspector	Yes
Highlight potential data type issues	No

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



☑ Log inputs and outputs for comparison plots ☐ Use scaled doubles to detect overflows

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges” on page 14-21“Propose Data Types Based on Derived Ranges” (Fixed-Point Designer).

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the MATLAB Coder app to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the **Log inputs and outputs for comparison plots** option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

Prerequisites

This example requires the following products:

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

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	`myFilter.m`	Entry-point MATLAB function

Type	Name	Description
Test file	myFilterTest.m	MATLAB script that tests myFilter.m
Plotting function	plotDiff.m	Custom plot function
MAT-file	filterData.mat	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
 y = myFilter(d(idx));
end
```

The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%             i) name
%            ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after
%             Fixed-Point Conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexprep(varName,'_','\\_');
    escapedFcnName = regexprep(fcnName,'_','\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

            % plot fixed point values
            y_vec = flatFixedVals;
            subplot(1, 2, 2);
            plotScatter(x_vec, y_vec, 100, fixedTitle);

        otherwise
            % Plot only output 'y' for this example, skip the rest
    end

end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');

    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

Open the MATLAB Coder App

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

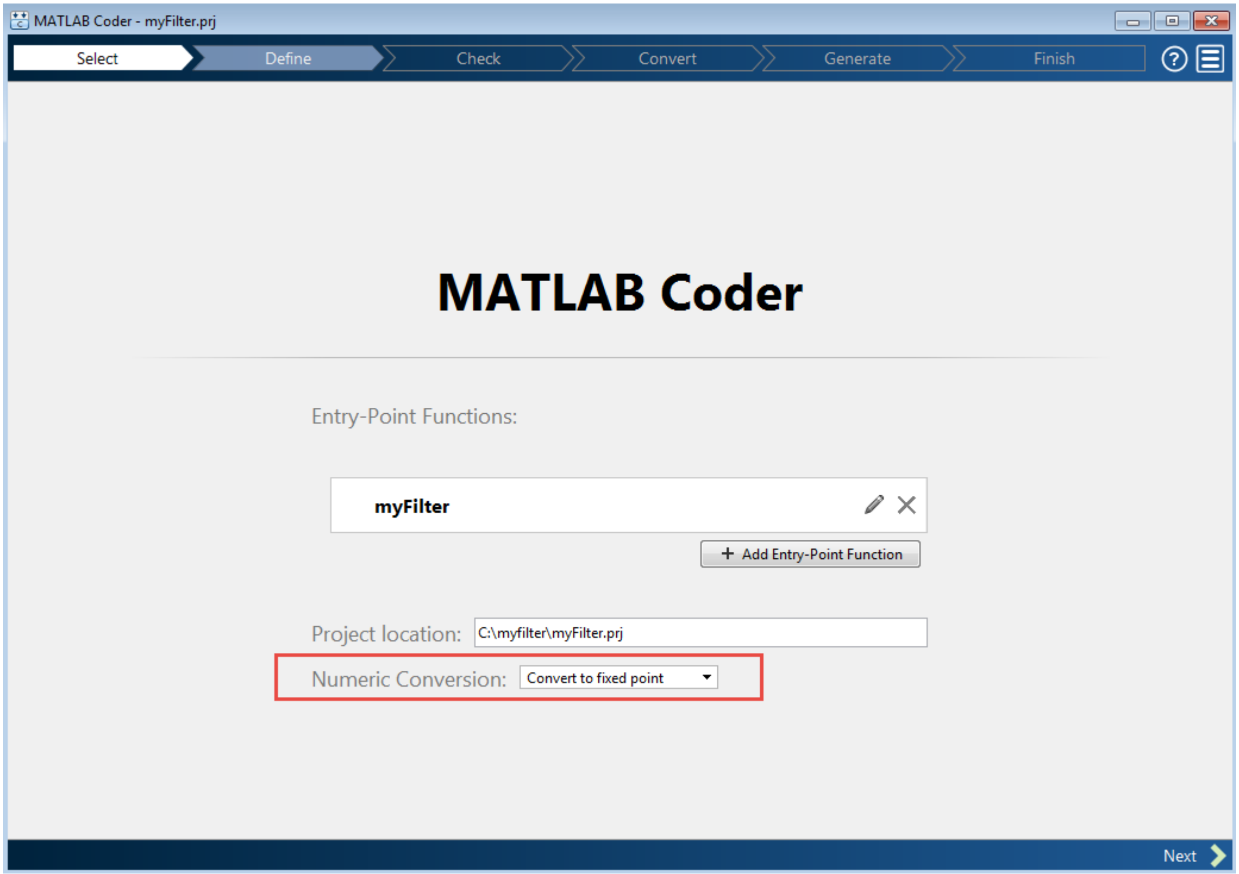
Select Source Files

To add the entry-point function `myFilter` to the project, browse to the file `myFilter.m`, and then click **Open**.

By default, the app saves information and settings for this project in the current folder in a file named `myFilter.prj`.

Enable Fixed-Point Conversion

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



Entry-Point Functions:

myFilter

Project location: C:\myfilter\myFilter.prj

Numeric Conversion: Convert to fixed point

2 Click **Next** to go to the **Define Input Types** step.

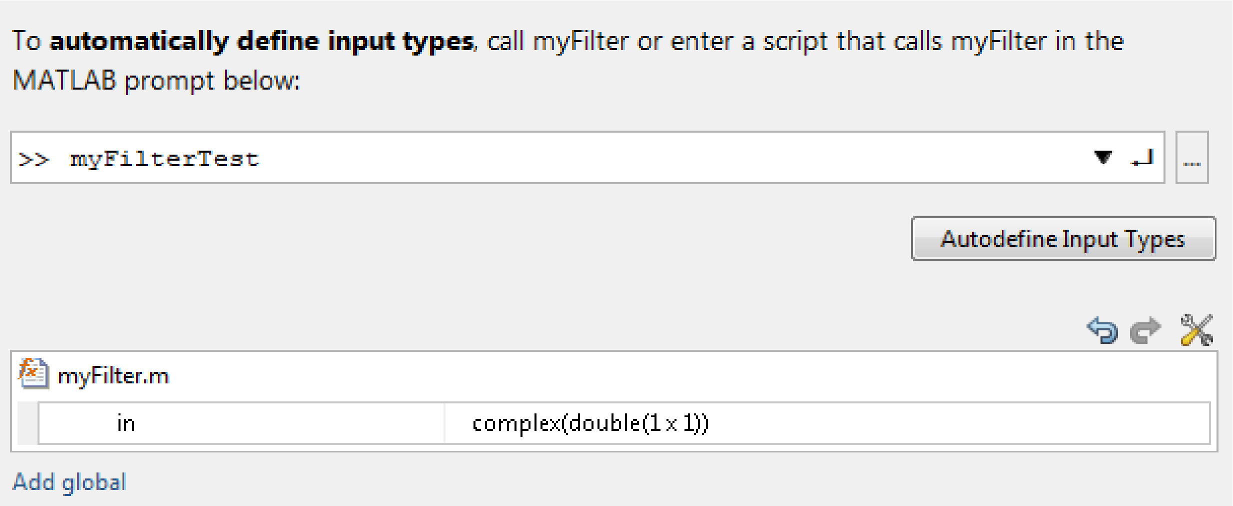
The app screens `myFilter.m` for code violations and code generation readiness issues. The app does not find issues in `myFilter.m`.

Define Input Types

1 On the **Define Input Types** page, to add `myFilterTest` as a test file, browse to `myFilterTest.m`, and then click **Open**.

2 Click **Autodefine Input Types**.

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



To **automatically define input types**, call myFilter or enter a script that calls myFilter in the MATLAB prompt below:

```
>> myFilterTest
```

myFilter.m

in	complex(double(1 x 1))

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

Check for Run-Time Issues

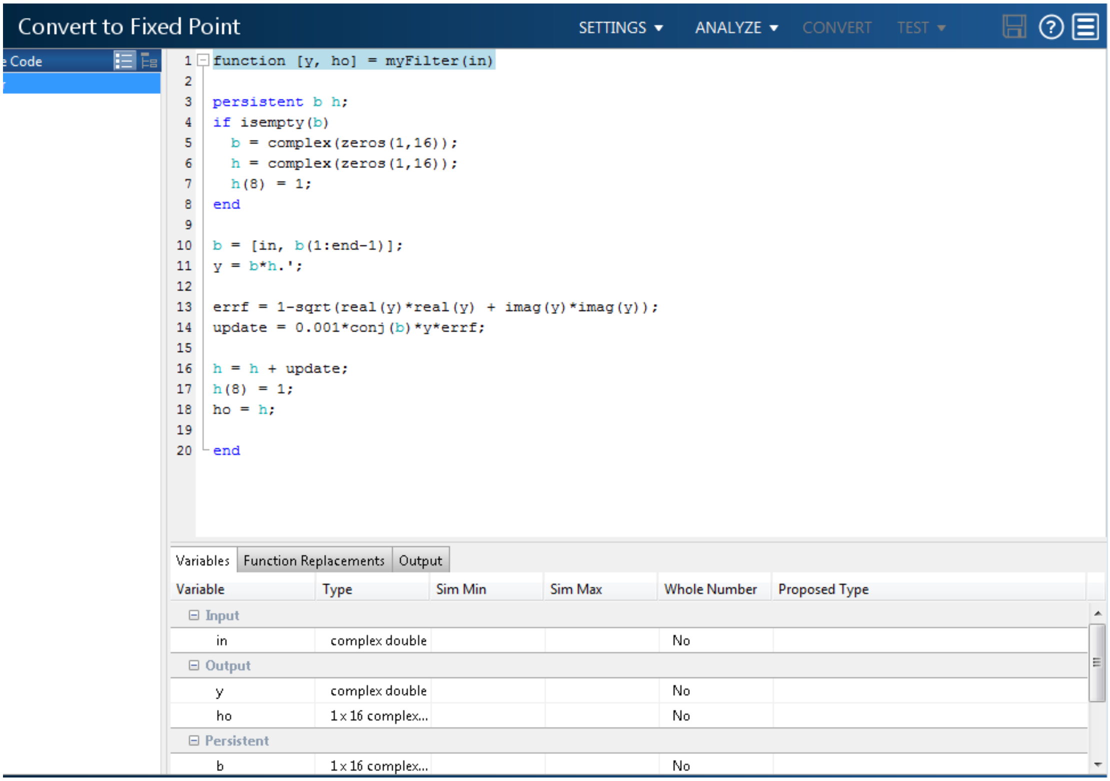
The **Check for Run-Time Issues** step generates instrumented MEX. `myFilter`. It runs the test file `myFilterTest` replacing calls to `myFilter` with calls to the generated MEX. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code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

1. Browse to the test file `myFiltertest.m`.
2. Click **Check for Issues**.

   The app does not detect issues.
3. Click **Next** to go to the **Convert to Fixed Point** step.

Convert to Fixed Point

1. The app displays compiled information for variables in your code. For more information, see “View and Modify Variable Information” on page 14-79“View and Modify Variable Information” (Fixed-Point Designer).



```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
in	complex double			No	
⊟ Output					
y	complex double			No	
ho	1 x 16 complex...			No	
⊟ Persistent					
b	1 x 16 complex...			No	

2 To open the settings dialog box, click the **Settings** arrow ▾.

   a Verify that **Default word length** is set to `16`.

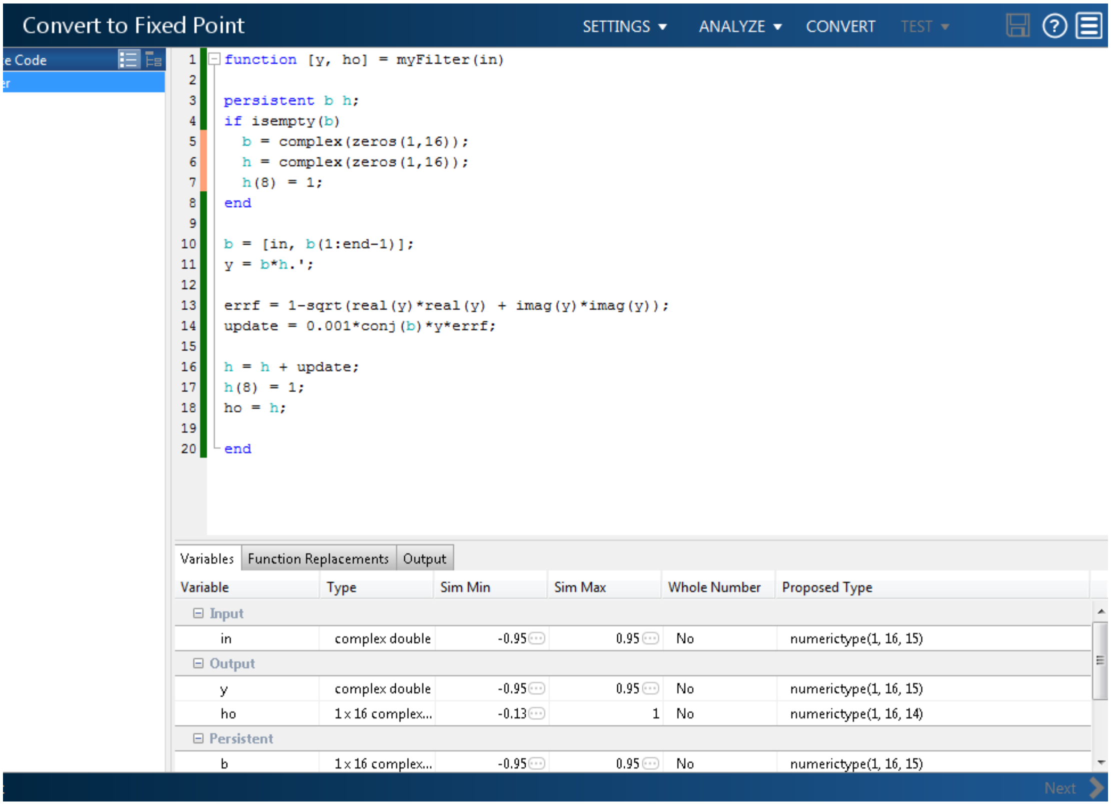
   b Under **Advanced**, set **Signedness** to `Signed`

   c Under **Plotting and Reporting**, set **Custom plot function** to `plotDiff`.

3 Click the **Analyze** arrow ▾. Verify that the test file is `myFilterTest`.

4 Click **Analyze**.

The test file, `myFilterTest`, runs and the app displays simulation minimum and maximum ranges on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

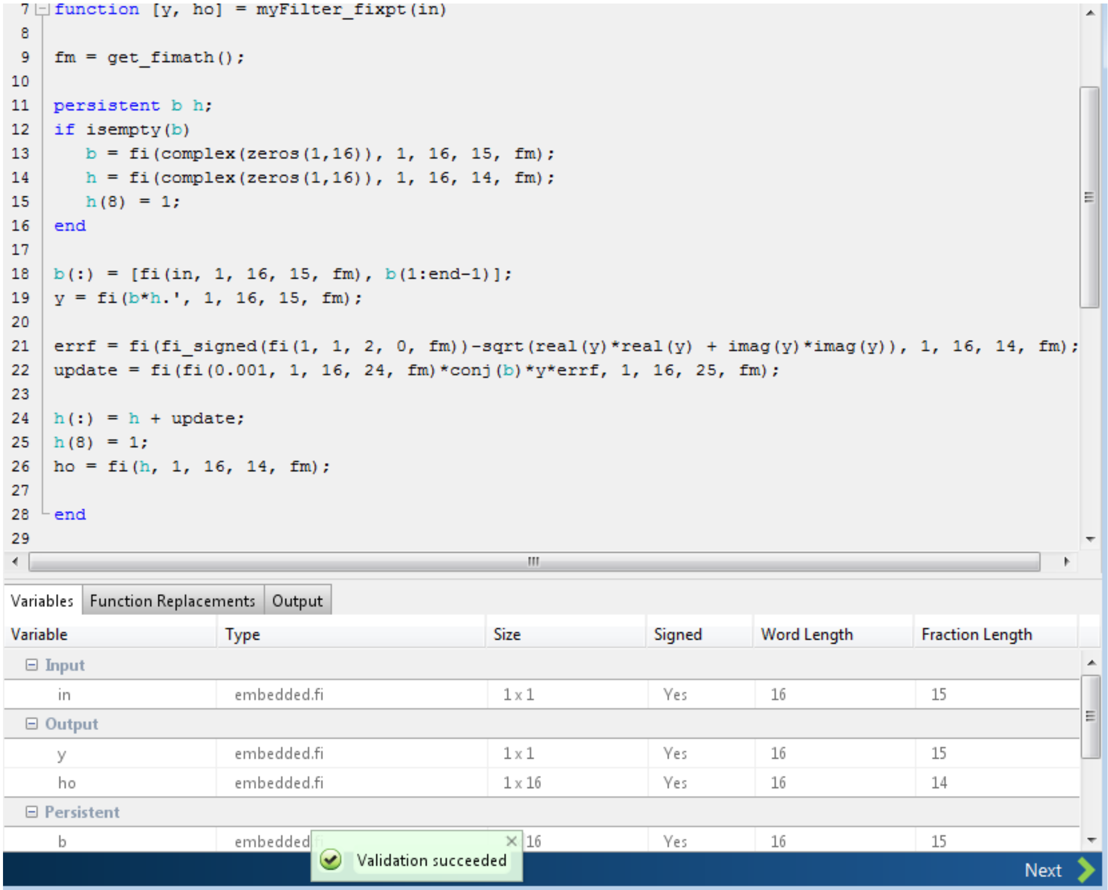


```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
in	complex double	-0.95	0.95	No	numerictype(1, 16, 15)
⊟ Output					
y	complex double	-0.95	0.95	No	numerictype(1, 16, 15)
ho	1 x 16 complex...	-0.13	1	No	numerictype(1, 16, 14)
⊟ Persistent					
b	1 x 16 complex...	-0.95	0.95	No	numerictype(1, 16, 15)

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

The software validates the proposed types and generates a fixed-point version of the entry-point function.



```
function [y, ho] = myFilter_fixpt(in)

fm = get_fimath();

persistent b h;
if isempty(b)
    b = fi(complex(zeros(1,16)), 1, 16, 15, fm);
    h = fi(complex(zeros(1,16)), 1, 16, 14, fm);
    h(8) = 1;
end

b(:) = [fi(in, 1, 16, 15, fm), b(1:end-1)];
y = fi(b*h.', 1, 16, 15, fm);

errf = fi(fi_signed(fi(1, 1, 2, 0, fm))-sqrt(real(y)*real(y) + imag(y)*imag(y)), 1, 16, 14, fm);
update = fi(fi(0.001, 1, 16, 24, fm)*conj(b)*y*errf, 1, 16, 25, fm);

h(:) = h + update;
h(8) = 1;
ho = fi(h, 1, 16, 14, fm);

end
```

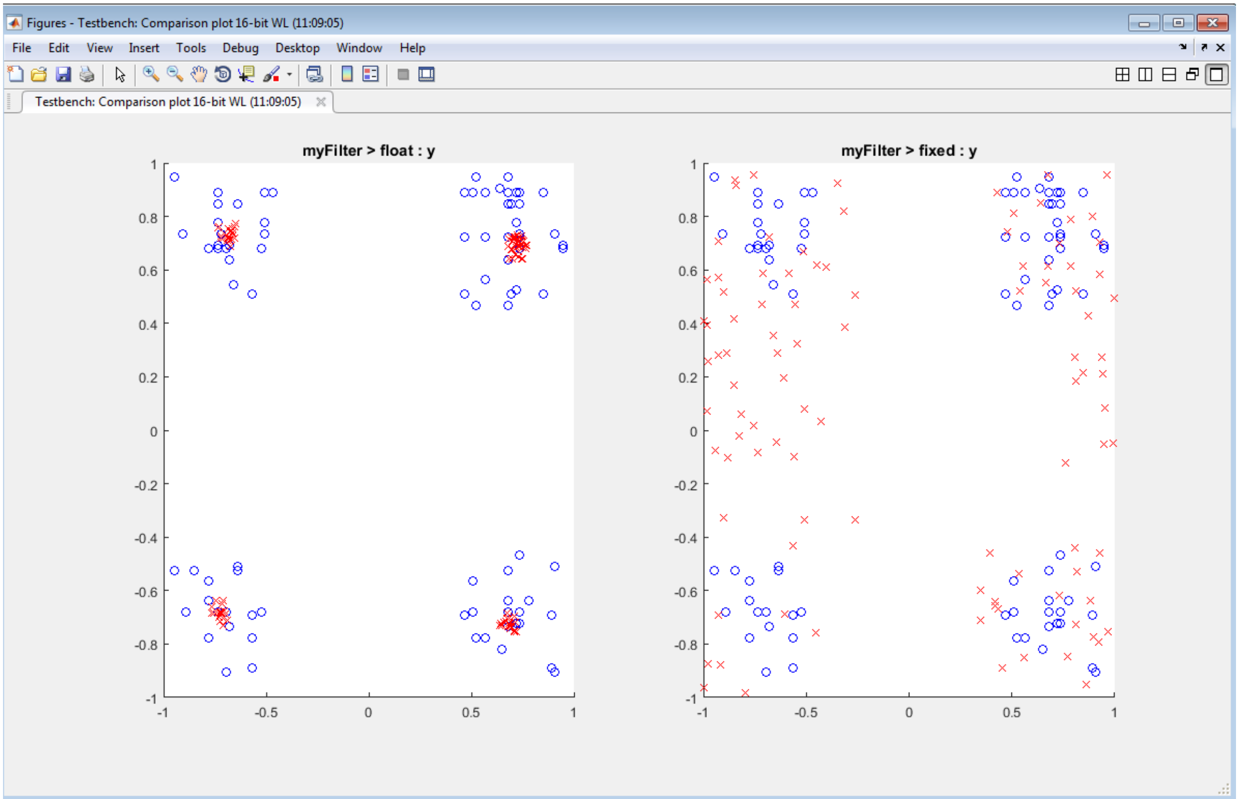
Variable	Type	Size	Signed	Word Length	Fraction Length
⊟ Input					
in	embedded.fi	1 x 1	Yes	16	15
⊟ Output					
y	embedded.fi	1 x 1	Yes	16	15
ho	embedded.fi	1 x 16	Yes	16	14
⊟ Persistent					
b	embedded	16	Yes	16	15

✓ Validation succeeded

Test Numerics and View Comparison Plots

1 Click **Test** arrow ▼, select **Log inputs and outputs for comparison plots**, and then click **Test**.

The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the app uses this function to generate the comparison plot. The plot shows that the fixed-point results do not closely match the floating-point results.

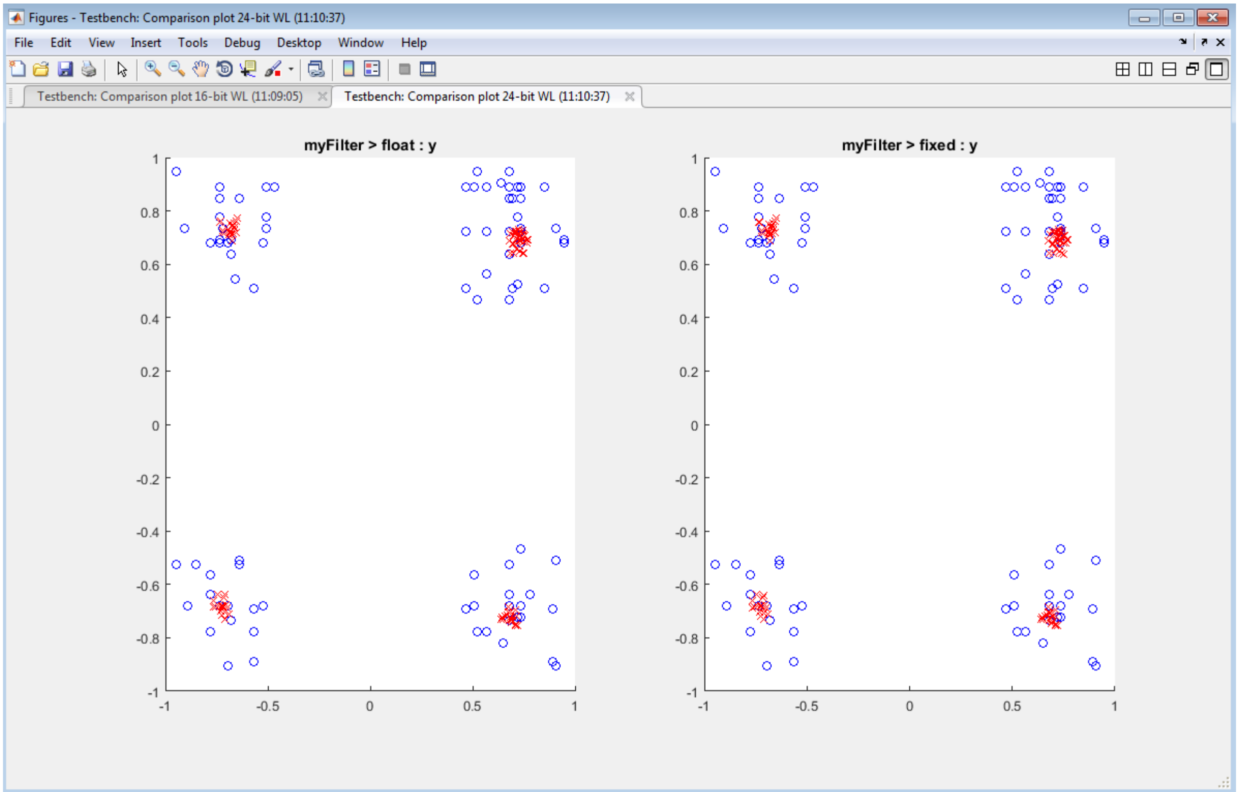

Figures - Testbench: Comparison plot 16-bit WL (11:09:05)
File Edit View Insert Tools Debug Desktop Window Help
Testbench: Comparison plot 16-bit WL (11:09:05)
myFilter > float : y
myFilter > fixed : y
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1

2 In the settings, increase the **DefaultWordLength** to 24 and then convert to fixed point again.

The app converts `myFilter.m` to fixed point and proposes fixed-point data types using the new default word length.

3 Run the test numerics step again.

The increased word length improves the results. This time, the plot shows that the fixed-point results match the floating-point results.


Figures - Testbench: Comparison plot 24-bit WL (11:10:37)
File
Edit
View
Insert
Tools
Debug
Desktop
Window
Help
Testbench: Comparison plot 16-bit WL (11:09:05)
Testbench: Comparison plot 24-bit WL (11:10:37)
myFilter > float : y
myFilter > fixed : y
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1

View and Modify Variable Information

View Variable Information

On the **Convert to Fixed Point** page of the MATLAB Coder app, you can view information about the variables in the MATLAB functions. To view information about the variables that you select in the **Source Code** pane, use the **Variables** tab or place your cursor over a variable in the code window. For more information, see “Viewing Variables” on page 14-97.

You can view the variable information:

- **Variable**

  Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.
- **Type**

  The original size, type, and complexity of each variable.
- **Sim Min**

  The minimum value assigned to the variable during simulation.
- **Sim Max**

  The maximum value assigned to the variable during simulation.

To search for a variable in the MATLAB code window and on the **Variables** tab, use `Ctrl+F`. The app highlights occurrences of the variable in the code.

Modify Variable Information

If you modify variable information, the app highlights the modified values using bold text. You can modify the following fields:

- **Static Min**

  You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 14-82.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Static Max**

  You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 14-82.

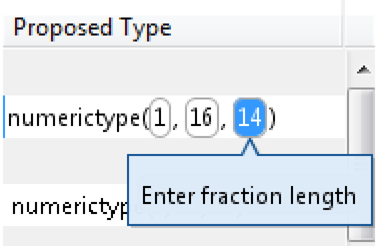
  Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.
- **Whole Number**

  The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

  Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.
- **Proposed Type**

  You can modify the signedness, word length, and fraction length settings individually:

  - On the **Variables** tab, modify the value in the **ProposedType** field.

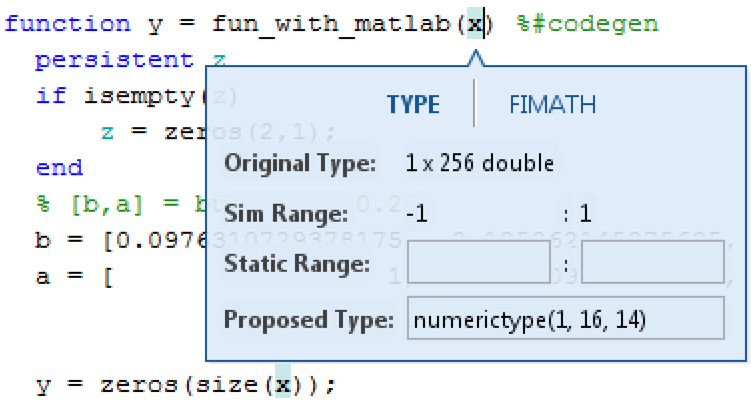


Proposed Type

numerictype(1, 16, 14)

Enter fraction length

  - In the code window, select a variable, and then modify the **ProposedType** field.



```
function y = fun_with_matlab(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = b
  b = [0.0976
  a = [

  y = zeros(size(x));
```

Original Type: 1 x 256 double

Sim Range: -1 : 1

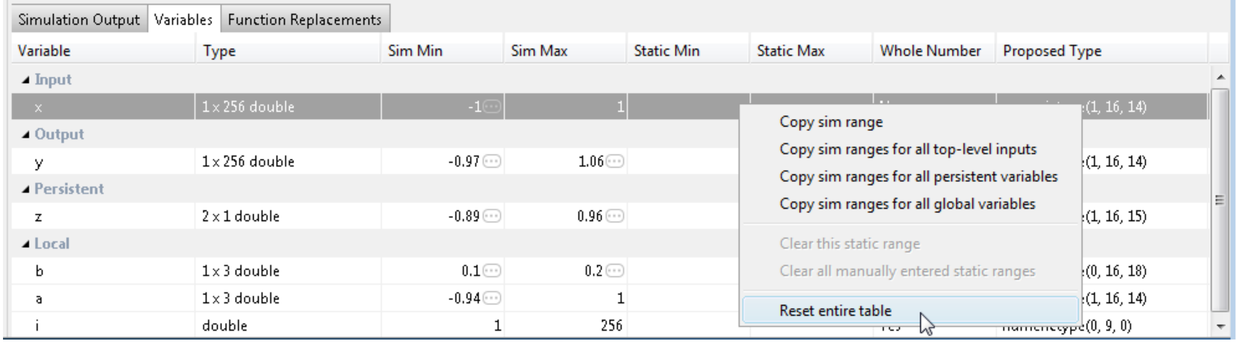
Static Range: :

Proposed Type: numerictype(1, 16, 14)

If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see "Log Data for Histogram" on page 14-99.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select `Reset entire table`.



Simulation Output
Variables
Function Replacements

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
x	1 x 256 double	-1	1				(1, 16, 14)
Output							
y	1 x 256 double	-0.97	1.06				(1, 16, 14)
Persistent							
z	2 x 1 double	-0.89	0.96				(1, 16, 15)
Local							
b	1 x 3 double	0.1	0.2				(0, 16, 18)
a	1 x 3 double	-0.94	1				(1, 16, 14)
i	double	1	256				numerictype(0, 9, 0)

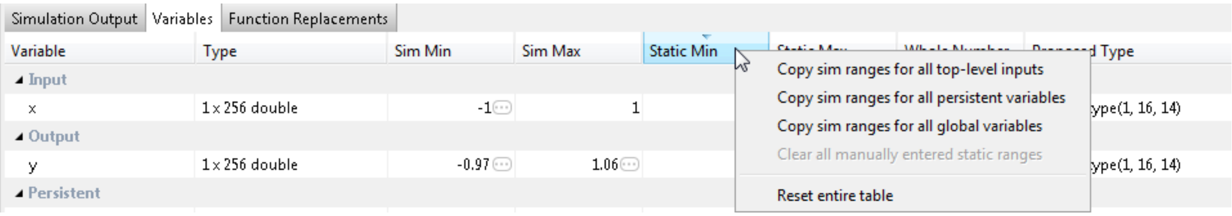
Copy sim range
Copy sim ranges for all top-level inputs
Copy sim ranges for all persistent variables
Copy sim ranges for all global variables
Clear this static range
Clear all manually entered static ranges
Reset entire table

- To revert the type of a selected variable to the type computed by the app, right-click the field and select `Undo changes`.
- To revert changes to variables, right-click the field and select `Undo changes for all variables`.

- To clear a static range value, right-click an edited field and select `Clear this static range`.
- To clear manually entered static range values, right-click anywhere on the **Variables** tab and select `Clear all manually entered static ranges`.

Promote Sim Min and Sim Max Values

With the MATLAB Coder app, you can promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.


Simulation Output
Variables
Function Replacements
Copy sim ranges for all top-level inputs
Copy sim ranges for all persistent variables
Copy sim ranges for all global variables
Clear all manually entered static ranges
Reset entire table

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
◢ Input							
x	1 x 256 double	-1	1				ype(1, 16, 14)
◢ Output							
y	1 x 256 double	-0.97	1.06				ype(1, 16, 14)
◢ Persistent							

To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select `Copy sim range`.
- Simulation ranges for top-level inputs, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all top-level inputs`.
- Simulation ranges for persistent variables, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all persistent variables`.

Automated Fixed-Point Conversion

In this section...

Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the MATLAB Coder app or at the command line using the `codegen` function `-float2fixed` option. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

You can manually enter static ranges. These manually entered ranges take precedence over simulation ranges and the app uses them when proposing data types. In addition, you can modify and lock the proposed type so that the app cannot change it. For more information, see "Locking Proposed Data Types" on page 14-90.

For a list of supported MATLAB features and functions, see "MATLAB Language Features Supported for Automated Fixed-Point Conversion" (Fixed-Point Designer).

During fixed-point conversion, you can:

- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- Propose fraction lengths based on default word lengths.

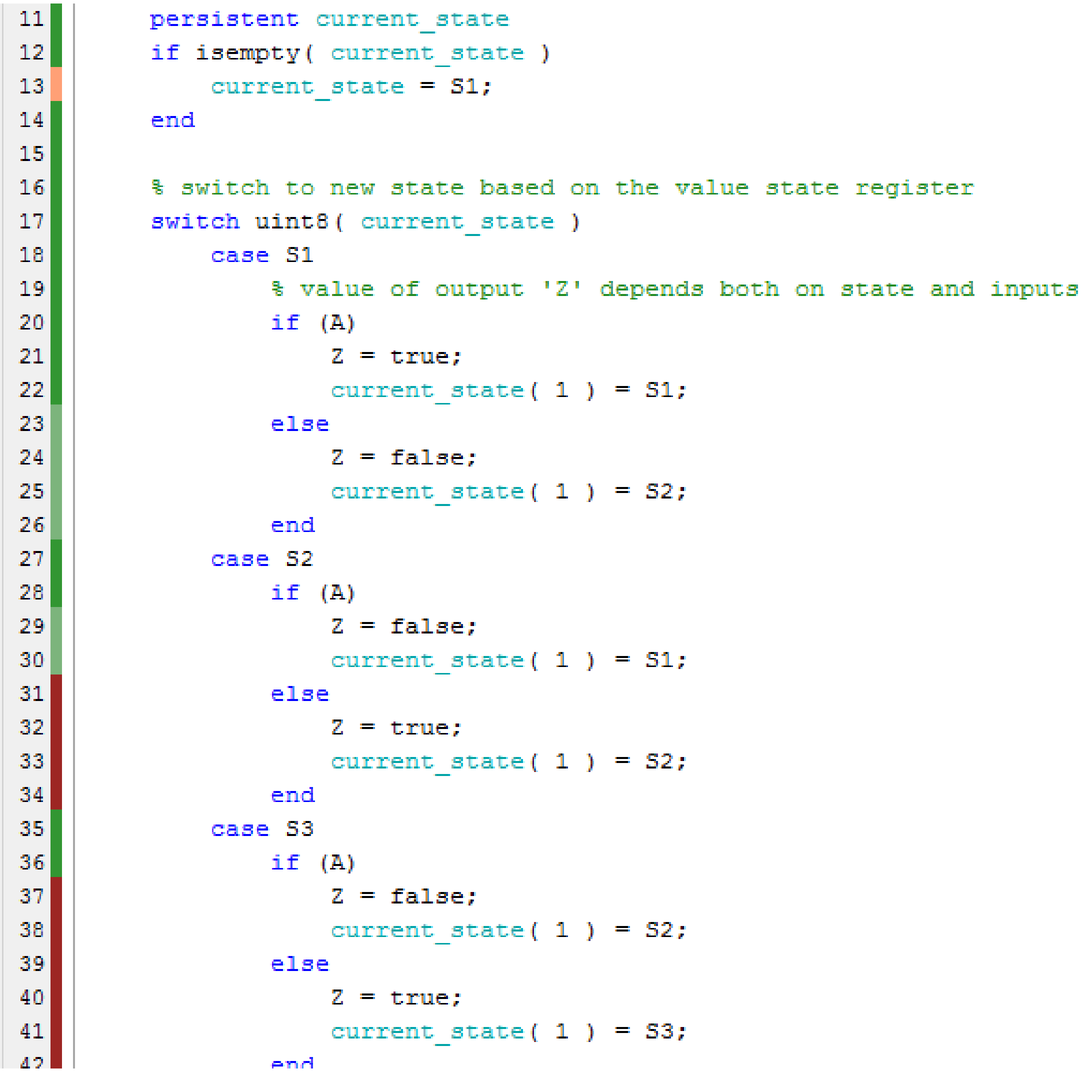
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- Validate that you can build your project with the proposed data types.
- Test numerics by running the test file with the fixed-point types applied.
- View a histogram of bits that each variable uses.
- Detect overflows.

Code Coverage

By default, the app shows code coverage results. Your test files must exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want.

Reviewing code coverage results helps you to verify that your test files are exercising the algorithm adequately. If the code coverage is inadequate, modify the test files or add more test files to increase coverage. If you simulate multiple test files in one run, the app displays cumulative coverage. However, if you specify multiple test files, but run them one at a time, the app displays the coverage of the file that ran last.

The app displays a color-coded coverage bar to the left of the code.



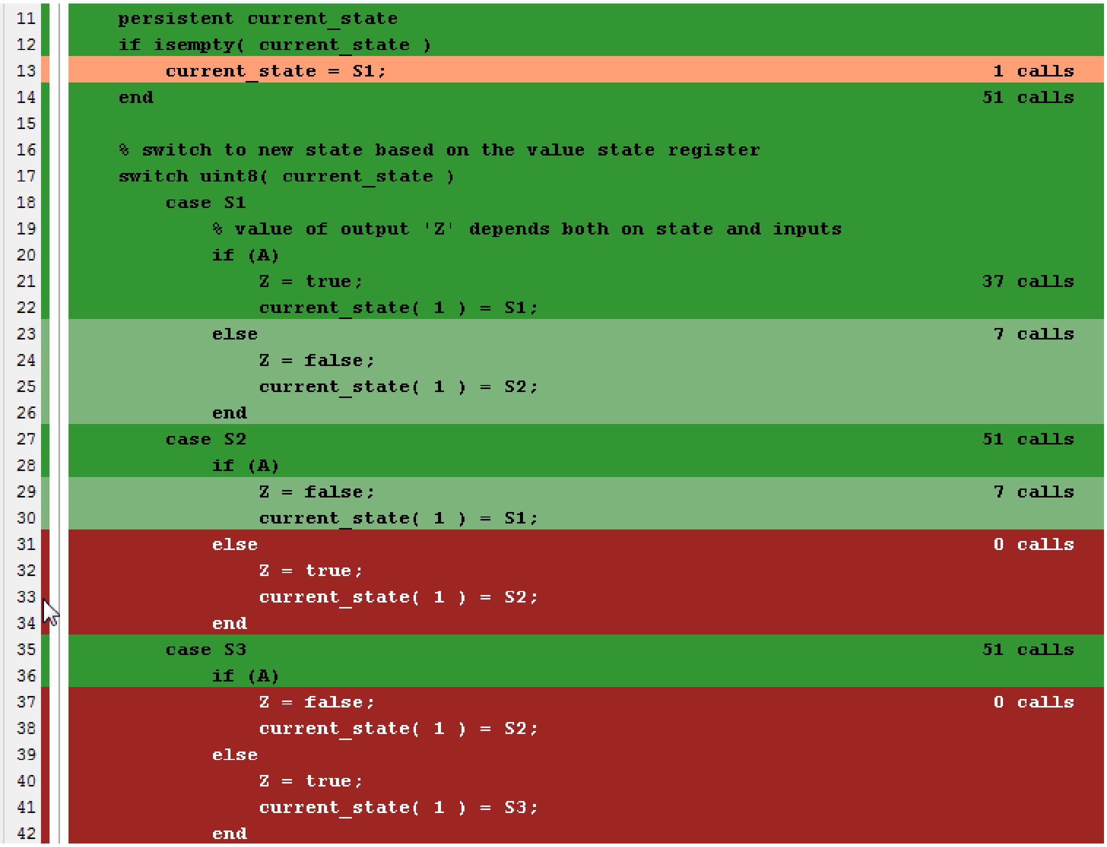
```
persistent current_state
if isempty( current_state )
    current_state = S1;
end

% switch to new state based on the value state register
switch uint8( current_state )
    case S1
        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state( 1 ) = S1;
        else
            Z = false;
            current_state( 1 ) = S2;
        end
    case S2
        if (A)
            Z = false;
            current_state( 1 ) = S1;
        else
            Z = true;
            current_state( 1 ) = S2;
        end
    case S3
        if (A)
            Z = false;
            current_state( 1 ) = S2;
        else
            Z = true;
            current_state( 1 ) = S3;
        end
```

This table describes the color coding.

Coverage Bar Color	Indicates
Green	One of the following situations:

Coverage Bar Color	Indicates
	• The entry-point function executes multiple times and the code executes more than one time. • The entry-point function executes one time and the code executes one time. Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you place your cursor over the coverage bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that the section executes.



```
    persistent current_state
    if isempty( current_state )
        current_state = S1;                                      1 calls
    end                                                         51 calls

    % switch to new state based on the value state register
    switch uint8( current_state )
        case S1
            % value of output 'Z' depends both on state and inputs
            if (A)
                Z = true;                                       37 calls
                current_state( 1 ) = S1;
            else                                                 7 calls
                Z = false;
                current_state( 1 ) = S2;
            end
        case S2                                                 51 calls
            if (A)
                Z = false;                                       7 calls
                current_state( 1 ) = S1;
            else                                                 0 calls
                Z = true;
                current_state( 1 ) = S2;
            end
        case S3                                                 51 calls
            if (A)
                Z = false;                                       0 calls
                current_state( 1 ) = S2;
            else
                Z = true;
                current_state( 1 ) = S3;
            end
```

To verify that your test files are testing your algorithm over the intended operating range, review the code coverage results.

Coverage Bar Color	Action
Green	If you expect sections of code to execute more frequently than the coverage shows, either modify the MATLAB code or the test files.
Orange	This behavior is expected for initialization code, for example, the initialization of persistent variables. If you expect the code to execute more than one time, either modify the MATLAB code or the test files.
Red	If the code that does not execute is an error condition, this behavior is acceptable. If you expect the code to execute, either modify the

Coverage Bar Color	Action
	MATLAB code or the test files. If the code is written conservatively and has upper and lower boundary limits, and you cannot modify the test files to reach this code, add static minimum and maximum values. See "Computing Derived Ranges" on page 14-89.

Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage can speed up simulation. To turn off code coverage, on the **Convert to Fixed Point** page:

1. Click the **Analyze** arrow ▼.
2. Clear the **Show code coverage** check box.

Proposing Data Types

The app proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data (also known as static ranges), or both. If you run a simulation and compute derived ranges, the app merges the simulation and derived ranges.

Note: You cannot propose data types based on derived ranges for MATLAB classes.

You can manually enter static ranges. These manually entered ranges take precedence over simulation ranges and the app uses them when proposing data types. You can modify and lock the proposed type so that the tool cannot change it. For more information, see "Locking Proposed Data Types" on page 14-90.

Running a Simulation

During fixed-point conversion, the app generates an instrumented MEX function for your entry-point MATLAB file. If the build completes without errors, the app displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the app provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the app displays them on the **Function Replacements** tab. See “Function Replacements” on page 14-101.

Before running a simulation, specify the test file or files that you want to run. When you run a simulation, the app runs the test file, calling the instrumented MEX function. If you modify the MATLAB design code, the app automatically generates an updated MEX function before running a test file.

If the test file runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If you manually enter static ranges for a variable, the manually entered ranges take precedence over the simulation ranges. If you manually modify the proposed types by typing or using the histogram, the data types are locked so that the app cannot modify them.

If the test file fails, the errors are displayed on the **Output** tab.

Test files must exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test file covers the operating range of the algorithm with the accuracy that you want. You can add test files and select to run more than one test file during the simulation. If you run multiple test files, the app merges the simulation results.

Optionally, you can select to log data for histograms. After running a simulation, you can view the histogram for each variable. For more information, see “Log Data for Histogram” on page 14-99.

Computing Derived Ranges

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values or proposed data types for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can manually enter ranges or promote simulation ranges to use as static ranges. Manually entered static ranges always take precedence over simulation ranges.

If you know what data type your hardware target uses, set the proposed data types to match this type. Manually entered data types are locked so that the app cannot modify them. The app uses these data types to calculate the input minimum and maximum values and to derive ranges for other variables. For more information, see “Locking Proposed Data Types” on page 14-90.

When you select **Compute Derived Ranges**, the app runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces `+/-Inf` derived ranges, consider defining ranges for all persistent variables.

Optionally, you can select **Quick derived range analysis**. With this option, the app performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. When the timeout is reached, the app aborts the analysis.

Locking Proposed Data Types

You can lock proposed data types against changes by the app using one of the following methods:

- Manually setting a proposed data type in the app.
- Right-clicking a type proposed by the tool and selecting `Lock computed value`.

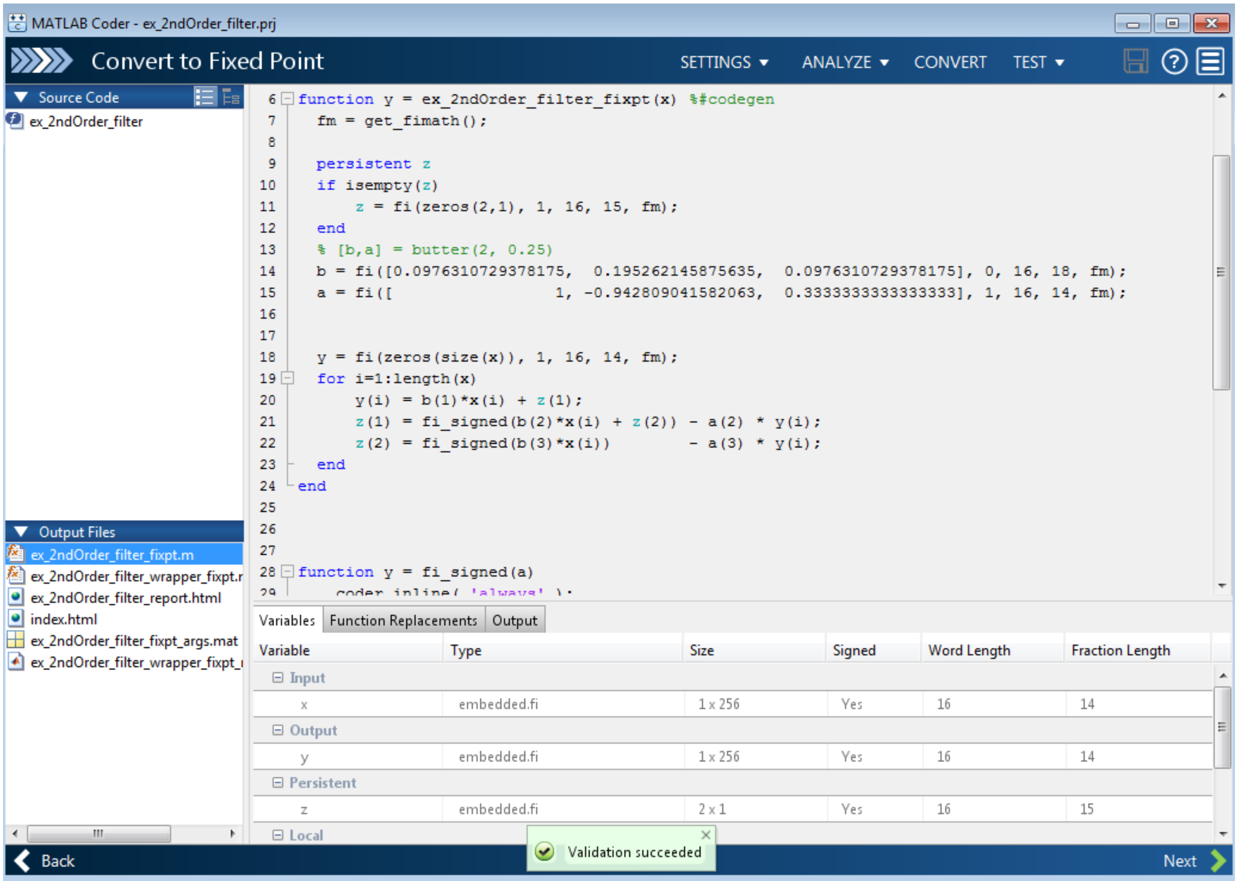
The app displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- Manually overwriting it.
- Right-clicking it and selecting `Undo changes`. This action unlocks only the selected type.
- Right-clicking and selecting `Undo changes for all variables`. This action unlocks all locked proposed types.

Viewing Functions

During the **Convert to Fixed Point** step of the fixed-point conversion process, you can view a list of functions in your project in the left pane. This list also includes function specializations and class methods. When you select a function from the list, the MATLAB code for that function or class method is displayed in the code window and the variables that they use are displayed on the **Variables** tab.

After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the app retains the original function name in the fixed-point file name and appends the fixed-point suffix. For example, here the fixed-point version of `ex_2ndOrder_filter.m` is `ex_2ndOrder_filter_fixpt.m`.


MATLAB Coder - ex_2ndOrder_filter.prj
Convert to Fixed Point
SETTINGS ▾
ANALYZE ▾
CONVERT
TEST ▾
Source Code
ex_2ndOrder_filter
Output Files
ex_2ndOrder_filter_fixpt.m
ex_2ndOrder_filter_wrapper_fixpt.r
ex_2ndOrder_filter_report.html
index.html
ex_2ndOrder_filter_fixpt_args.mat
ex_2ndOrder_filter_wrapper_fixpt_i
Variables
Function Replacements
Output
Back
Next

```
function y = ex_2ndOrder_filter_fixpt(x) %#codegen
  fm = get_fimath();

  persistent z
  if isempty(z)
      z = fi(zeros(2,1), 1, 16, 15, fm);
  end
  % [b,a] = butter(2, 0.25)
  b = fi([0.097631072937817,  0.195262145875635,  0.097631072937817], 0, 16, 18, fm);
  a = fi([                 1, -0.942809041582063,  0.333333333333333], 1, 16, 14, fm);


  y = fi(zeros(size(x)), 1, 16, 14, fm);
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
      z(2) = fi_signed(b(3)*x(i))        - a(3) * y(i);
  end
end



function y = fi_signed(a)
    coder.inline( 'always' );
```

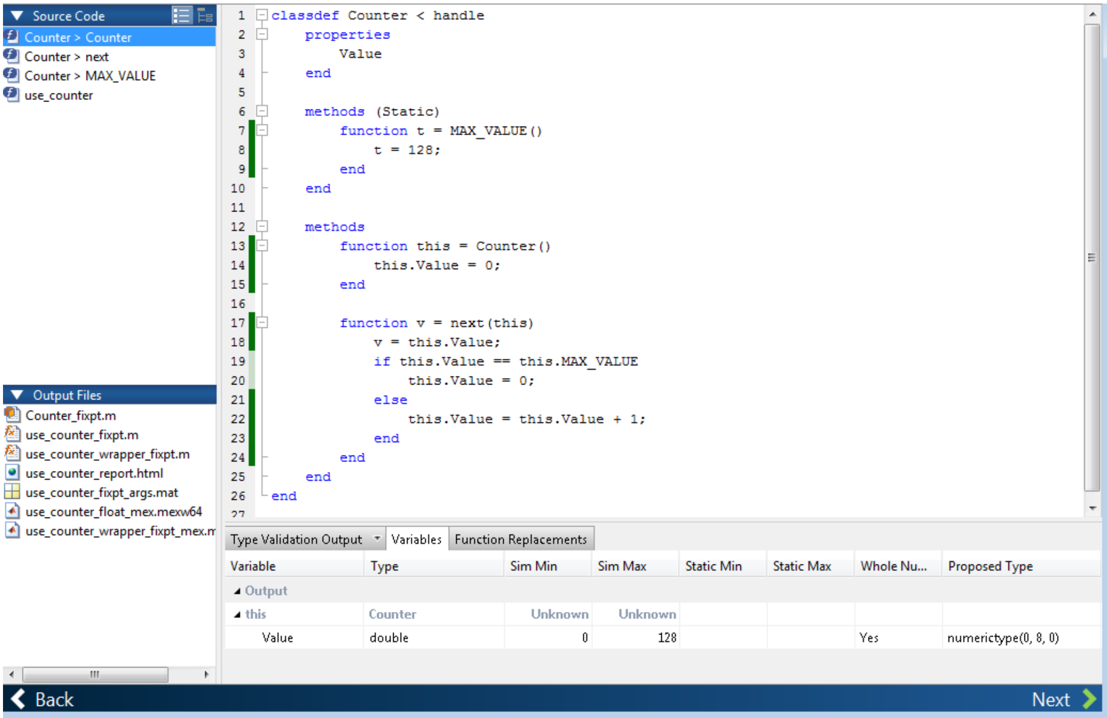
Variable	Type	Size	Signed	Word Length	Fraction Length
⊟ Input					
x	embedded.fi	1 x 256	Yes	16	14
⊟ Output					
y	embedded.fi	1 x 256	Yes	16	14
⊟ Persistent					
z	embedded.fi	2 x 1	Yes	16	15
⊟ Local					

✓ Validation succeeded

Classes

The app displays information for the class and each of its methods. For example, consider a class, `Counter`, that has a static method, `MAX_VALUE`, and a method, `next`.

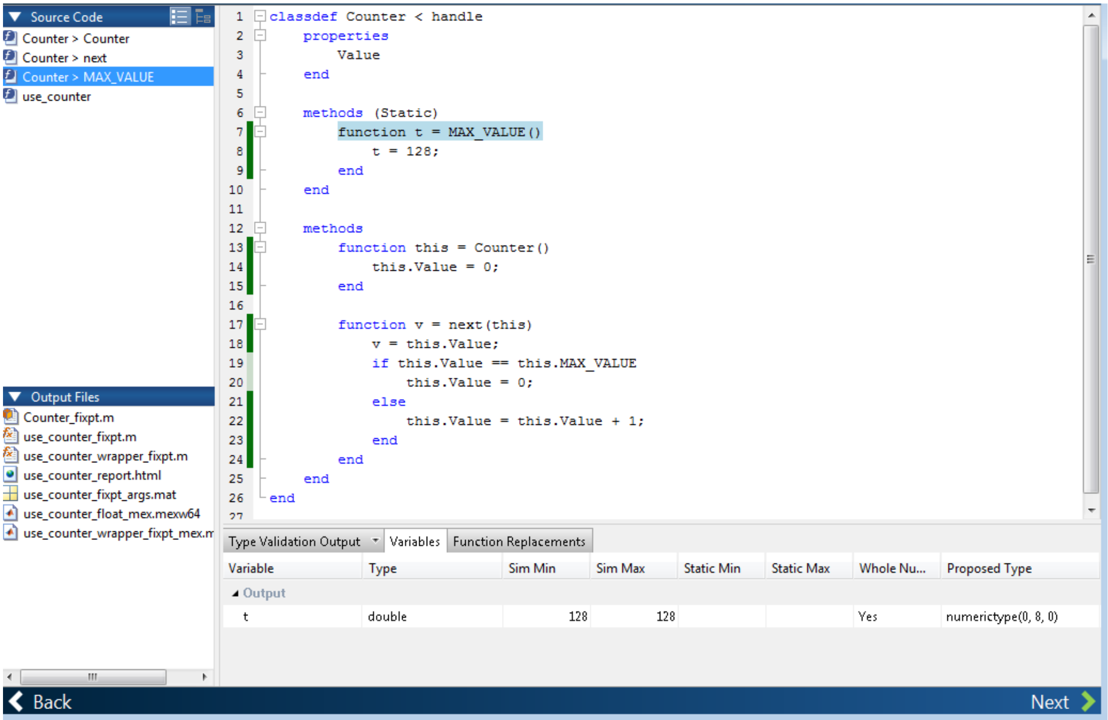
If you select the class, the app displays the class and its properties on the **Variables** tab.



```
classdef Counter < handle
    properties
        Value
    end

    methods (Static)
        function t = MAX_VALUE()
            t = 128;
        end
    end

    methods
        function this = Counter()
            this.Value = 0;
        end

        function v = next(this)
            v = this.Value;
            if this.Value == this.MAX_VALUE
                this.Value = 0;
            else
                this.Value = this.Value + 1;
            end
        end
    end
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Output							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)

If you select a method, the app displays only the variables that the method uses.



- Counter > Counter
- Counter > next
- Counter > MAX_VALUE
- use_counter

```
classdef Counter < handle
    properties
        Value
    end

    methods (Static)
        function t = MAX_VALUE()
            t = 128;
        end
    end

    methods
        function this = Counter()
            this.Value = 0;
        end

        function v = next(this)
            v = this.Value;
            if this.Value == this.MAX_VALUE
                this.Value = 0;
            else
                this.Value = this.Value + 1;
            end
        end
    end
end
```

- Counter_fixpt.m
- use_counter_fixpt.m
- use_counter_wrapper_fixpt.m
- use_counter_report.html
- use_counter_fixpt_args.mat
- use_counter_float_mex.mexw64
- use_counter_wrapper_fixpt_mex.m

Type Validation Output | Variables | Function Replacements

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Output							
t	double	128	128			Yes	numerictype(0, 8, 0)

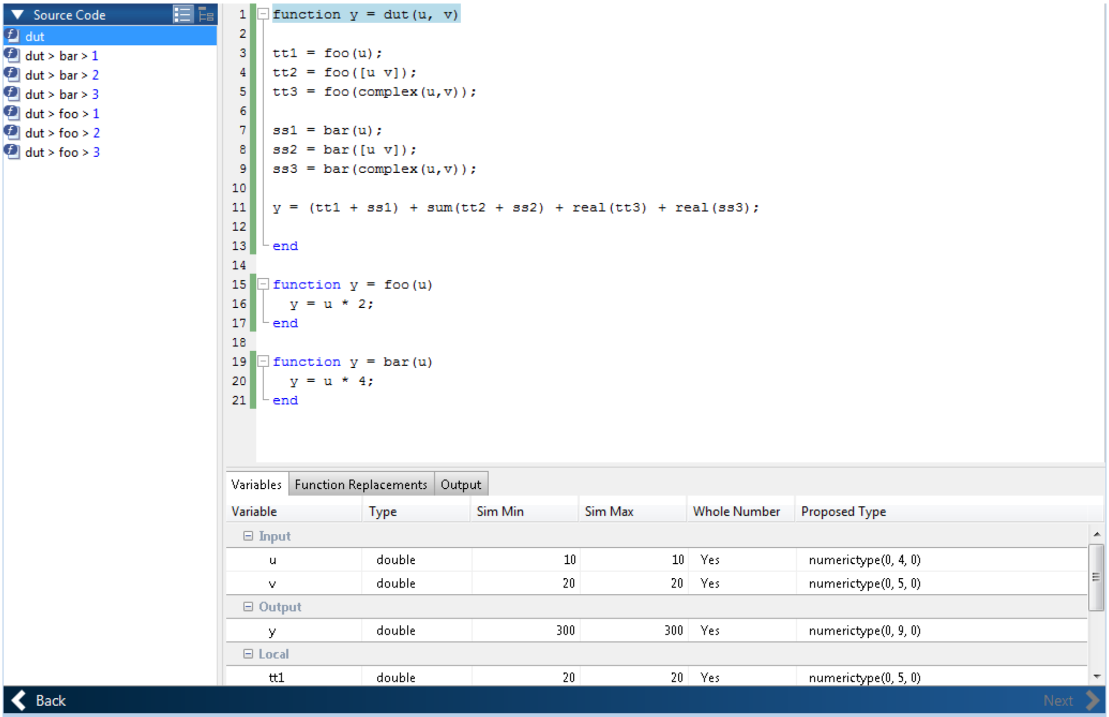
Specializations

If a function is specialized, the app lists each specialization and numbers them sequentially. For example, consider a function, `dut`, that calls subfunctions, `foo` and `bar`, multiple times with different input types.

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

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
```

```
end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```

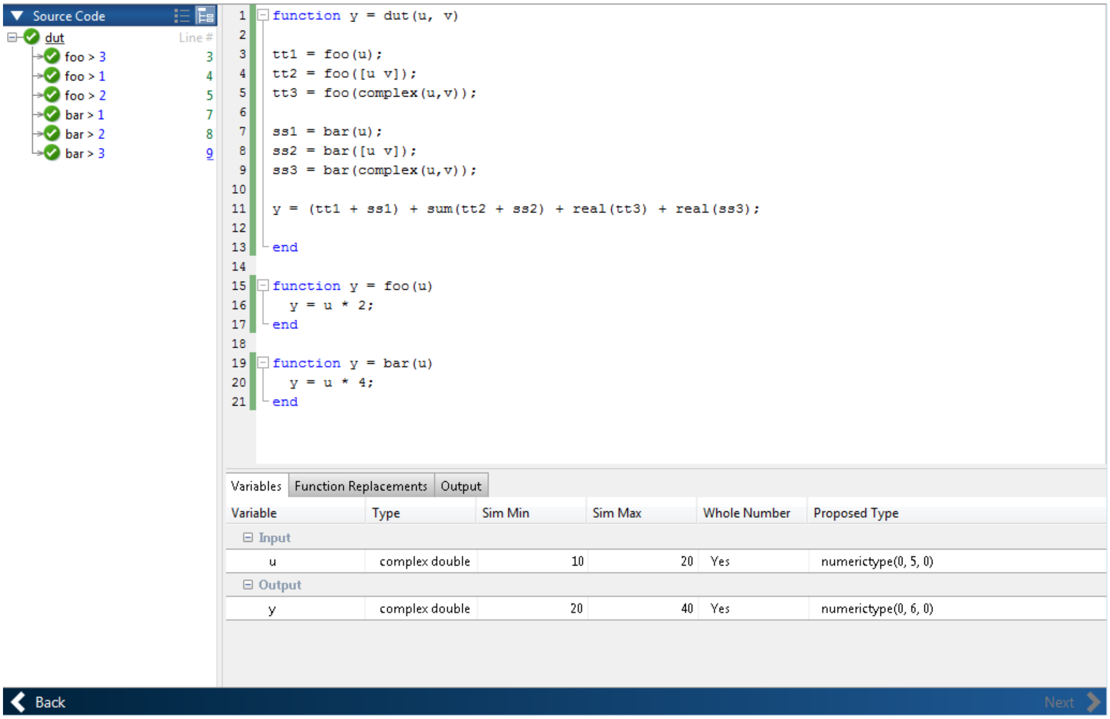
If you select the top-level function, the app displays all the variables on the **Variables** tab.



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

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
u	double	10	10	Yes	numerictype(0, 4, 0)
v	double	20	20	Yes	numerictype(0, 5, 0)
⊟ Output					
y	double	300	300	Yes	numerictype(0, 9, 0)
⊟ Local					
tt1	double	20	20	Yes	numerictype(0, 5, 0)

If you select the tree view, the app also displays the line numbers for the call to each specialization.

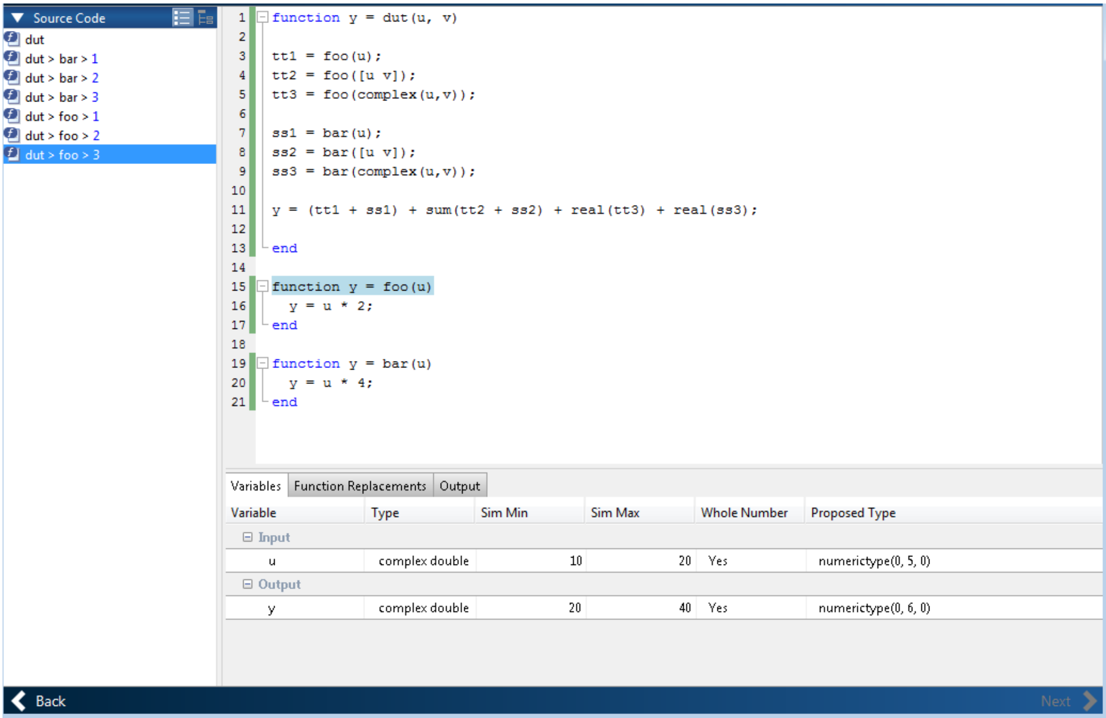


	Line #
dut	
foo > 3	3
foo > 1	4
foo > 2	5
bar > 1	7
bar > 2	8
bar > 3	9

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

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
    y = u * 2;
end

function y = bar(u)
    y = u * 4;
end
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
u	complex double	10	20	Yes	numerictype(0, 5, 0)
⊟ Output					
y	complex double	20	40	Yes	numerictype(0, 6, 0)

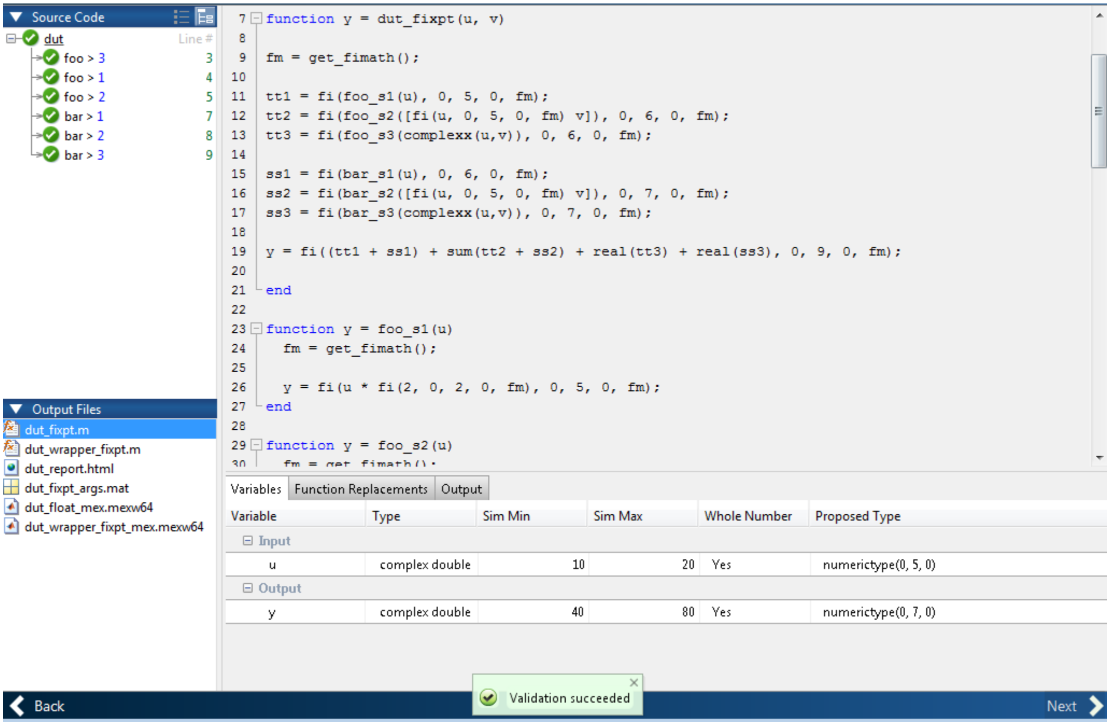
If you select a specialization, the app displays only the variables that the specialization uses.



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

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
  y = u * 2;
end

function y = bar(u)
  y = u * 4;
end
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
⊟ Input					
u	complex double	10	20	Yes	numerictype(0, 5, 0)
⊟ Output					
y	complex double	20	40	Yes	numerictype(0, 6, 0)

In the generated fixed-point code, the number of each fixed-point specialization matches the number in the **Source Code** list, which makes it easy to trace between the floating-point and fixed-point versions of your code. For example, the generated fixed-point function for `foo > 1` is named `foo_s1`.



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

fm = get_fimath();

tt1 = fi(foo_s1(u), 0, 5, 0, fm);
tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
tt3 = fi(foo_s3(complexx(u,v)), 0, 6, 0, fm);

ss1 = fi(bar_s1(u), 0, 6, 0, fm);
ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
ss3 = fi(bar_s3(complexx(u,v)), 0, 7, 0, fm);

y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);

end

function y = foo_s1(u)
  fm = get_fimath();

  y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
end

function y = foo_s2(u)
  fm = get_fimath();
```

Variable	Type	Sim Min	Sim Max	Whole Number	Proposed Type
Input					
u	complex double	10	20	Yes	numerictype(0, 5, 0)
Output					
y	complex double	40	80	Yes	numerictype(0, 7, 0)

Validation succeeded

Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

  You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the app.
- **Static Min** and **Static Max** — The static minimum and maximum values.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

When you compute derived ranges, the app runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the app.

- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

  The app determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the app uses the edited values in subsequent analyses. You can revert to the types proposed by the app.
- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

  Because the app does not apply data types to expressions, it does not display proposed types for them. Instead, it displays their original data types.

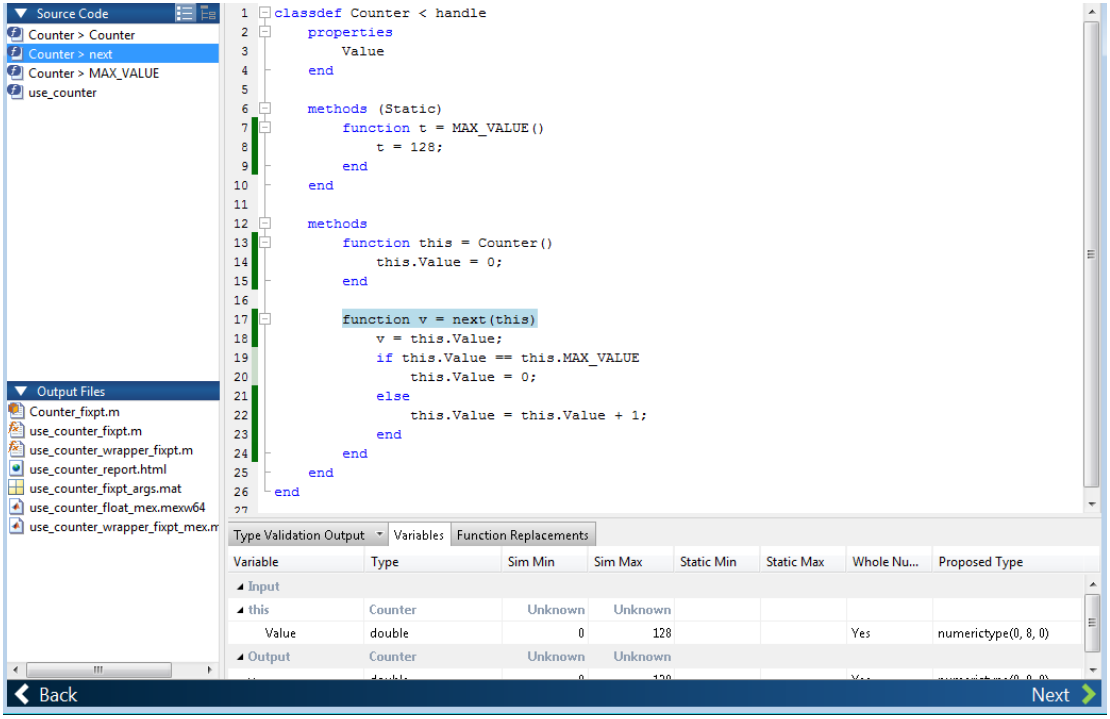
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

You can use `Ctrl+F` to search for variables in the MATLAB code and on the **Variables** tab. The app highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

Viewing Information for MATLAB Classes

The app displays:

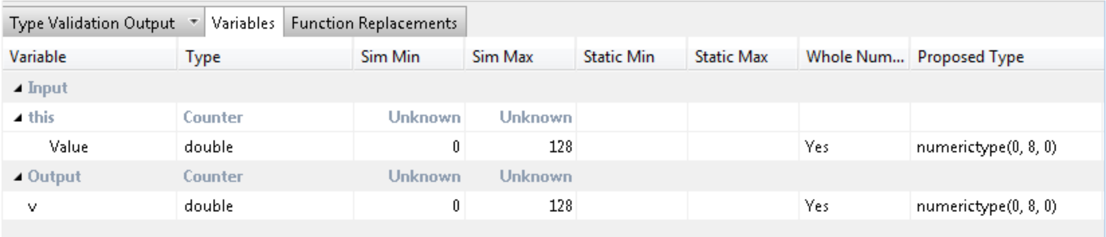
- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Source Code** list on the **Convert to Fixed Point** page to select which class or class method to view. If you select a class method, the app highlights the method in the code window.



```
classdef Counter < handle
    properties
        Value
    end

    methods (Static)
        function t = MAX_VALUE()
            t = 128;
        end
    end

    methods
        function this = Counter()
            this.Value = 0;
        end

        function v = next(this)
            v = this.Value;
            if this.Value == this.MAX_VALUE
                this.Value = 0;
            else
                this.Value = this.Value + 1;
            end
        end
    end
end
```

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Nu...	Proposed Type
Input							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)
Output	Counter	Unknown	Unknown				

- Information about MATLAB classes on the **Variables** tab.



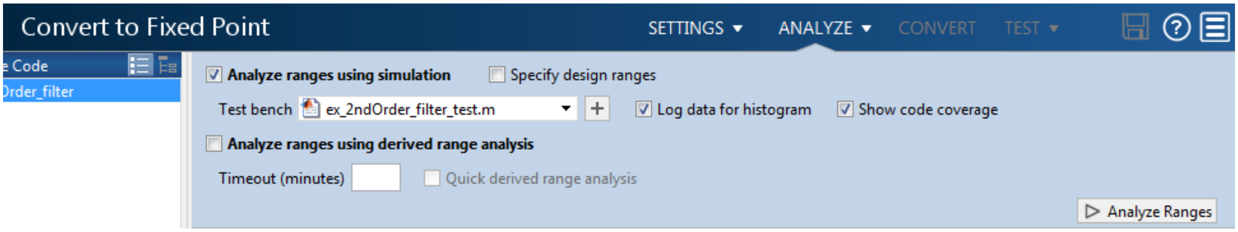
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)
Output	Counter	Unknown	Unknown				
v	double	0	128			Yes	numerictype(0, 8, 0)

Log Data for Histogram

To log data for histograms:

- On the **Convert to Fixed Point** page, click the **Analyze** arrow ▼.

- Select **Log data for histogram**.



☑ Analyze ranges using simulation □ Specify design ranges

Test bench ex_2ndOrder_filter_test.m ☑ Log data for histogram ☑ Show code coverage

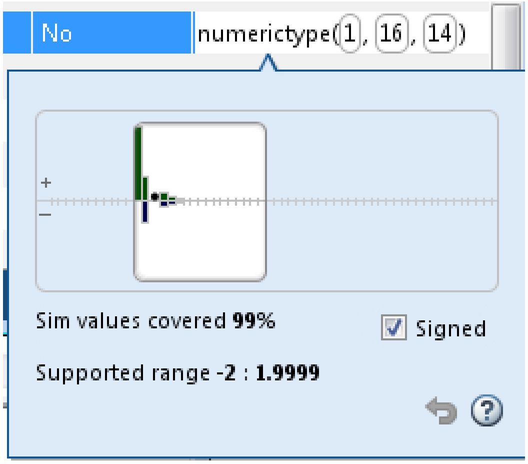
□ Analyze ranges using derived range analysis

Timeout (minutes) □ Quick derived range analysis

- Click **Analyze Ranges**.

After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1,16,14)`.

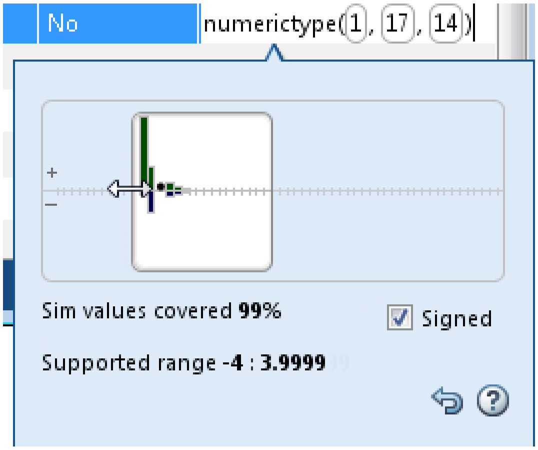


Sim values covered **99%** ☑ Signed

Supported range **-2** : **1.9999**

You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



No | numerictype(1, 17, 14)

Sim values covered **99%**

☑ Signed

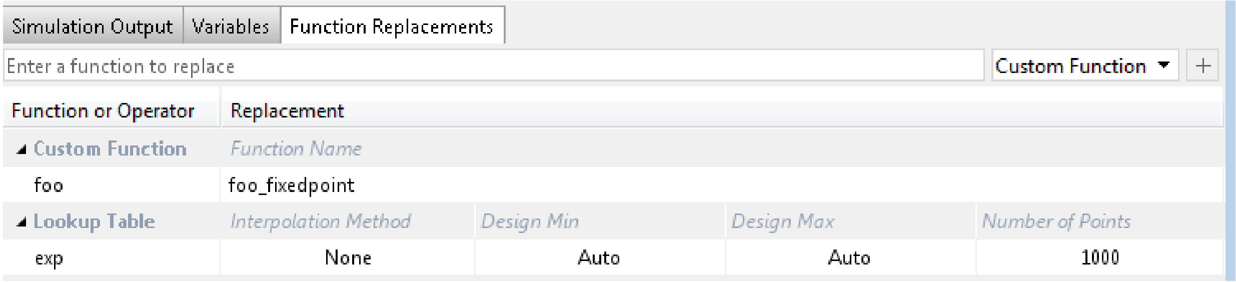
Supported range **-4 : 3.9999**

- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click ↩.

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the app lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.



Function or Operator	Replacement			
◢ Custom Function	*Function Name*			
foo	foo_fixedpoint			
◢ Lookup Table	*Interpolation Method*	*Design Min*	*Design Max*	*Number of Points*
exp	None	Auto	Auto	1000

You can add and remove function replacements from this list. If you enter a function replacement for a function, the replacement function is used when you build the project. If you do not enter a replacement, the app uses the type specified in the original MATLAB code for the function.

Note: Using this table, you can replace the names of the functions but you cannot replace argument patterns.

If code generation readiness screening is disabled, the list of unsupported functions on the **Function Replacements** tab can be incomplete or incorrect. In this case, add the functions manually. See “Code Generation Readiness Screening in the MATLAB Coder App” on page 17-38.

Validating Types

Converting the code to fixed point validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

Testing Numerics

After converting code to fixed point and validating the proposed fixed-point data types, click **Test** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test file to define inputs or run a simulation, the app uses this test file to test numerics. Optionally, you can add test files and select to run more than one test file. The app compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the app generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For nonscalar outputs, only the error information is shown.

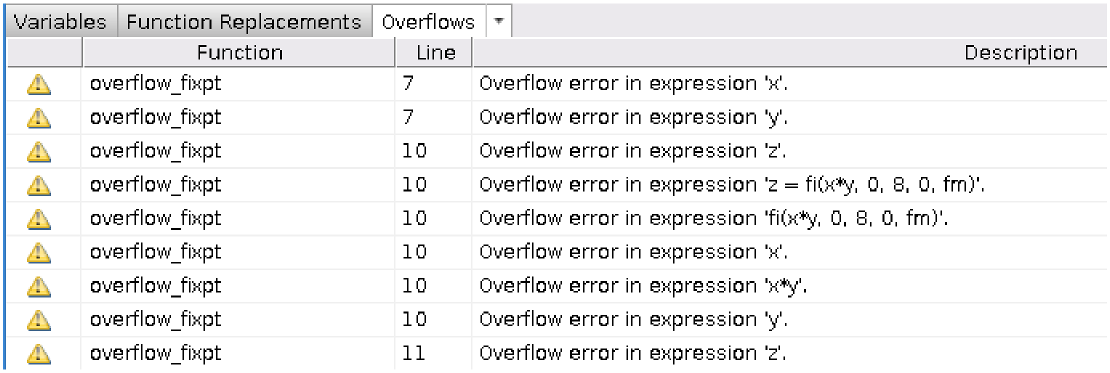
After fixed-point simulation, if the numerical results do not meet the accuracy that you want, modify fixed-point data type settings and repeat the type validation and numerical testing steps. You might have to iterate through these steps multiple times to achieve the results that you want.

Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion app runs the simulation using scaled double versions of the proposed fixed-point types. Because scaled doubles store their data in double-precision floating-point, they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type. .

If the app detects overflows, on its **Overflow** tab, it provides:

- A list of variables and expressions that overflowed
- Information on how much each variable overflowed
- A link to the variables or expressions in the code window



	Function	Line	Description
⚠	overflow_fixpt	7	Overflow error in expression 'x'.
⚠	overflow_fixpt	7	Overflow error in expression 'y'.
⚠	overflow_fixpt	10	Overflow error in expression 'z'.
⚠	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'x'.
⚠	overflow_fixpt	10	Overflow error in expression 'x*y'.
⚠	overflow_fixpt	10	Overflow error in expression 'y'.
⚠	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the app also provides overflow information for these expressions.

See Also

“Detect Overflows” on page 14-40

Convert Fixed-Point Conversion Project to MATLAB Scripts

This example shows how to convert a MATLAB Coder project to MATLAB scripts when the project includes automated fixed-point conversion. You can use the `-tocode` option of the `coder` command to create a pair of scripts for fixed-point conversion and fixed-point code generation. You can use the scripts to repeat the project workflow in a command-line workflow. Before you convert the project to the scripts, you must complete the **Test** step of the fixed-point conversion process.

Prerequisites

This example uses the following files:

- Project file `ex_2ndOrder_filter.prj`
- Entry-point file `ex_2ndOrder_filter.m`
- Test bench file `ex_2ndOrder_filter_test.m`
- Generated fixed-point MATLAB file `ex_2ndOrder_filter_fixpt.m`

To obtain these files, complete the example “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 14-7, including these steps:

1. Complete the **Test** step of the fixed-point conversion process.
2. Configure the project to build a C/C++ static library.

Generate the Scripts

1. Change to the folder that contains the project file `ex_2ndOrder_filter.prj`.
2. Use the `-tocode` option of the `coder` command to convert the project to the scripts. Use the `-script` option to specify the file name for the scripts.

   ```
   coder -tocode ex_2ndOrder_filter -script ex_2ndOrder_filter_script.m
   ```

   The `coder` command generates two scripts in the current folder:

   `ex_2ndOrder_filter_script.m` contains the MATLAB commands to:

   - Create a code configuration object that has the same settings as the project.
   - Run the `codegen` command to convert the fixed-point MATLAB function `ex_2ndOrder_filter_fixpt` to a fixed-point C function.

The fixedPointConverter command generates a script in the current folder. ex_2ndOrder_filter_script_fixpt.m contains the MATLAB commands to:

- Create a floating-point to fixed-point conversion configuration object that has the same fixed-point conversion settings as the project.
- Run the codegen command to convert the MATLAB function ex_2ndOrder_filter to the fixed-point MATLAB function ex_2ndOrder_filter_fixpt.

  The suffix in the script file name is the generated fixed-point file name suffix specified by the project file. In this example, the suffix is the default value _fixpt.

The coder command overwrites existing files that have the same names as the generated scripts. If you omit the -script option, the coder command writes the scripts to the Command Window.

Run Script That Generates Fixed-Point C Code

To run the script that generates fixed-point C code from fixed-point MATLAB code, the fixed-point MATLAB function specified in the script must be available.

1. Make sure that the fixed-point MATLAB function ex_2ndOrder_filter_fixpt.m is on the search path.

   ```
   addpath c:\coder\ex_2ndOrder_filter\codegen\ex_2ndOrder_filter\fixpt
   ```
2. Run the script:

   ```
   ex_2ndOrder_filter_script
   ```

   The code generator creates a C static library with the name ex_2ndOrder_filter_fixpt in the folder codegen\lib\ex_2ndOrder_filter_fixpt. The variables cfg and ARGS appear in the base workspace.

Run Script That Generates Fixed-Point MATLAB Code

If you do not have the fixed-point MATLAB function, or if you want to regenerate it, use the script that generates the fixed-point MATLAB function from the floating-point MATLAB function.

1 Make sure that the current folder contains the entry-point function `ex_2ndOrder_filter.m` and the test bench file `ex_2ndOrder_filter_test.m`.

2 Run the script.

```
ex_2ndOrder_filter_script_fixpt
```

The code generator creates `ex_2ndOrder_filter_fixpt.m` in the folder `codegen\ex_2ndOrder_filter\fixpt`. The variables `cfg` and `ARGS` appear in the base workspace.

See Also

coder.FixptConfig | `codegen` | `coder`

Related Examples

- "Convert MATLAB Code to Fixed-Point C Code" on page 14-5
- "Propose Fixed-Point Data Types Based on Simulation Ranges" on page 14-7
- "Convert MATLAB Coder Project to MATLAB Script" on page 20-42

Generated Fixed-Point Code

In this section...

Location of Generated Fixed-Point Files

By default, the fixed-point conversion process generates files in a folder named `codegen/fcn_name/fixpt` in your local working folder. `fcn_name` is the name of the MATLAB function that you are converting to fixed point.

File name	Description
`fcn_name_fixpt.m`	Generated fixed-point MATLAB code. To integrate this fixed-point code into a larger application, consider generating a MEX-function for the function and calling this MEX-function in place of the original MATLAB code.
`fcn_name_fixpt_exVal.mat`	MAT-file containing: • A structure for the input arguments. • The name of the fixed-point file.
`fcn_name_fixpt_report.html`	Link to the type proposal report that displays the generated fixed-point code and the proposed type information.
`fcn_name_report.html`	Link to the type proposal report that displays the original MATLAB code and the proposed type information.

File name	Description
`fcn_name_wrapper_fixpt.m`	File that converts the floating-point data values supplied by the test file to the fixed-point types determined for the inputs during the conversion step. These fixed-point values are fed into the converted fixed-point function, `fcn_name_fixpt`.

Minimizing `fi`-casts to Improve Code Readability

The conversion process tries to reduce the number of `fi`-casts by analyzing the floating-point code. If an arithmetic operation is comprised of only compile-time constants, the conversion process does not cast the operands to fixed point individually. Instead, it casts the entire expression to fixed point.

For example, here is the fixed-point code generated for the constant expression `x = 1/sqrt(2)` when the selected word length is 14.

Original MATLAB Code	Generated Fixed-Point Code
`x = 1/sqrt(2);`	`x = fi(1/sqrt(2), 0, 14, 14, fm);` `fm` is the local `fimath`.

Avoiding Overflows in the Generated Fixed-Point Code

The conversion process avoids overflows by:

- Using full-precision arithmetic unless you specify otherwise.
- Avoiding arithmetic operations that involve double and `fi` data types. Otherwise, if the word length of the `fi` data type is not able to represent the value in the double constant expression, overflows occur.
- Avoiding overflows when adding and subtracting non fixed-point variables and fixed-point variables.

  The fixed-point conversion process casts non-`fi` expressions to the corresponding `fi` type.

  For example, consider the following MATLAB algorithm.

```
% A = 5;
% B = ones(300, 1)
function y = fi_plus_non_fi(A, B)
  % '1024' is non-fi, cast it
  y = A + 1024;
  % 'size(B, 1)*length(A)' is a non-fi, cast it
  y = A + size(B, 1)*length(A);
end
```

The generated fixed-point code is:

```
%#codegen
% A = 5;
% B = ones(300, 1)
function y = fi_plus_non_fi_fixpt(A, B)
  % '1024' is non-fi, cast it
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
              'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
              'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

  y = fi(A + fi(1024, 0, 11, 0, fm), 0, 11, 0, fm);
  % 'size(B, 1)*length(A)' is a non-fi, cast it
  y(:) = A + fi(size(B, fi(1, 0, 1, 0, fm))*length(A), 0, 9, 0, fm);
end
```

Controlling Bit Growth

The conversion process controls bit growth by using subscripted assignments, that is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. Using subscripted assignment keeps fixed-point variables fixed point rather than inadvertently turning them into doubles. Maintaining the fixed-point type reduces the number of type declarations in the generated code. Subscripted assignment also prevents bit growth which is useful when you want to maintain a particular data type for the output.

Avoiding Loss of Range or Precision

Avoiding Loss of Range or Precision in Unsigned Subtraction Operations

When the result of the subtraction is negative, the conversion process promotes the left operand to a signed type.

For example, consider the following MATLAB algorithm.

```
% A = 1;
% B = 5
function [y,z] = unsigned_subtraction(A,B)
  y = A - B;

  C = -20;
  z = C - B;
end
```

In the original code, both A and B are unsigned and the result of A-B can be negative. In the generated fixed-point code, A is promoted to signed. In the original code, C is signed, so does not require promotion in the generated code.

```
%#codegen
% A = 1;
% B = 5
function [y,z] = unsigned_subtraction_fixpt(A,B)

fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
              'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
              'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);
y = fi(fi_signed(A) - B, 1, 3, 0, fm);
C = fi(-20, 1, 6, 0, fm);
z = fi(C - B, 1, 6, 0, fm);
end


function y = fi_signed(a)
coder.inline( 'always' );
if isfi( a ) && ~(issigned( a ))
  nt = numerictype( a );
  new_nt = numerictype( 1, nt.WordLength + 1, nt.FractionLength );
  y = fi( a, new_nt, fimath( a ) );
else
  y = a;
end
end
```

Avoiding Loss of Range When Concatenating Arrays of Fixed-Point Numbers

If you concatenate matrices using vertcat and horzcat, the conversion process uses the largest numerictype among the expressions of a row and casts the leftmost element to that type. This type is then used for the concatenated matrix to avoid loss of range.

For example, consider the following MATLAB algorithm.

```
% A = 1, B = 100, C = 1000
function [y, z] = lb_node(A, B, C)
  %% single rows
  y = [A B C];
  %% multiple rows
  z = [A 5; A B; A C];
end
```

In the generated fixed-point code:

- For the expression `y = [A B C]`, the leftmost element, `A`, is cast to the type of `C` because `C` has the largest type in the row.
- For the expression `[A 5; A B; A C]`:
  - In the first row, `A` is cast to the type of `C` because `C` has the largest type of the whole expression.
  - In the second row, `A` is cast to the type of `B` because `B` has the larger type in the row.
  - In the third row, `A` is cast to the type of `C` because `C` has the larger type in the row.

```
%#codegen
% A = 1, B = 100, C = 1000
function [y, z] = lb_node_fixpt(A, B, C)
  %% single rows
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
              'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, ...
              'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

  y = fi([fi(A, 0, 10, 0, fm) B C], 0, 10, 0, fm);

  %% multiple rows
  z = fi([fi(A, 0, 10, 0, fm) 5; fi(A, 0, 7, 0, fm) B;...
        fi(A, 0, 10, 0, fm) C], 0, 10, 0, fm);
end
```

Handling Non-Constant mpower Exponents

If the function that you are converting has a scalar input, and the `mpower` exponent input is not constant, the conversion process sets the `fimath` `ProductMode` to

SpecifyPrecision in the generated code. With this setting , the output data type can be determined at compile time.

For example, consider the following MATLAB algorithm.

```
% a = 1
% b = 3
function y = exp_operator(a, b)
  % exponent is a constant so no need to specify precision
  y = a^3;
  % exponent is not a constant, use 'SpecifyPrecision' for 'ProductMode'
  y = b^a;
end
```

In the generated fixed-point code, for the expression `y = a^3` , the exponent is a constant, so there is no need to specify precision. For the expression, `y = b^a`, the exponent is not constant, so the `ProductMode` is set to `SpecifyPrecision`.

```
%#codegen
% a = 1
% b = 3
function y = exp_operator_fixpt(a, b)
  % exponent is a constant so no need to specify precision
  fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
              'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128,...
              'SumMode', 'FullPrecision', 'MaxSumWordLength', 128);

  y = fi(a^3, 0, 2, 0, fm);
  % exponent is not a constant, use 'SpecifyPrecision' for 'ProductMode'
  y(:) = fi(b, 'ProductMode', 'SpecifyPrecision',...
               'ProductWordLength', 2, 'ProductFractionLength', 0 )^a;
end
```

Fixed-Point Code for MATLAB Classes

In this section...

Automated Conversion Support for MATLAB Classes

The automated fixed-point conversion process:

- Proposes fixed-point data types based on simulation ranges for MATLAB classes. It does not propose data types based on derived ranges for MATLAB classes.

  After simulation, the MATLAB Coder app:

  - Function list contains class constructors, methods, and specializations.
  - Code window displays the objects used in each function.
  - Provides code coverage for methods.

  For more information, see “Viewing Information for MATLAB Classes” on page 14-98.
- Supports class methods, properties, and specializations. For each specialization of a class, `class_name`, the conversion generates a separate `class_name_fixpt.m` file. For every instantiation of a class, the generated fixed-point code contains a call to the constructor of the appropriate specialization.
- Supports classes that have `get` and `set` methods such as `get.PropertyName`, `set.PropertyName`. These methods are called when properties are read or assigned. The `set` methods can be specialized. Sometimes, in the generated fixed-point code, assignment statements are transformed to function calls.

Unsupported Constructs

The automated conversion process does not support:

- Class inheritance.
- Packages.
- Constructors that use `nargin` and `varargin`.

Coding Style Best Practices

When you write MATLAB code that uses MATLAB classes:

- Initialize properties in the class constructor.
- Replace constant properties with static methods.

For example, consider the counter class.

```
classdef Counter < handle
  properties
    Value = 0;
  end

  properties(Constant)
    MAX_VALUE = 128
  end

  methods
    function out = next(this)
      out = this.Count;
      if this.Value == this.MAX_VALUE
        this.Value = 0;
      else
        this.Value = this.Value + 1;
      end
    end
  end
end
```

To use the automated fixed-point conversion process, rewrite the class to have a static class that initializes the constant property MAX_VALUE and a constructor that initializes the property Value.

```
classdef Counter < handle
  properties
    Value;
  end

  methods(Static)
    function t = MAX_VALUE()
      t = 128;
    end
  end
  methods
    function this = Counter()
      this.Value = 0;
    end
    function out = next(this)
      out = this.Value;
      if this.Value == this.MAX_VALUE
        this.Value = 0;
      else
        this.Value = this.Value + 1;
      end
    end
  end
end
```

Automated Fixed-Point Conversion Best Practices

In this section...

Create a Test File

A best practice for structuring your code is to separate your core algorithm from other code that you use to test and verify the results. Create a test file to call your original MATLAB algorithm and fixed-point versions of the algorithm. For example, as shown in the following table, you might set up some input data to feed into your algorithm, and then, after you process that data, create some plots to verify the results. Since you need to convert only the algorithmic portion to fixed point, it is more efficient to structure your code so that you have a test file, in which you create your inputs, call your algorithm, and plot the results, and one (or more) algorithmic files, in which you do the core processing.

Original code	Best Practice	Modified code

`% TEST INPUT`
`x = randn(100,1);`
 	
`% ALGORITHM`
`y = zeros(size(x));`
`y(1) = x(1);`
`for n=2:length(x)`
`  y(n)=y(n-1) + x(n);`
`end`
 	
`% VERIFY RESULTS`
`yExpected=cumsum(x);`
	Issue
	
	Generation of test input and verification of results are intermingled with the algorithm code.
	
	Fix
	
	Create a test file that is separate from your algorithm.
		Test file
		
		`% TEST INPUT`
		`x = randn(100,1);`
		
		`% ALGORITHM`
		`y = cumulative_sum(x);`
		
		`% VERIFY RESULTS`
		`yExpected = cumsum(x);`
		`plot(y-yExpected)`
		`title('Error')`

Original code	Best Practice	Modified code

`plot(y-yExpected)`
`title('Error')`
	Put the algorithm in its own function.	Algorithm in its own function
		`function y = cumulative_sum(x)`
		`  y = zeros(size(x));`
		`  y(1) = x(1);`
		`  for n=2:length(x)`
		`    y(n) = y(n-1) + x(n);`
		`  end`
		`end`

You can use the test file to:

- Verify that your floating-point algorithm behaves as you expect before you convert it to fixed point. The floating-point algorithm behavior is the baseline against which you compare the behavior of the fixed-point versions of your algorithm.
- Propose fixed-point data types.
- Compare the behavior of the fixed-point versions of your algorithm to the floating-point baseline.
- Help you determine initial values for static ranges.

By default, the MATLAB Coder app shows code coverage results. Your test files should exercise the algorithm over its full operating range so that the simulation ranges are accurate. For example, for a filter, realistic inputs are impulses, sums of sinusoids, and chirp signals. With these inputs, using linear theory, you can verify that the outputs are correct. Signals that produce maximum output are useful for verifying that your system does not overflow. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results help you verify that your test file is exercising the algorithm adequately. Review code flagged with a red code coverage bar because this code is not executed. If the code coverage is inadequate, modify the test file or add more test files to increase coverage. See “Code Coverage” on page 14-84.

Prepare Your Algorithm for Code Acceleration or Code Generation

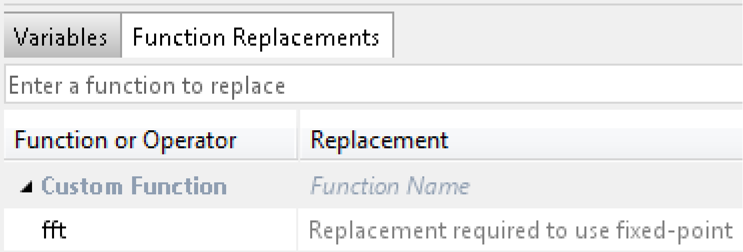
The automated conversion process instruments your code and provides data type proposals to help you convert your algorithm to fixed point.

MATLAB algorithms that you want to convert to fixed point automatically must comply with code generation requirements and rules. To view the subset of the MATLAB language that is supported for code generation, see “Functions and Objects Supported for C/C++ Code Generation — Alphabetical List” on page 3-2.

To help you identify unsupported functions or constructs in your MATLAB code, add the `%#codegen` pragma to the top of your MATLAB file. The MATLAB Code Analyzer flags functions and constructs that are not available in the subset of the MATLAB language supported for code generation. This advice appears in real time as you edit your code in the MATLAB editor. For more information, see “Check Code with the Code Analyzer” on page 18-6. The software provides a link to a report that identifies calls to functions and the use of data types that are not supported for code generation. For more information, see “Check Code by Using the Code Generation Readiness Tool” on page 18-8.

Check for Fixed-Point Support for Functions Used in Your Algorithm

The app flags unsupported function calls found in your algorithm on the **Function Replacements** tab. For example, if you use the `fft` function, which is not supported for fixed point, the tool adds an entry to the table on this tab and indicates that you need to specify a replacement function to use for fixed-point operations.



Function or Operator	Replacement
◢ Custom Function	Function Name
fft	Replacement required to use fixed-point

You can specify additional replacement functions. For example, functions like `sin`, `cos`,and `sqrt` might support fixed point, but for better efficiency, you might want to consider an alternative implementation like a lookup table or CORDIC-based algorithm. The app provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. See “Replacing Functions Using Lookup Table Approximations” on page 14-124.

Manage Data Types and Control Bit Growth

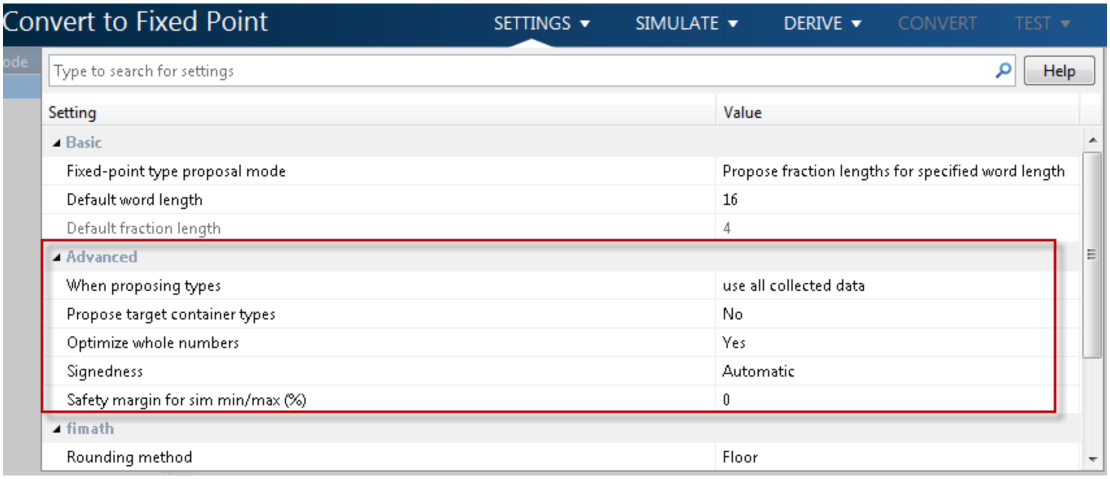
The automated fixed-point conversion process automatically manages data types and controls bit growth. It controls bit growth by using subscripted assignments, that is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. In addition to preventing bit growth, subscripted assignment reduces the number of casts in the generated fixed-point code and makes the code more readable.

Convert to Fixed Point

What Are Your Goals for Converting to Fixed Point?

Before you start the conversion, consider your goals for converting to fixed point. Are you implementing your algorithm in C or HDL? What are your target constraints? The answers to these questions determine many fixed-point properties such as the available word length, fraction length, and math modes, as well as available math libraries.

To set up these properties, use the **Advanced** settings.



Setting	Value
◢ Basic	
Fixed-point type proposal mode	Propose fraction lengths for specified word length
Default word length	16
Default fraction length	4
◢ Advanced	
When proposing types	use all collected data
Propose target container types	No
Optimize whole numbers	Yes
Signedness	Automatic
Safety margin for sim min/max (%)	0
◢ fimath	
Rounding method	Floor

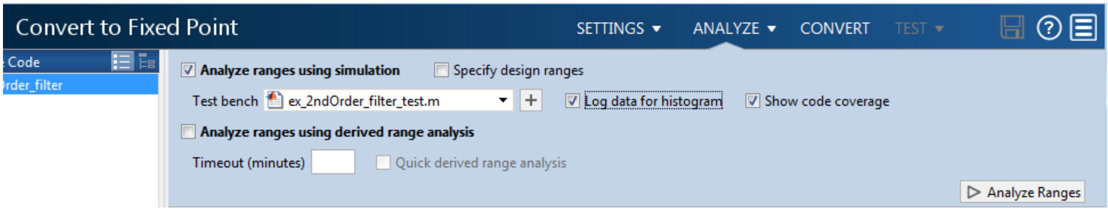
For more information, see “Specify Type Proposal Options” on page 14-35.

Run With Fixed-Point Types and Compare Results

Create a test file to validate that the floating-point algorithm works as expected before converting it to fixed point. You can use the same test file to propose fixed-point data types, and to compare fixed-point results to the floating-point baseline after the conversion. For more information, see “Running a Simulation” on page 14-88 and “Log Data for Histogram” on page 14-99 .

Use the Histogram to Fine-Tune Data Type Settings

To fine-tune fixed-point type settings, use the histogram. To log data for histograms, in the app, click the **Analyze** arrow ▼ and select `Log data for histogram`.



☑ Analyze ranges using simulation

□ Specify design ranges

Test bench ex_2ndOrder_filter_test.m

☑ Log data for histogram

☑ Show code coverage

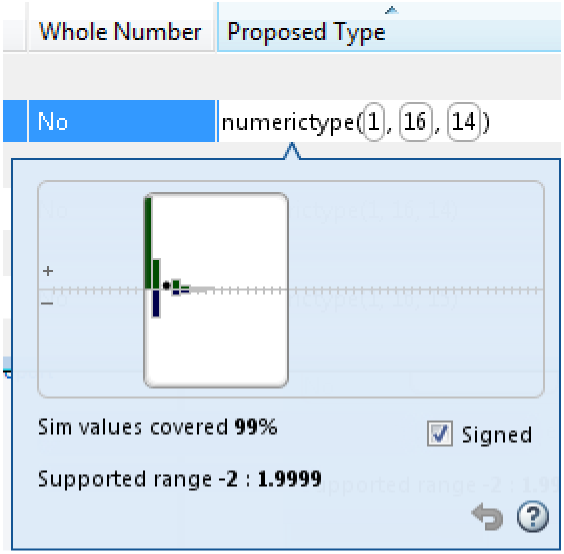
□ Analyze ranges using derived range analysis

Timeout (minutes)

□ Quick derived range analysis

After simulation and static analysis:

- To view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.



Whole Number	Proposed Type
No	numerictype(1, 16, 14)

Sim values covered 99%

☑ Signed

Supported range -2 : 1.9999

You can view the effect of changing the proposed data types by dragging the edges of the bounding box in the histogram window to change the proposed data type and selecting or clearing the **Signed** option.

- If the values overflow and the range cannot fit the proposed type, the table shows proposed types in red.

When the tool applies data types, it generates an html report that provides overflow information and highlights overflows in red. Review the proposed data types.

Optimize Your Algorithm

Use fimath to Get Optimal Types for C or HDL

`fimath` properties define the rules for performing arithmetic operations on `fi` objects, including math, rounding, and overflow properties. You can use the `fimath` `ProductMode` and `SumMode` properties to retain optimal data types for C or HDL. HDL can have arbitrary word length types in the generated HDL code whereas C requires container types (`uint8`, `uint16`, `uint32`). Use the **Advanced** settings, see “Specify Type Proposal Options” on page 14-35.

C

The `KeepLSB` setting for `ProductMode` and `SumMode` models the behavior of integer operations in the C language, while `KeepMSB` models the behavior of many DSP devices. Different rounding methods require different amounts of overhead code. Setting the `RoundingMethod` property to `Floor`, which is equivalent to two's complement truncation, provides the most efficient rounding implementation. Similarly, the standard method for handling overflows is to wrap using modulo arithmetic. Other overflow handling methods create costly logic. Whenever possible, set `OverflowAction` to `Wrap`.

MATLAB Code	Best Practice	Generated C Code

Code being compiled
`function y = adder(a,b)`
`  y = a + b;`
`end`
Note: In the app, set **Default word length** to `16`.
	Issue
	With the default word length set to 16 and the default `fimath` settings, additional code is generated to implement saturation overflow, nearest rounding, and full-precision arithmetic.
		`int adder(short a, short b)`
		`{`
		`  int y;`
		`  int i0;`
		`  int i1;`
		`  int i2;`
		`  int i3;`
		`  i0 = a;`
		`  i1 = b;`
		`  if ((i0 & 65536) != 0) {`
		`    i2 = i0 | -65536;`
		`  } else {`
		`    i2 = i0 & 65535;`
		`  }`

<table>
<tr><th>MATLAB Code</th><th colspan="2">Best Practice</th><th>Generated C Code</th></tr>
<tr><td rowspan="8"></td><td colspan="2"></td><td><pre> if ((i1 & 65536) != 0) {
    i3 = i1 | -65536;
  } else {
    i3 = i1 & 65535;
  }

  i0 = i2 + i3;
  if ((i0 & 65536) != 0) {
    y = i0 | -65536;
  } else {
    y = i0 & 65535;
  }

  return y;
}</pre></td></tr>
<tr><td colspan="2"><b>Fix</b>

To make the generated C code more efficient, choose fixed-point math settings that match your processor types.

To customize fixed-point type proposals, use the app <b>Settings</b>. Select <b>fimath</b> and then set:</td><td rowspan="7"><pre>int adder(short a, short b)
{
  return a + b;
}</pre></td></tr>
<tr><td>Rounding method</td><td>Floor</td></tr>
<tr><td>Overflow action</td><td>Wrap</td></tr>
<tr><td>Product mode</td><td>KeepLSB</td></tr>
<tr><td>Sum mode</td><td>KeepLSB</td></tr>
<tr><td>Product word length</td><td>32</td></tr>
<tr><td>Sum word length</td><td>32</td></tr>
</table>

HDL

For HDL code generation, set:

- `ProductMode` and `SumMode` to `FullPrecision`

- `Overflow action` to `Wrap`
- `Rounding method` to `Floor`

Replace Built-in Functions with More Efficient Fixed-Point Implementations

Some MATLAB built-in functions can be made more efficient for fixed-point implementation. For example, you can replace a built-in function with a Lookup table implementation, or a CORDIC implementation, which requires only iterative shift-add operations. For more information, see “Function Replacements” on page 14-101.

Reimplement Division Operations Where Possible

Often, division is not fully supported by hardware and can result in slow processing. When your algorithm requires a division, consider replacing it with one of the following options:

- Use bit shifting when the denominator is a power of two. For example, `bitsra(x,3)` instead of `x/8`.
- Multiply by the inverse when the denominator is constant. For example, `x*0.2` instead of `x/5`.
- If the divisor is not constant, use a temporary variable for the division. Doing so results in a more efficient data type proposal and, if overflows occur, makes it easier to see which expression is overflowing.

Eliminate Floating-Point Variables

For more efficient code, the automated fixed-point conversion process eliminates floating-point variables. The one exception to this is loop indices because they usually become integer types. It is good practice to inspect the fixed-point code after conversion to verify that there are no floating-point variables in the generated fixed-point code.

Avoid Explicit Double and Single Casts

For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts.

Instead of using casts, supply a replacement function. For more information, see “Function Replacements” on page 14-101.

Replacing Functions Using Lookup Table Approximations

The MATLAB Coder software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations in a MATLAB Coder project, see “Replace the exp Function with a Lookup Table” on page 14-49 and “Replace a Custom Function with a Lookup Table” on page 14-58.

To use lookup table approximations in the programmatic workflow, see `coder.approximation`, “Replace the exp Function with a Lookup Table” on page 15-24, and “Replace a Custom Function with a Lookup Table” on page 15-26.

MATLAB Language Features Supported for Automated Fixed-Point Conversion

Fixed-Point Designer supports the following MATLAB language features in automated fixed-point conversion:

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data (see "Generate Code for Variable-Size Data" on page 20-108). Range computation for variable–sized data is supported via simulation mode only. Variable-sized data is not supported for comparison plotting.
- Subscripting (see "Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation" (Fixed-Point Designer))
- Complex numbers (see "Code Generation for Complex Data" (Fixed-Point Designer))
- Numeric classes (see "Supported Variable Types" (Fixed-Point Designer))
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic (see "Code Acceleration and Code Generation from MATLAB" (Fixed-Point Designer))
- Program control statements `if`, `switch`, `for`, `while`, and `break`
- Arithmetic, relational, and logical operators
- Local functions
- Global variables
- Persistent variables
- Structures, including arrays of structures. Range computation for structures is supported via simulation mode only.
- Characters

  The complete set of Unicode characters is not supported for code generation. Characters are restricted to 8 bits of precision in generated code. Because many mathematical operations require more than 8 bits of precision, it is recommended that you do not perform arithmetic with characters if you intend to convert your MATLAB algorithm to fixed point.
- MATLAB classes. Range computation for MATLAB classes is supported via simulation mode only.

  Automated conversion supports:

  - Class properties
  - Constructors
  - Methods
  - Specializations

  It does not support class inheritance or packages. For more information, see “Fixed-Point Code for MATLAB Classes” (Fixed-Point Designer).
- Ability to call functions (see “Resolution of Function Calls for Code Generation” on page 13-2)
- Subset of MATLAB toolbox functions (see “Functions Supported for Code Acceleration or C Code Generation” (Fixed-Point Designer)).
- Subset of DSP System Toolbox System objects.

  The DSP System Toolbox System objects supported for automated conversion are:

  - dsp.ArrayVectorAdder
  - dsp.BiquadFilter
  - dsp.FIRDecimator
  - dsp.FIRInterpolator
  - dsp.FIRFilter (Direct Form and Direct Form Transposed only)
  - dsp.FIRRateConverter
  - dsp.LowerTriangularSolver
  - dsp.LUFactor
  - dsp.UpperTriangularSolver
  - dsp.VariableFractionalDelay
  - dsp.Window

Inspecting Data Using the Simulation Data Inspector

In this section...

What Is the Simulation Data Inspector?

The Simulation Data Inspector allows you to view data logged during the fixed-point conversion process. You can use it to inspect and compare the inputs and outputs to the floating-point and fixed-point versions of your algorithm.

For fixed-point conversion, there is no programmatic interface for the Simulation Data Inspector.

Import Logged Data

Before importing data into the Simulation Data Inspector, you must have previously logged data to the base workspace or to a MAT-file.

Export Logged Data

The Simulation Data Inspector provides the capability to save data collected by the fixed-point conversion process to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

Group Signals

You can customize the organization of your logged data in the Simulation Data Inspector **Runs** pane. By default, data is first organized by run. You can then organize your data by logged variable or no hierarchy.

Run Options

You can configure the Simulation Data Inspector to:

- Append New Runs

  In the Run Options dialog box, the default is set to add new runs to the bottom of the run list. To append new runs to the top of the list, select **Add new runs at top**.
- Specify a Run Naming Rule

  To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Options**.

Create Report

You can create a report of the runs or comparison plots. Specify the name and location of the report file. By default, the Simulation Data Inspector overwrites existing files. To preserve existing reports, select **If report exists, increment file name to prevent overwriting**.

Comparison Options

To change how signals are matched when runs are compared, specify the **Align by** and **Then by** parameters and then click **OK**.

Enabling Plotting Using the Simulation Data Inspector

To enable the Simulation Data Inspector in the Fixed-Point Conversion tool, see “Enable Plotting Using the Simulation Data Inspector” on page 14-67.

To enable the Simulation Data Inspector in the programmatic workflow, see “Enable Plotting Using the Simulation Data Inspector” on page 15-28.

Save and Load Simulation Data Inspector Sessions

If you have data in the Simulation Data Inspector and you want to archive or share the data to view in the Simulation Data Inspector later, save the Simulation Data Inspector session. When you save a Simulation Data Inspector session, the MAT-file contains:

- All runs, data, and properties from the **Runs** and **Comparisons** panes.
- Check box selection state for data in the **Runs** pane.

Save a Session to a MAT-File

1. On the **Visualize** tab, click **Save**.
2. Browse to where you want to save the MAT-file to, name the file, and click **Save**.

Load a Saved Simulation Data Inspector Simulation

1. On the **Visualize** tab, click **Open**.
2. Browse, select the MAT-file saved from the Simulation Data Inspector, and click **Open**.
3. If data in the session is plotted on multiple subplots, on the **Format** tab, click **Subplots** and select the subplot layout.

Custom Plot Functions

The Fixed-Point Conversion tool provides a default time series based plotting function. The conversion process uses this function at the test numerics step to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. For example, plots that show eye diagrams and bit error differences are more suitable in the communications domain and histogram difference plots are more suitable in image processing designs.

You can choose to use a custom plot function at the test numerics step. The Fixed-Point Conversion tool facilitates custom plotting by providing access to the raw logged input and output data before and after fixed-point conversion. You supply a custom plotting function to visualize the differences between the floating-point and fixed-point results. If you specify a custom plot function, the fixed-point conversion process calls the function for each input and output variable, passes in the name of the variable and the function that uses it, and the results of the floating-point and fixed-point simulations.

Your function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.

  Use this information to:

  - Customize plot headings and axes.
  - Choose which variables to plot.
  - Generate different error metrics for different output variables.
- A cell array to hold the logged floating-point values for the variable.

  This cell array contains values observed during floating-point simulation of the algorithm during the test numerics phase. You might need to reformat this raw data.
- A cell array to hold the logged values for the variable after fixed-point conversion.

  This cell array contains values observed during fixed-point simulation of the converted design.

For example, `function customComparisonPlot(varInfo, floatVarVals, fixedPtVarVals)`.

To use a custom plot function, in the Fixed-Point Conversion tool, select **Advanced**, and then set **Custom plot function** to the name of your plot function.

In the programmatic workflow, set the coder.FixptConfig configuration object `PlotFunction` property to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 15-29.

Data Type Issues in Generated Code

Within the fixed-point conversion HTML report, you have the option to highlight MATLAB code that results in double, single, or expensive fixed-point operations. Consider enabling these checks when trying to achieve a strict single, or fixed-point design.

These checks are disabled by default.

Enable the Highlight Option in the MATLAB Coder App

1. On the **Convert to Fixed Point** page, to open the **Settings** dialog box, click the **Settings** arrow ▼.
2. Under **Plotting and Reporting**, set **Highlight potential data type issues** to `Yes`.

When conversion is complete, open the fixed-point conversion HTML report to view the highlighting. Click **View report** in the **Type Validation Output** tab.

Enable the Highlight Option at the Command Line

1. Create a fixed-point code configuration object:

   ```
   cfg = coder.config('fixpt');
   ```
2. Set the `HighlightPotentialDataTypeIssues` property of the configuration object to `true`.

   ```
   cfg.HighlightPotentialDataTypeIssues = true;
   ```

Stowaway Doubles

When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone. This check highlights all expressions that result in a double operation.

For a strict-single precision design, specify a standard math library that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Standard math library** to `C99 (ISO)`.

Stowaway Singles

This check highlights all expressions that result in a single operation.

Expensive Fixed-Point Operations

The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see “Tips for Making Generated Code More Efficient” (Fixed-Point Designer).

Cumbersome Operations

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that has word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local fimath properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

Automated Fixed-Point Conversion Using Programmatic Workflow

Convert MATLAB Code to Fixed-Point C Code

This example shows how to generate fixed-point C code from floating-point MATLAB code using the programmatic workflow.

Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name. For example:

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'fun_with_matlab_test';
```

Configure the Fixed-Point Configuration Object for Type Proposal

The fixed-point conversion software can propose types based on simulation ranges, derived ranges, or both.

- For type proposal using only simulation ranges, enable the collection and reporting of simulation range data. By default, derived range analysis is disabled.

  ```
  fixptcfg.ComputeSimulationRanges = true;
  ```
- For type proposal using only derived ranges:

  1. Specify the design range for input parameters. For example:

     ```
     fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
     ```
  2. Enable derived range analysis. Disable collection and reporting of simulation range data.

     ```
     fixptcfg.ComputeDerivedRanges = true;
     fixptcfg.ComputeSimulationRanges = false;
     ```

Enable Numerics Testing

Select to run the test file to verify the generated fixed-point MATLAB code.

```
fixptcfg.TestNumerics = true;
```

Enable Plotting

Log inputs and outputs for comparison plotting. Select to plot using a custom function or Simulation Data Inspector. For example, to plot using Simulation Data Inspector:

```
fixptcfg.LogIOForComparisonPlotting = true;
fixptcfg.PlotWithSimulationDataInspector = true;
```

Configure Additional Fixed-Point Configuration Object Properties

Configure additional fixed-point configuration object properties as necessary. For example, define the default fixed-point word length:

```
fixptcfg.DefaultWordLength = 16;
```

Set Up the C Code Generation Configuration Object

Create a code configuration object for generation of a C static library, dynamic library, or executable. Enable the code generation report. For example:

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

Generate Fixed-Point C Code

Use the `codegen` function to convert the floating-point MATLAB function to fixed-point C code. For example:

```
codegen -float2fixed fixptcfg -config cfg fun_with_matlab
```

View the Type Proposal Report

Click the link to the type proposal report for the entry-point function.

View the Comparison Plots

If you selected to log inputs and outputs for comparison plots, the conversion process generates comparison plots.

- If you selected to use Simulation Data Inspector for these plots, the Simulation Data Inspector opens. Use Simulation Data Inspector to view and compare the floating-point and fixed-point run information.
- If you selected to use a custom plotting function for these plots, the conversion process uses the custom function to generate the plots.

View the Generated Fixed-Point MATLAB and Fixed-Point C Code

Click the **View Report** link that follows the type proposal report. To view the fixed-point MATLAB code, click the **MATLAB code** tab. To view the fixed-point C code, click the **C code** tab.

See Also

coder.FixptConfig

Related Examples

- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 15-5
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-11
- “Enable Plotting Using the Simulation Data Inspector” on page 15-28

More About

- “Automated Fixed-Point Conversion” on page 14-83

Propose Fixed-Point Data Types Based on Simulation Ranges

This example shows how to propose fixed-point data types based on simulation range data using the `codegen` function.

Prerequisites

To complete this example, you must install the following products:

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

  You can use `mex -setup` to change the default compiler. See "Change Default Compiler" (MATLAB).

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\ex_2ndOrder_filter`.
2. Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

Type	Name	Description
Function code	`ex_2ndOrder_filter.m`	Entry-point MATLAB function
Test file	`ex_2ndOrder_filter_test.m`	MATLAB script that tests `ex_2ndOrder_filter.m`

The ex_2ndOrder_filter Function

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
```

```
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i = 1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

The ex_2ndOrder_filter_test Script

The test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse to cover the full intended operating range of the system. The script then plots the outputs.

```
% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256;                     % Number of points
t = linspace(0,1,N);         % Time vector from 0 to 1 second
f1 = N/2;                    % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N);          % Step
x_impulse = zeros(1,N);      % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
  y(i,:) = ex_2ndOrder_filter(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'}
clf
for i = 1:size(x,1)
  subplot(size(x,1),1,i)
  plot(t,x(i,:),t,y(i,:))
  title(titles{i})
  legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'ex_2ndOrder_filter_test';
```

Set Up the C Code Generation Configuration Object

Create a code configuration object to generate a C static library. Enable the code generation report.

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

Collect Simulation Ranges and Generate Fixed-Point Code

Use the `codegen` function to convert the floating-point MATLAB function, `ex_2ndOrder_filter`, to fixed-point C code. Set the default word length for the fixed-point data types to 16.

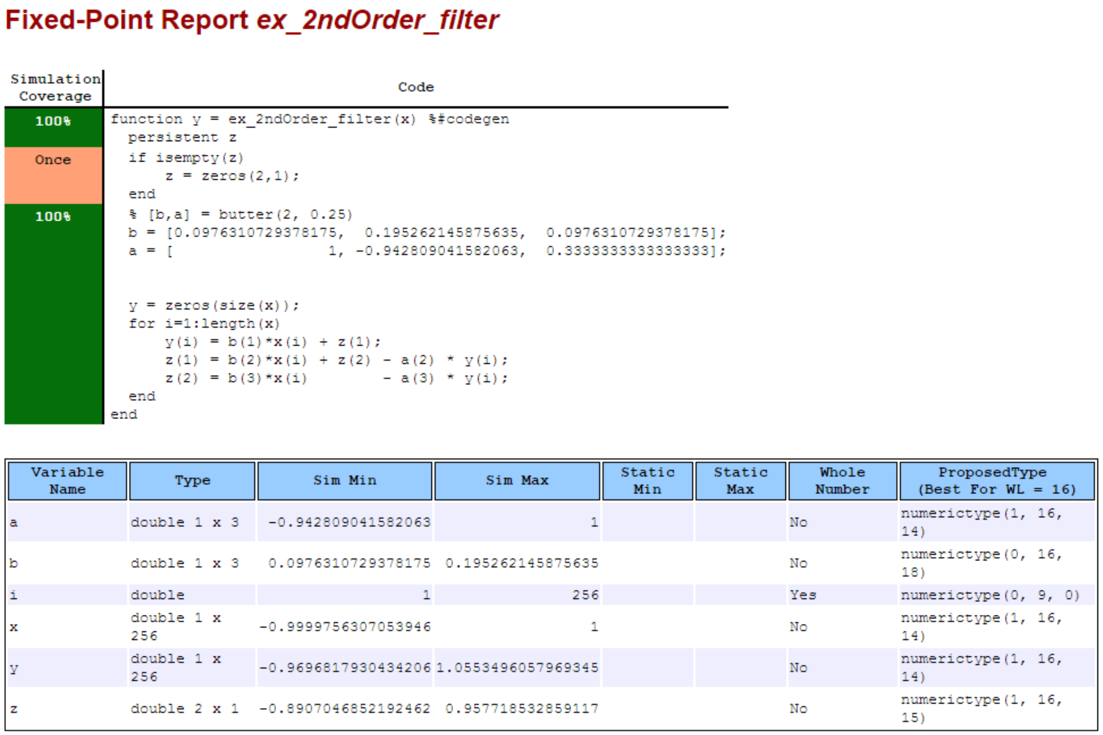
```
fixptcfg.ComputeSimulationRanges = true;
fixptcfg.DefaultWordLength = 16;

% Derive ranges  and generate fixed-point code
codegen -float2fixed fixptcfg -config cfg ex_2ndOrder_filter
```

`codegen` analyzes the floating-point code. Because you did not specify the input types for the `ex_2ndOrder_filter` function, the conversion process infers types by simulating the test file. The conversion process then derives ranges for variables in the algorithm. It uses these derived ranges to propose fixed-point types for these variables. When the conversion is complete, it generates a type proposal report.

View Range Information

Click the link to the type proposal report for the `ex_2ndOrder_filter` function, `ex_2ndOrder_filter_report.html`.

The report opens in a web browser.



Fixed-Point Report *ex_2ndOrder_filter*

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                 1, -0.942809041582063,  0.333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

Variable Name	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	ProposedType (Best For WL = 16)
a	double 1 x 3	-0.942809041582063	1			No	numerictype(1, 16, 14)
b	double 1 x 3	0.0976310729378175	0.195262145875635			No	numerictype(0, 16, 18)
i	double	1	256			Yes	numerictype(0, 9, 0)
x	double 1 x 256	-0.9999756307053946	1			No	numerictype(1, 16, 14)
y	double 1 x 256	-0.9696817930434206	1.0553496057969345			No	numerictype(1, 16, 14)
z	double 2 x 1	-0.8907046852192462	0.957718532859117			No	numerictype(1, 16, 15)

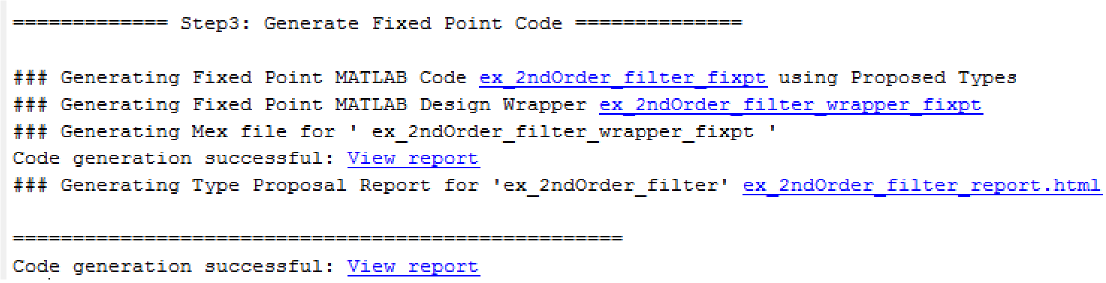
View Generated Fixed-Point MATLAB Code

`codegen` generates a fixed-point version of the `ex_2ndOrder_filter.m` function, `ex_2ndOrder_filter_fixpt.m`, and a wrapper function that calls `ex_2ndOrder_filter_fixpt`. These files are generated in the `codegen\ex_2ndOrder_filter\fixpt` folder in your local working folder.

```
function y = ex_2ndOrder_filter_fixpt(x) %#codegen
  fm = get_fimath();

  persistent z
  if isempty(z)
      z = fi(zeros(2,1), 1, 16, 15, fm);
  end
  % [b,a] = butter(2, 0.25)
  b = fi([0.0976310729378175,  0.195262145875635,...
  0.0976310729378175], 0, 16, 18, fm);
  a = fi([                 1, -0.942809041582063,...
  0.3333333333333333], 1, 16, 14, fm);


  y = fi(zeros(size(x)), 1, 16, 14, fm);
  for i=1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
      z(2) = fi_signed(b(3)*x(i))        - a(3) * y(i);
  end
end



function y = fi_signed(a)
    coder.inline( 'always' );
    if isfi( a ) && ~(issigned( a ))
        nt = numerictype( a );
        new_nt = numerictype( 1, nt.WordLength + 1, nt.FractionLength );
        y = fi( a, new_nt, fimath( a ) );
    else
        y = a;
    end
end

function fm = get_fimath()
 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode',...
 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision',...
 'MaxSumWordLength', 128);
end
```

View Generated Fixed-Point C Code

To view the code generation report for the C code generation, click the **View Report** link that follows the type proposal report.



```
============= Step3: Generate Fixed Point Code ==============

### Generating Fixed Point MATLAB Code ex_2ndOrder_filter_fixpt using Proposed Types
### Generating Fixed Point MATLAB Design Wrapper ex_2ndOrder_filter_wrapper_fixpt
### Generating Mex file for ' ex_2ndOrder_filter_wrapper_fixpt '
Code generation successful: View report
### Generating Type Proposal Report for 'ex_2ndOrder_filter' ex_2ndOrder_filter_report.html

===================================================
Code generation successful: View report
```

The code generation report opens and displays the generated code for `ex_2ndOrder_filter_fixpt.c`.

See Also

coder.FixptConfig | `codegen`

Related Examples

- “Convert MATLAB Code to Fixed-Point C Code” on page 14-5
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-11

Propose Fixed-Point Data Types Based on Derived Ranges

This example shows how to propose fixed-point data types based on static ranges using the codegen function. The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time so you can save time by deriving ranges instead.

Prerequisites

To complete this example, you must install the following products:

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

  You can use mex -setup to change the default compiler. See "Change Default Compiler" (MATLAB).

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the dti.m and dti_test.m files to your local working folder.

Type	Name	Description
Function code	dti.m	Entry-point MATLAB function
Test file	dti_test.m	MATLAB script that tests dti.m

The dti Function

The dti function implements a Discrete Time Integrator in MATLAB.

```
function [y, clip_status] = dti(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
init_val = 1;
gain_val = 1;
limit_upper = 500;
limit_lower = -500;

% variable to hold state between consecutive calls to this block
persistent u_state
if isempty(u_state)
    u_state = init_val+1;
end

% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
 y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
 y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end

% Update State
tprod = gain_val * u_in;
u_state = y + tprod;

function b = subFunction(a)
b = a*a;
```

The dti_test Function

The test script runs the dti function with a sine wave input. The script then plots the input and output signals.

```
% dti_test
% cleanup
clear dti

% input signal
x_in = sin(2.*pi.*(0:0.001:2)).';

pause(10)

len = length(x_in);
y_out = zeros(1,len);
is_clipped_out = zeros(1,len);

for ii=1:len
    data = x_in(ii);
    % call to the dti function
    init_val = 0;
    gain_val = 1;
    upper_limit = 500;
    lower_limit = -500;

    % call to the design that does DTI
    [y_out(ii), is_clipped_out(ii)] = dti(data);

end

figure('Name', [mfilename, '_plot'])
subplot(2,1,1)
plot(1:len,x_in)
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (Sin)')

subplot(2,1,2)
plot(1:len,y_out)
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (DTI)')
```

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

Set Up the Fixed-Point Configuration Object

Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
```

Specify Design Ranges

Specify design range information for the `dti` function input parameter `u_in`.

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
```

Enable Plotting Using the Simulation Data Inspector

Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;
fixptcfg.LogIOForComparisonPlotting = true;
fixptcfg.PlotWithSimulationDataInspector = true;
```

Set Up the C Code Generation Configuration Object

Create a code configuration object to generate a C static library. Enable the code generation report.

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

Derive Ranges and Generate Fixed-Point Code

Use the `codegen` function to convert the floating-point MATLAB function, `dti`, to fixed-point C code. Set the default word length for the fixed-point data types to 16.

```
fixptcfg.ComputeDerivedRanges = true;
fixptcfg.ComputeSimulationRanges = false;
fixptcfg.DefaultWordLength = 16;

% Derive ranges  and generate fixed-point code
```

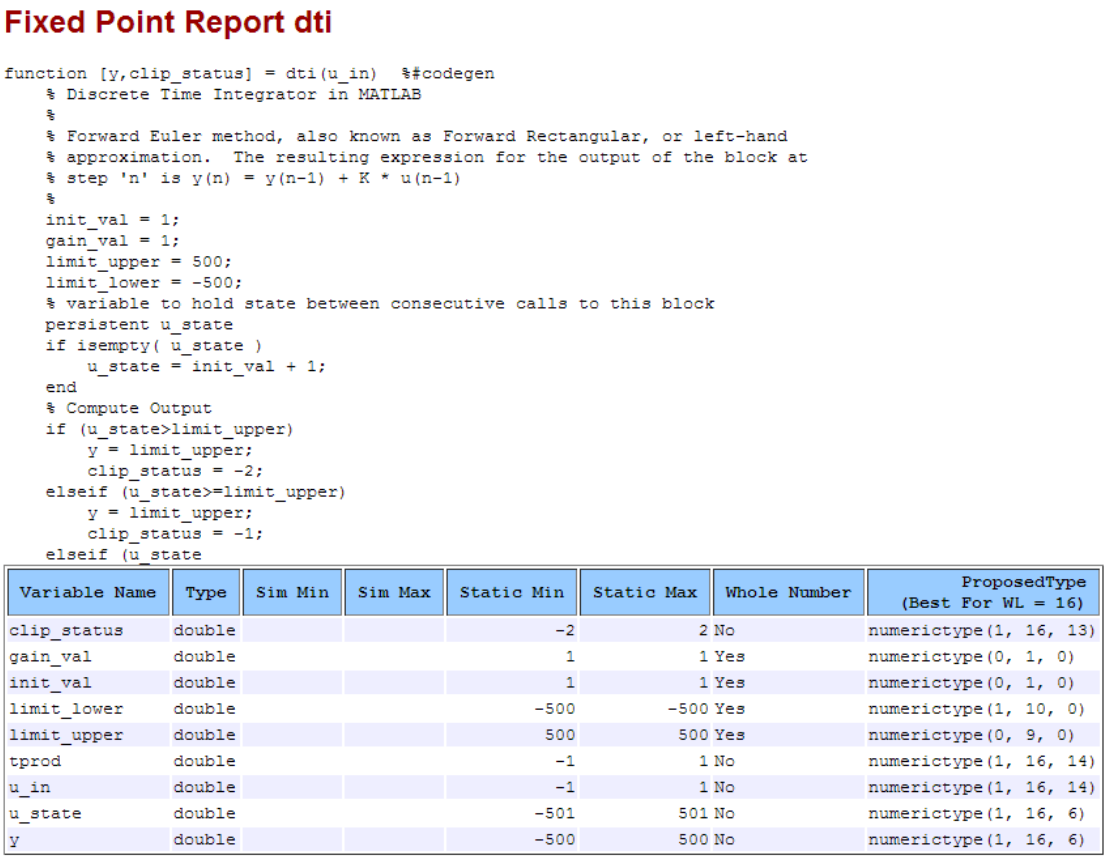
```
codegen -float2fixed fixptcfg -config cfg dti
```

`codegen` analyzes the floating-point code. Because you did not specify the input types for the `dti` function, the conversion process infers types by simulating the test file. The conversion process then derives ranges for variables in the algorithm. It uses these derived ranges to propose fixed-point types for these variables. When the conversion is complete, it generates a type proposal report.

View Derived Range Information

Click the link to the type proposal report for the `dti` function, `dti_report.html`.

The report opens in a web browser.



Fixed Point Report dti

```
function [y,clip_status] = dti(u_in)  %#codegen
    % Discrete Time Integrator in MATLAB
    %
    % Forward Euler method, also known as Forward Rectangular, or left-hand
    % approximation.  The resulting expression for the output of the block at
    % step 'n' is y(n) = y(n-1) + K * u(n-1)
    %
    init_val = 1;
    gain_val = 1;
    limit_upper = 500;
    limit_lower = -500;
    % variable to hold state between consecutive calls to this block
    persistent u_state
    if isempty( u_state )
        u_state = init_val + 1;
    end
    % Compute Output
    if (u_state>limit_upper)
        y = limit_upper;
        clip_status = -2;
    elseif (u_state>=limit_upper)
        y = limit_upper;
        clip_status = -1;
    elseif (u_state
```

Variable Name	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	ProposedType (Best For WL = 16)
clip_status	double			-2	2	No	numerictype(1, 16, 13)
gain_val	double			1	1	Yes	numerictype(0, 1, 0)
init_val	double			1	1	Yes	numerictype(0, 1, 0)
limit_lower	double			-500	-500	Yes	numerictype(1, 10, 0)
limit_upper	double			500	500	Yes	numerictype(0, 9, 0)
tprod	double			-1	1	No	numerictype(1, 16, 14)
u_in	double			-1	1	No	numerictype(1, 16, 14)
u_state	double			-501	501	No	numerictype(1, 16, 6)
y	double			-500	500	No	numerictype(1, 16, 6)

View Generated Fixed-Point MATLAB Code

codegen generates a fixed-point version of the dti function, dti_fxpt.m, and a wrapper function that calls dti_fxpt. These files are generated in the codegen\dti \fixpt folder in your local working folder.

```
function [y, clip_status] = dti_fixpt(u_in) %#codegen
% Discrete Time Integrator in MATLAB
%
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
fm = get_fimath();

init_val = fi(1, 0, 1, 0, fm);
gain_val = fi(1, 0, 1, 0, fm);
limit_upper = fi(500, 0, 9, 0, fm);
limit_lower = fi(-500, 1, 10, 0, fm);

% variable to hold state between consecutive calls to this block
persistent u_state;
if isempty(u_state)
    u_state = fi(init_val+fi(1, 0, 1, 0, fm), 1, 16, 6, fm);
end

% Compute Output
if (u_state > limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-2, 1, 16, 13, fm);
elseif (u_state >= limit_upper)
    y = fi(limit_upper, 1, 16, 6, fm);
    clip_status = fi(-1, 1, 16, 13, fm);
elseif (u_state < limit_lower)
 y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(2, 1, 16, 13, fm);
elseif (u_state <= limit_lower)
 y = fi(limit_lower, 1, 16, 6, fm);
    clip_status = fi(1, 1, 16, 13, fm);
else
    y = fi(u_state, 1, 16, 6, fm);
    clip_status = fi(0, 1, 16, 13, fm);
end
```

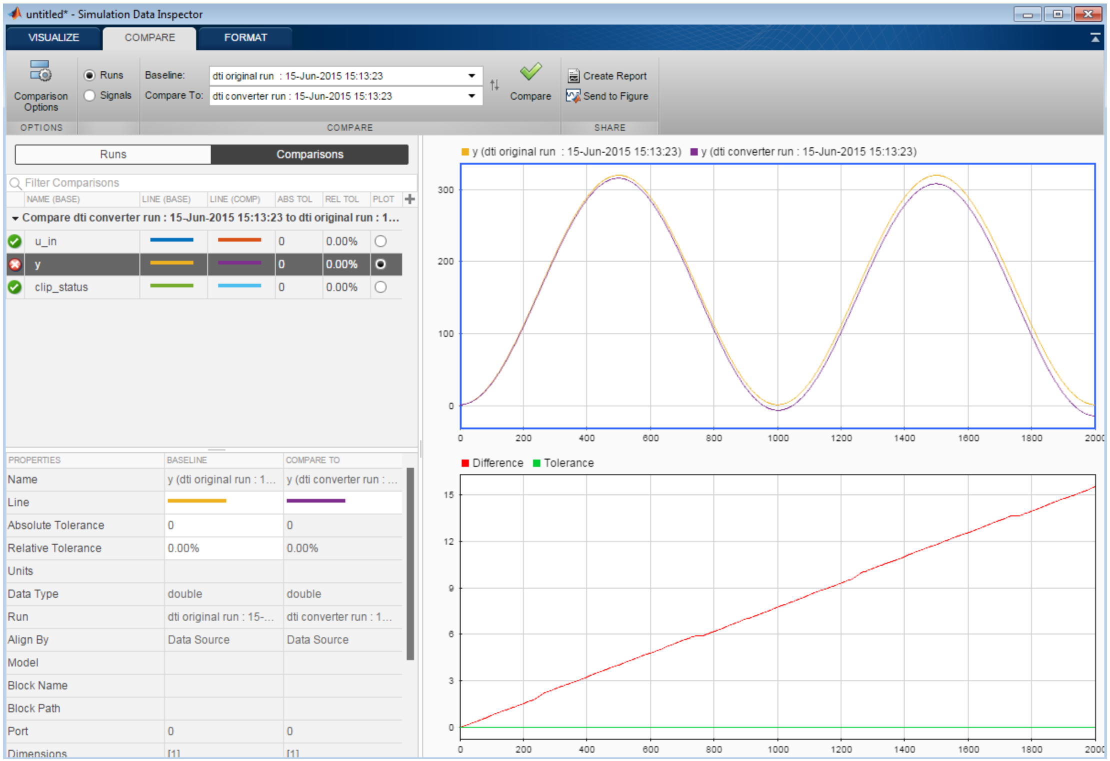
```
% Update State
tprod = fi(gain_val * u_in, 1, 16, 14, fm);
u_state(:) = y + tprod;
end


function fm = get_fimath()
 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode',...
 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'FullPrecision',...
 'MaxSumWordLength', 128);
end
```

Compare Floating-Point and Fixed-Point Runs

Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.

You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-point values for the output `y`, on the **Compare** tab, select `y`, and then click **Compare Runs**.

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



● Runs
○ Signals

Baseline: dti original run : 15-Jun-2015 15:13:23

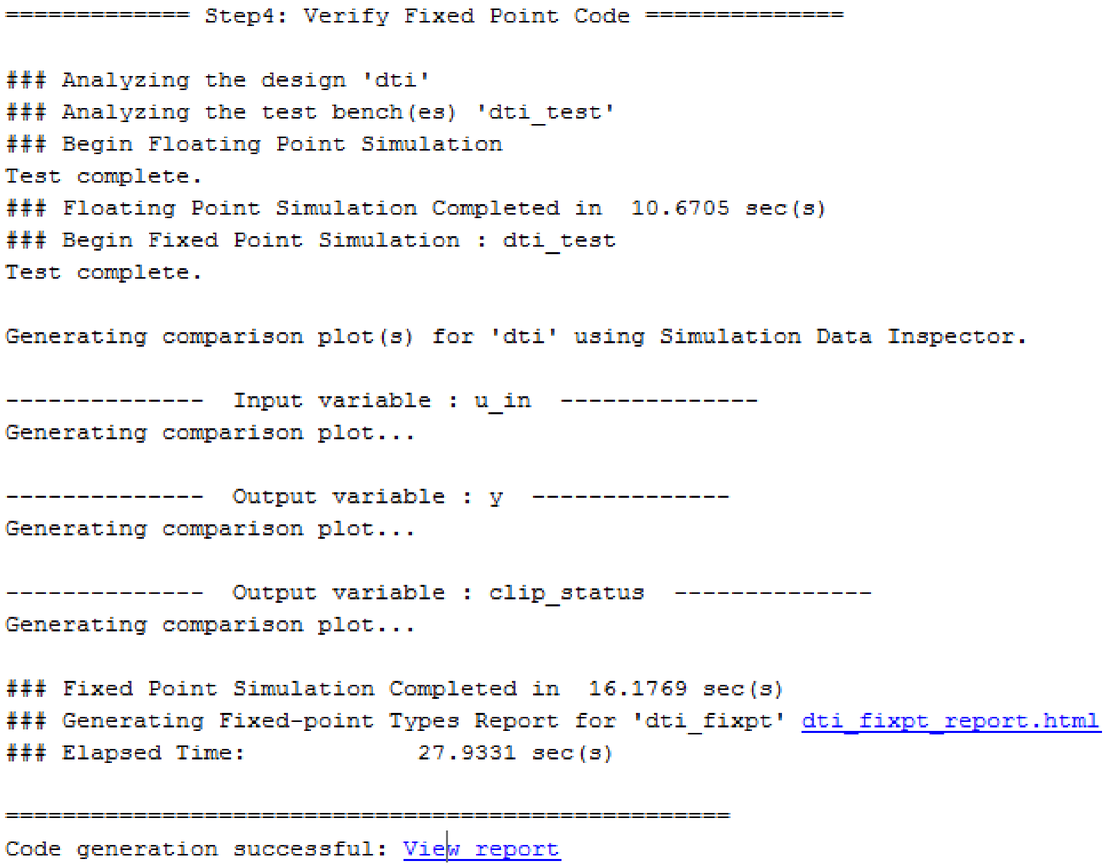
Compare To: dti converter run : 15-Jun-2015 15:13:23

	NAME (BASE)	LINE (BASE)	LINE (COMP)	ABS TOL	REL TOL	PLOT
▾	Compare dti converter run : 15-Jun-2015 15:13:23 to dti original run : 1...					
✓	u_in	━	━	0	0.00%	○
☒	y	━	━	0	0.00%	●
✓	clip_status	━	━	0	0.00%	○

PROPERTIES	BASELINE	COMPARE TO
Name	y (dti original run : 1...	y (dti converter run : ...
Line	━	━
Absolute Tolerance	0	0
Relative Tolerance	0.00%	0.00%
Units		
Data Type	double	double
Run	dti original run : 15-...	dti converter run : 1...
Align By	Data Source	Data Source
Model		
Block Name		
Block Path		
Port	0	0
Dimensions	[1]	[1]

View Generated Fixed-Point C Code

To view the code generation report for the C code generation, click the **View Report** link that follows the type proposal report.



```
============= Step4: Verify Fixed Point Code ==============

### Analyzing the design 'dti'
### Analyzing the test bench(es) 'dti_test'
### Begin Floating Point Simulation
Test complete.
### Floating Point Simulation Completed in  10.6705 sec(s)
### Begin Fixed Point Simulation : dti_test
Test complete.

Generating comparison plot(s) for 'dti' using Simulation Data Inspector.

--------------  Input variable : u_in  --------------
Generating comparison plot...

--------------  Output variable : y  --------------
Generating comparison plot...

--------------  Output variable : clip_status  --------------
Generating comparison plot...

### Fixed Point Simulation Completed in  16.1769 sec(s)
### Generating Fixed-point Types Report for 'dti_fixpt' dti_fixpt_report.html
### Elapsed Time:             27.9331 sec(s)

===================================================
Code generation successful: View report
```

The code generation report opens and displays the generated code for `dti_fixpt.c`.

See Also

coder.FixptConfig | `codegen`

Related Examples

- "Convert MATLAB Code to Fixed-Point C Code" on page 14-5
- "Propose Fixed-Point Data Types Based on Simulation Ranges" on page 15-5

Detect Overflows

This example shows how to detect overflows at the command line. At the numerical testing stage in the conversion process, the tool simulates the fixed-point code using scaled doubles. It then reports which expressions in the generated code produce values that would overflow the fixed-point data type.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer

In a local, writable folder, create a function, `overflow`.

```
function y = overflow(b,x,reset)
    if nargin<3, reset = true; end
    persistent z p
    if isempty(z) || reset
        p = 0;
        z = zeros(size(b));
    end
    [y,z,p] = fir_filter(b,x,z,p);
end
function [y,z,p] = fir_filter(b,x,z,p)
    y = zeros(size(x));
    nx = length(x);
    nb = length(b);
    for n = 1:nx
        p=p+1; if p>nb, p=1; end
        z(p) = x(n);
        acc = 0;
        k = p;
        for j=1:nb
            acc = acc + b(j)*z(k);
            k=k-1; if k<1, k=nb; end
        end
        y(n) = acc;
    end
end
```

Create a test file, `overflow_test.m` to exercise the `overflow` algorithm.

```
function overflow_test
    % The filter coefficients were computed using the FIR1 function from
    % Signal Processing Toolbox.
    %   b = fir1(11,0.25);
    b = [-0.004465461051254
         -0.004324228005260
         +0.012676739550326
         +0.074351188907780
         +0.172173206073645
         +0.249588554524763
         +0.249588554524763
         +0.172173206073645
         +0.074351188907780
         +0.012676739550326
         -0.004324228005260
         -0.004465461051254]';

    % Input signal
    nx = 256;
    t = linspace(0,10*pi,nx)';

    % Impulse
    x_impulse = zeros(nx,1); x_impulse(1) = 1;

    % Max Gain
    % The maximum gain of a filter will occur when the inputs line up with the
    % signs of the filter's impulse response.
    x_max_gain = sign(b)';
    x_max_gain = repmat(x_max_gain,ceil(nx/length(b)),1);
    x_max_gain = x_max_gain(1:nx);


    % Sums of sines
    f0=0.1; f1=2;
    x_sines = sin(2*pi*t*f0) + 0.1*sin(2*pi*t*f1);

    % Chirp
    f_chirp = 1/16;                  % Target frequency
    x_chirp = sin(pi*f_chirp*t.^2);  % Linear chirp

    x = [x_impulse, x_max_gain, x_sines, x_chirp];
    titles = {'Impulse', 'Max gain', 'Sum of sines', 'Chirp'};
    y = zeros(size(x));

    for i=1:size(x,2)
        reset = true;
        y(:,i) = overflow(b,x(:,i),reset);
    end

    test_plot(1,titles,t,x,y)

end
function test_plot(fig,titles,t,x,y1)
    figure(fig)
    clf
    sub_plot = 1;
    font_size = 10;
    for i=1:size(x,2)
        subplot(4,1,sub_plot)
        sub_plot = sub_plot+1;
        plot(t,x(:,i),'c',t,y1(:,i),'k')
        axis('tight')
        xlabel('t','FontSize',font_size);
        title(titles{i},'FontSize',font_size);
        ax = gca;
        ax.FontSize = 10;
    end
    figure(gcf)
end
```

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `overflow_test`.

```
fixptcfg.TestBenchName = 'overflow_test';
```

Set the default word length to 16.

```
fixptcfg.DefaultWordLength = 16;
```

Enable overflow detection.

```
fixptcfg.TestNumerics = true;
fixptcfg.DetectFixptOverflows = true;
```

Set the `fimath` `Product mode` and `Sum mode` to `KeepLSB`. These settings models the behavior of integer operations in the C language.

```
fixptcfg.fimath = ...
['fimath(''RoundingMethod'',''Floor'',''OverflowAction'',' ...
'''Wrap'',''ProductMode'',''KeepLSB'',''SumMode'',''KeepLSB'')'];
```

Create a code generation configuration object to generate a standalone C static library.

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

Convert the floating-point MATLAB function, `overflow`, to fixed-point C code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -float2fixed fixptcfg -config cfg overflow
```

The numerics testing phase reports an overflow.

```
Overflow error in expression 'acc + b( j )*z( k )'. Percentage of Current Range = 104%.
```

Determine if the addition or the multiplication in this expression overflowed. Set the `fimath` ProductMode to `FullPrecision` so that the multiplication will not overflow, and then run the `codegen` command again.

```
fixptcfg.fimath = ['fimath(''RoundingMethod'',''Floor'',''OverflowAction'',' ...
 '''Wrap'',''ProductMode'',''FullPrecision'',''SumMode'',''KeepLSB'')'];
codegen -float2fixed fixptcfg -config cfg overflow
```

The numerics testing phase still reports an overflow, indicating that it is the addition in the expression that is overflowing.

Replace the exp Function with a Lookup Table

This example shows how to replace the exp function with a lookup table approximation in the generated fixed-point code using the codegen function.

Prerequisites

To complete this example, you must install the following products:

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

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

Create Algorithm and Test Files

1 Create a MATLAB function, my_fcn.m, that calls the exp function.

```
function y = my_fcn(x)
  y = exp(x);
end
```

2 Create a test file, my_fcn_test.m, that uses my_fcn.m.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Configure Approximation

Create a function replacement configuration object to approximate the exp function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

Set Up Configuration Object

Create a coder.FixptConfig object, fixptcfg. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'my_fcn_test';
fixptcfg.TestNumerics = true;
fixptcfg.DefaultWordLength = 16;
fixptcfg.addApproximation(q);
```

Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to my_fcn_fixpt.

The generated code contains a lookup table approximation, replacement_exp, for the exp function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, my_fcn_fixpt, calls this approximation instead of calling exp.

```
function y = my_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_exp(x), 0, 16, 1, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

Replace a Custom Function with a Lookup Table

This example shows how to replace a custom function with a lookup table approximation function using the `codegen` function.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create a function replacement configuration object to approximate custom_fcn. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...
                        'CandidateFunction',@custom_fcn, 'NumberOfPoints',50);
```

Create a coder.FixptConfig object, fixptcfg. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'custom_test';
fixptcfg.TestNumerics = true;
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg call_custom_fcn
```

codegen generates fixed-point MATLAB code in call_custom_fcn_fixpt.m.

To view the generated fixed-point code, click the link to call_custom_fcn_fixpt.

The generated code contains a lookup table approximation, replacement_custom_fcn, for the custom_fcn function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. The lookup table uses 50 points as specified. By default, it uses linear interpolation and the minimum and maximum values detected by running the test file.

The generated fixed-point function, call_custom_fcn_fixpt, calls this approximation instead of calling custom_fcn.

```
function y = call_custom_fcn_fixpt(x)
  fm = get_fimath();

  y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

Enable Plotting Using the Simulation Data Inspector

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point input and output data logged using the codegen function. At the MATLAB command line:

1. Create a fixed-point configuration object and configure the test file name.

   ```
   fixptcfg = coder.config('fixpt');
   fixptcfg.TestBenchName = 'dti_test';
   ```
2. Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

   ```
   fixptcfg.TestNumerics = true;
   fixptcfg.LogIOForComparisonPlotting = true;
   fixptcfg.PlotWithSimulationDataInspector = true;
   ```
3. Generate fixed-point MATLAB code using codegen.

   ```
   codegen -float2fixed fixptcfg -config cfg dti
   ```

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges” on page 15-11.

Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the `codegen` function to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the `LogIOForComparisonPlotting` option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

Prerequisites

To complete this example, you must install the following products:

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

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	myFilter.m	Entry-point MATLAB function
Test file	myFilterTest.m	MATLAB script that tests myFilter.m
Plotting function	plotDiff.m	Custom plot function
MAT-fiile	filterData.mat	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end
```

The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%             i) name
%             ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after
%               Fixed-Point conversion.
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % convert from cell to matrix
    floatVals = cell2mat(floatVals);
    fixedVals = cell2mat(fixedVals);

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexprep(varName,'_','\\_');
    escapedFcnName = regexprep(fcnName,'_','\\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;
            subplot(1, 2, 1);
            plotScatter(x_vec, y_vec, 100, floatTitle);

            % plot fixed point values
            y_vec = flatFixedVals;
            subplot(1, 2, 2);
            plotScatter(x_vec, y_vec, 100, fixedTitle);

        otherwise
            % Plot only output 'y' for this example, skip the rest
    end

end

function plotScatter(x_vec, y_vec, n, figTitle)
    % plot the last n samples
    x_plot = x_vec(end-n+1:end);
    y_plot = y_vec(end-n+1:end);

    hold on
    scatter(real(x_plot),imag(x_plot), 'bo');

    hold on
    scatter(real(y_plot),imag(y_plot), 'rx');

    title(figTitle);
end
```

Set Up Configuration Object

1 Create a `coder.FixptConfig` object.

```
fxptcfg = coder.config('fixpt');
```

2 Specify the test file name and custom plot function name. Enable logging and numerics testing.

```
fxptcfg.TestBenchName = 'myFilterTest';
fxptcfg.PlotFunction = 'plotDiff';
fxptcfg.TestNumerics = true;
fxptcfg. LogIOForComparisonPlotting = true;
fxptcfg.DefaultWordLength = 16;
```

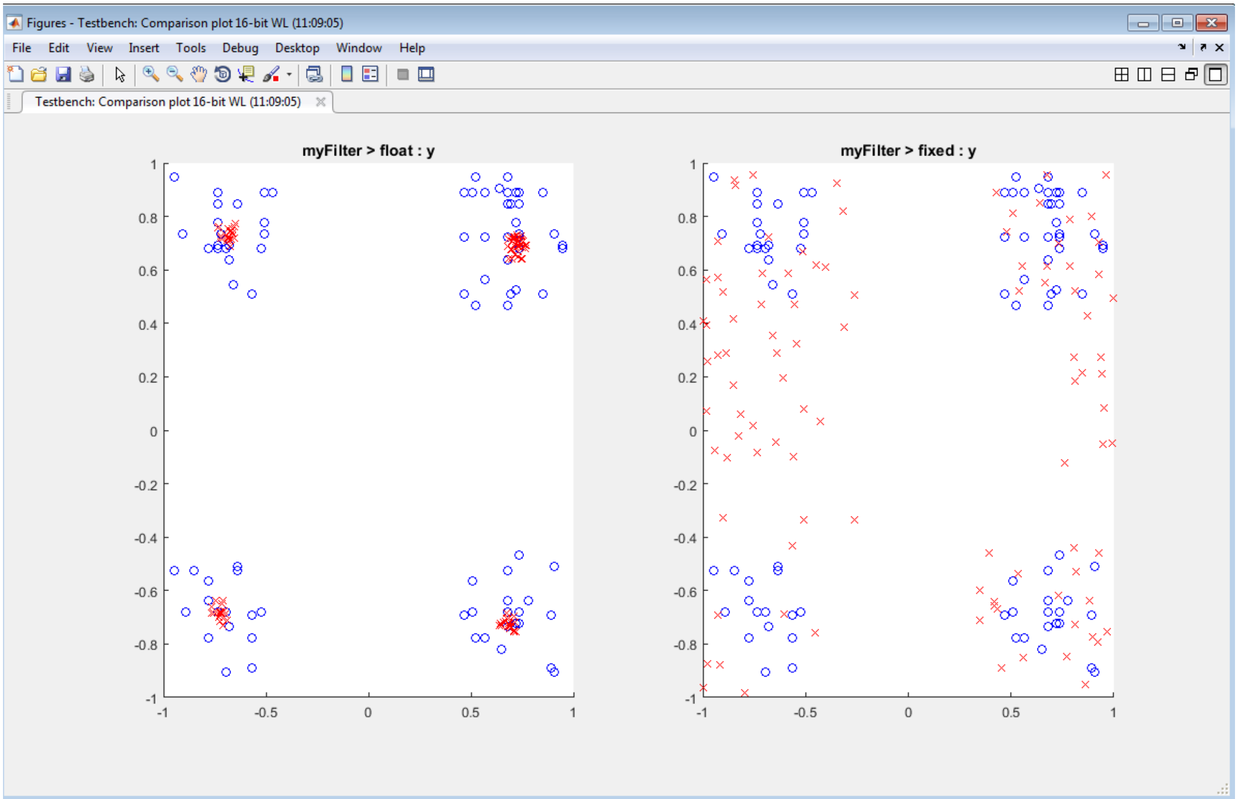
Convert to Fixed Point

Convert the floating-point MATLAB function, `myFilter`, to fixed-point MATLAB code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The conversion process generates fixed-point code using a default word length of `16` and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.

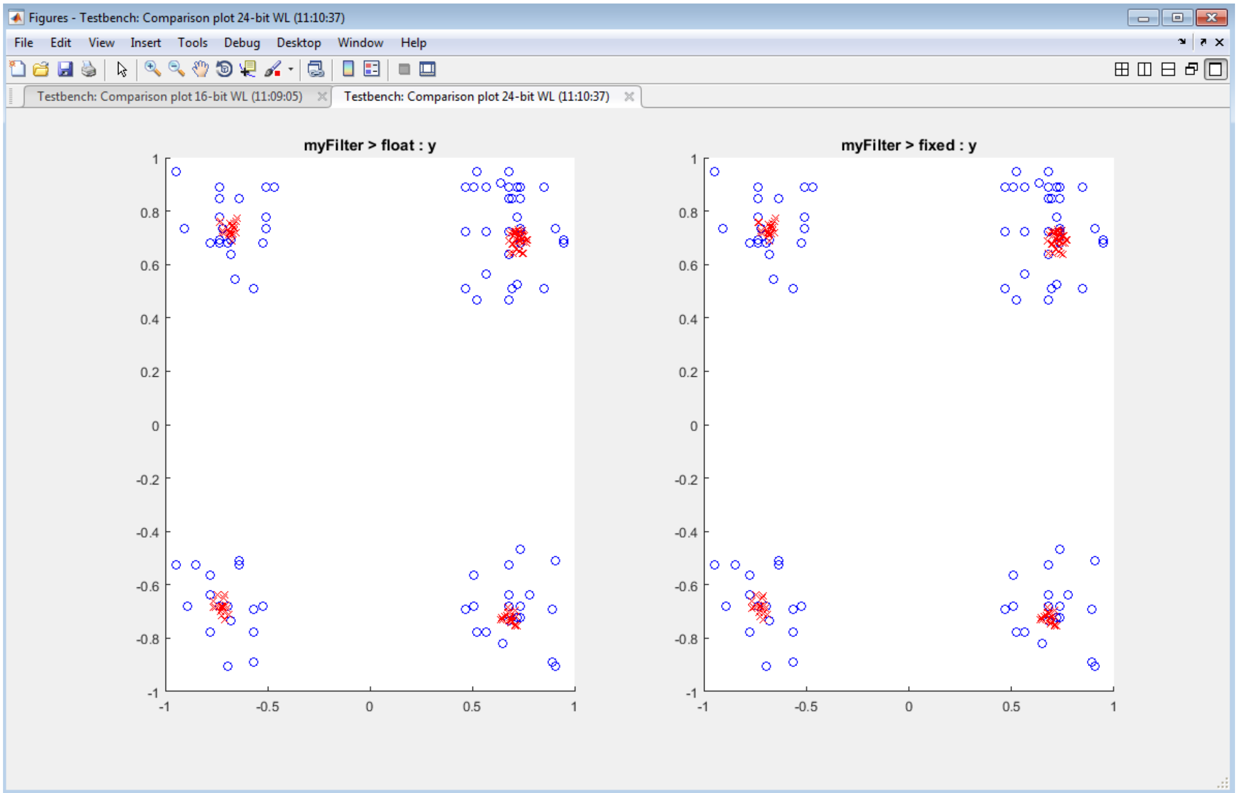


The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to `24` and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.



16

Single-Precision Conversion

Generate Single-Precision C Code at the Command Line

In this section...

This example shows how to generate single-precision C code from double-precision MATLAB code at the command line.

Prerequisites

To complete this example, install the following products:

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

  To change the default compiler, you can use `mex -setup`. See “Change Default Compiler” (MATLAB).

Create a Folder and Copy Relevant Files

1 Create a local working folder, for example, `c:\ex_2ndOrder_filter`.

2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

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

3 Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

Type	Name	Description
Function code	`ex_2ndOrder_filter.m`	Entry-point MATLAB function
Test file	`ex_2ndOrder_filter_test.m`	MATLAB script that tests `ex_2ndOrder_filter.m`

The ex_2ndOrder_filter Function

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i = 1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

The ex_2ndOrder_filter_test Script

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse. The script then plots the outputs.

```
% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256;                   % Number of points
t = linspace(0,1,N);       % Time vector from 0 to 1 second
f1 = N/2;                  % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N);        % Step
x_impulse = zeros(1,N);    % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
  y(i,:) = ex_2ndOrder_filter(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'}
clf
for i = 1:size(x,1)
  subplot(size(x,1),1,i)
  plot(t,x(i,:),t,y(i,:))
  title(titles{i})
  legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Determine the Type of the Input Argument

To determine the type of the input argument `x`, use `coder.getArgTypes` to run the test file `ex_2ndOrder_filter_test.m`

```
types = coder.getArgTypes('ex_2ndOrder_filter_test', 'ex_2ndOrder_filter');
```

The test file runs and displays the outputs of the filter for each of the input signals. `coder.getArgTypes` determines that the input type of `x` is 1x256 double.

Generate and Run Single-Precision MEX to Verify Numerical Behavior

1. Before you generate single-precision C code, generate a single-precision MEX function that you can use to verify the behavior of the generated single-precision code. To indicate that you want the single-precision MEX code, use the `-singleC` option.

   ```
   codegen -singleC ex_2ndOrder_filter -args types -report
   ```

   During MEX generation, the code generator detects single-precision conversion issues. Before you generate C/C++ code, fix these issues. This example does not have single-precision conversion issues.

   The generated MEX accepts single-precision and double-precision input. You can use the same test file to run the double-precision MATLAB function and the single-precision MEX function. You do not have to modify the test file to call the single-precision MEX function.
2. Run the test file `ex_2ndOrder_filter_test.m`. This file calls the double-precision MATLAB function `ex_2ndOrder_filter.m`.

   ```
   ex_2ndOrder_filter_test
   ```
3. The test file runs and displays the outputs of the filter for each of the input signals.
4. Run the test file `ex_2ndOrder_filter_test`, replacing calls to the double-precision `ex_2ndOrder_filter` function with calls to the single-precision `ex_2ndOrder_filter_mex` function.

   ```
   coder.runTest('ex_2ndOrder_filter_test', 'ex_2ndOrder_filter')
   ```
5. The test file runs and displays the outputs of the filter for each of the input signals. The single-precision MEX function produces the same results as the double-precision MATLAB function.

Generate Single-Precision C Code

1. Create a code configuration object for generation of a C static library, dynamic library, or executable. To avoid use of double-precision implementations of math functions, specify `'C99'` for the standard math library.

```
cfg = coder.config('lib');
cfg.TargetLangStandard = 'C99 (ISO)'
```

2 To generate single-precision C code, call `codegen` with the `-singleC` option. Enable generation of the code generation report.

```
codegen -config cfg -singleC ex_2ndOrder_filter -args {types{1}} -report
```

View the Generated Single-Precision C Code

To view the code generation report for the C code generation, click the **View Report** link.

The code generation report displays the generated code for `ex_2ndOrder_filter.c`.

- Double-precision variables have type `float` in the C code.
- The index `i` is an integer.

View Potential Data Type Issues

When you generate single-precision code, `codegen` enables highlighting of potential data type issues in the code generation report. If `codegen` cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation.

Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

See Also

`codegen` | `coder.config` | `coder.getArgTypes` | `coder.runTest`

Related Examples

- "Generate Single-Precision C Code Using the MATLAB Coder App" on page 16-8
- "Generate Single-Precision MATLAB Code" on page 16-14

More About

- "Single-Precision Conversion Best Practices" on page 16-24

- “Warnings from Conversion to Single-Precision C/C++ Code” on page 16-28

Generate Single-Precision C Code Using the MATLAB Coder App

In this section...

This example shows how to generate single-precision C code from double-precision MATLAB code by using the MATLAB Coder app.

Prerequisites

To complete this example, install the following products:

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

  To change the default compiler, you can use `mex -setup`. See "Change Default Compiler" (MATLAB).

Create a Folder and Copy Relevant Files

1 Create a local working folder, for example, `c:\ex_2ndOrder_filter`.

2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

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

3 Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

Type	Name	Description
Function code	`ex_2ndOrder_filter.m`	Entry-point MATLAB function
Test file	`ex_2ndOrder_filter_test.m`	MATLAB script that tests `ex_2ndOrder_filter.m`

The ex_2ndOrder_filter Function

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i = 1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

The ex_2ndOrder_filter_test Script

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse. The script then plots the outputs.

```
% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256;                   % Number of points
t = linspace(0,1,N);       % Time vector from 0 to 1 second
f1 = N/2;                  % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N);        % Step
x_impulse = zeros(1,N);    % Impulse
x_impulse(1) = 1;

% Run the function under test
x = [x_chirp;x_step;x_impulse];
y = zeros(size(x));
for i = 1:size(x,1)
  y(i,:) = ex_2ndOrder_filter(x(i,:));
end

% Plot the results
titles = {'Chirp','Step','Impulse'}
clf
for i = 1:size(x,1)
  subplot(size(x,1),1,i)
  plot(t,x(i,:),t,y(i,:))
  title(titles{i})
  legend('Input','Output')
end
xlabel('Time (s)')
figure(gcf)

disp('Test complete.')
```

Open the MATLAB Coder App

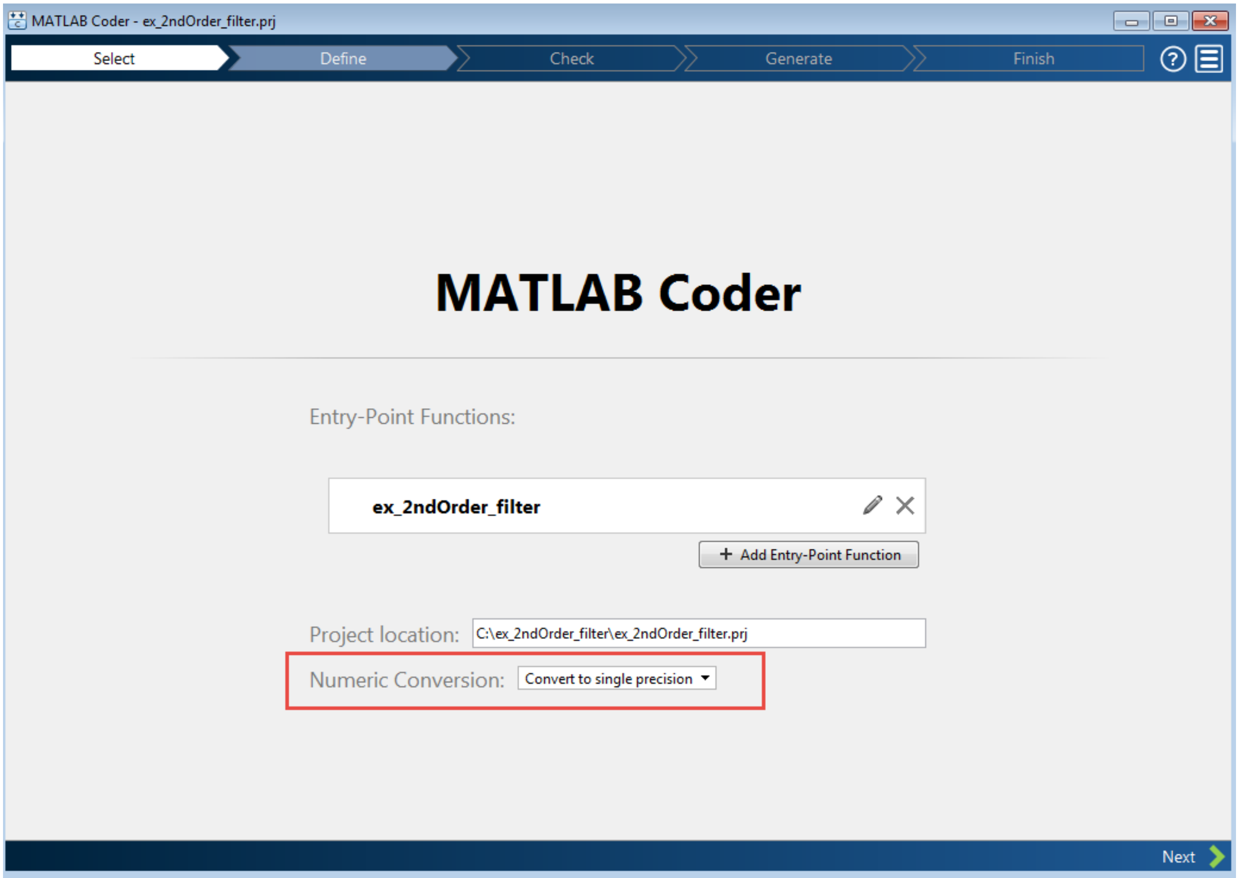
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

Select the Source Files

To add the entry-point function `ex_2ndOrder_filter` to the project, browse to the file `ex_2ndOrder_filter.m`, and then click **Open**. By default, the app saves information and settings for this project in the current folder in a file named `ex_2ndOrder_filter.prj`.

Enable Single-Precision Conversion

1 Set **Numeric Conversion** to `Convert to single precision`.



Project location: C:\ex_2ndOrder_filter\ex_2ndOrder_filter.prj

Numeric Conversion: Convert to single precision

2 Click **Next** to go to the **Define Input Types** step.

The app screens ex_2ndOrder_filter.m for code violations and code generation readiness issues. The app does not find issues in ex_2ndOrder_filter.m.

Define Input Types

1 On the **Define Input Types** page, to add ex_2ndOrder_filter_test as a test file, browse to ex_2ndOrder_filter_test. Click **Open**.

2 Click **Autodefine Input Types**.

   The test file runs and displays the outputs of the filter for each of the input signals. The app determines that the input type of x is double(1x256).

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

Check for Run-Time Issues

To detect and fix single-precision conversion issues, perform the **Check for Run-Time Issues** step.

1 On the **Check for Run-Time Issues** page, the app populates the test file field with ex_2ndOrder_filter_test, the test file that you used to define the input types.

2 Click **Check for Issues**.

   The app generates a single-precision MEX function from ex_2ndOrder_filter. It runs the test file ex_2ndOrder_filter_test replacing calls to ex_2ndOrder_filter with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. Click a message to highlight the problematic code in a window where you can edit the code. In this example, the app does not detect issues.

3 Click **Next** to go to the **Generate Code** page.

Generate Single-Precision C Code

1 In the **Generate** dialog box, set **Build type** to Static Library.

2 Set **Language** to **C**.

3 To avoid use of double-precision standard math functions, set the standard math library to C99 (ISO). Click **More Settings**. Then, click **Custom Code**. Set **Standard math library** to C99 (ISO).

4. For other settings, use the default values.
5. To generate the code, click **Generate**.

   MATLAB Coder builds the project and generates a C static library and supporting files in the default subfolder, `codegen/lib/ex_2ndOrder_filter`.

View the Generated C Code

The app displays the generated code for `ex_2ndOrder_filter.c`.

- Double-precision variables have type `float` in the C code.
- The index `i` is an integer.

View Potential Data Type Issues

When you generate single-precision code, the app enables highlighting of potential data type issues in the code generation report. If the app cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation.

To open the code generation report, click the `View Report` link. Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

Related Examples

- “Generate Single-Precision C Code at the Command Line” on page 16-2

More About

- “Single-Precision Conversion Best Practices” on page 16-24
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 16-28

Generate Single-Precision MATLAB Code

This example shows how to generate single-precision MATLAB code from double-precision MATLAB code. This example shows the single-precision conversion workflow that you use when you want to see single-precision MATLAB code or use verification options. Optionally, you can also generate single-precision C/C++ code.

Prerequisites

To complete this example, install the following products:

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

  To change the default compiler, you can use `mex -setup`. See “Change Default Compiler” (MATLAB).

Create a Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\ex_2ndOrder_filter`.
2. Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `ex_2ndOrder_filter.m` and `ex_2ndOrder_filter_test.m` files to your local working folder.

Type	Name	Description
Function code	`ex_2ndOrder_filter.m`	Entry-point MATLAB function
Test file	`ex_2ndOrder_filter_test.m`	MATLAB script that tests `ex_2ndOrder_filter.m`

The ex_2ndOrder_filter Function

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i = 1:length(x)
      y(i) = b(1)*x(i) + z(1);
      z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2) = b(3)*x(i)        - a(3) * y(i);
  end
end
```

The ex_2ndOrder_filter_test Script

It is a best practice to create a separate test script for preprocessing and postprocessing such as:

- Setting up input values.
- Calling the function under test.
- Outputting the test results.

To cover the full intended operating range of the system, the test script runs the `ex_2ndOrder_filter` function with three input signals: chirp, step, and impulse. The script then plots the outputs.

```
% ex_2ndOrder_filter_test
%
% Define representative inputs
N = 256;                   % Number of points
t = linspace(0,1,N);       % Time vector from 0 to 1 second
f1 = N/2;                  % Target frequency of chirp set to Nyquist
x_chirp = sin(pi*f1*t.^2); % Linear chirp from 0 to Fs/2 Hz in 1 second
x_step = ones(1,N);        % Step
x_impulse = zeros(1,N);    % Impulse
```

```
    x_impulse(1) = 1;

    % Run the function under test
    x = [x_chirp;x_step;x_impulse];
    y = zeros(size(x));
    for i = 1:size(x,1)
      y(i,:) = ex_2ndOrder_filter(x(i,:));
    end

    % Plot the results
    titles = {'Chirp','Step','Impulse'}
    clf
    for i = 1:size(x,1)
      subplot(size(x,1),1,i)
      plot(t,x(i,:),t,y(i,:))
      title(titles{i})
      legend('Input','Output')
    end
    xlabel('Time (s)')
    figure(gcf)

    disp('Test complete.')
```

Set Up the Single-Precision Configuration Object

Create a single-precision configuration object. Specify the test file name. Verify the single-precision code using the test file. Plot the error between the double-precision code and single-precision code. Use the default values for the other properties.

```
scfg = coder.config('single');
scfg.TestBenchName = 'ex_2ndOrder_filter_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

Generate Single-Precision MATLAB Code

To convert the double-precision MATLAB function, `ex_2ndOrder_filter`, to single-precision MATLAB code, use the `codegen` function with the `-double2single` option.

```
codegen -double2single scfg ex_2ndOrder_filter
```

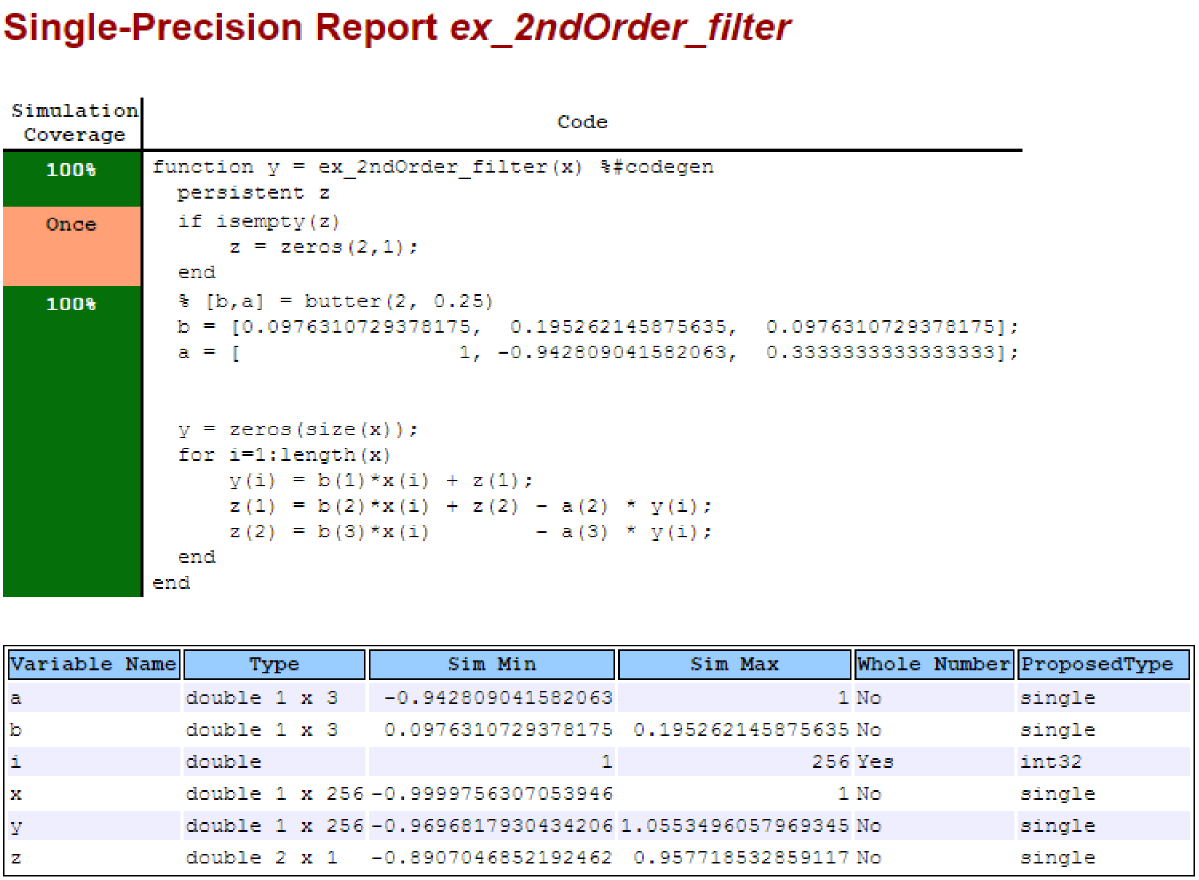
`codegen` analyzes the double-precision code. The conversion process infers types by running the test file because you did not specify the input types for the ex_2ndOrder_filter function. The conversion process selects single-precision types for the double-precision variables. It selects int32 for index variables. When the conversion is complete, codegen generates a type proposal report.

View the Type Proposal Report

To see the types that the conversion process selected for the variables, open the type proposal report for the ex_2ndOrder_filter function. Click the link ex_2ndOrder_filter_report.html.

The report opens in a web browser. The conversion process converted:

- Double-precision variables to single.
- The index i to int32. The conversion process casts index and dimension variables to int32.



Single-Precision Report *ex_2ndOrder_filter*

Simulation Coverage: 100% · Once · 100%

Code

```
function y = ex_2ndOrder_filter(x) %#codegen
  persistent z
  if isempty(z)
      z = zeros(2,1);
  end
  % [b,a] = butter(2, 0.25)
  b = [0.0976310729378175,  0.195262145875635,  0.0976310729378175];
  a = [                 1, -0.942809041582063,  0.3333333333333333];


  y = zeros(size(x));
  for i=1:length(x)
      y(i)  = b(1)*x(i) + z(1);
      z(1)  = b(2)*x(i) + z(2) - a(2) * y(i);
      z(2)  = b(3)*x(i)        - a(3) * y(i);
  end
end
```

Variable Name	Type	Sim Min	Sim Max	Whole Number	ProposedType
a	double 1 x 3	-0.942809041582063	1	No	single
b	double 1 x 3	0.0976310729378175	0.195262145875635	No	single
i	double	1	256	Yes	int32
x	double 1 x 256	-0.9999756307053946	1	No	single
y	double 1 x 256	-0.9696817930434206	1.0553496057969345	No	single
z	double 2 x 1	-0.8907046852192462	0.957718532859117	No	single

View Generated Single-Precision MATLAB Code

To view the report for the generation of the single-precision MATLAB code, in the Command Window:

1 Scroll to the `Generate Single-Precision Code` step. Click the **View report** link.

2 On the **MATLAB code** tab, under **Functions**, click `ex_2ndOrder_filter_single`.

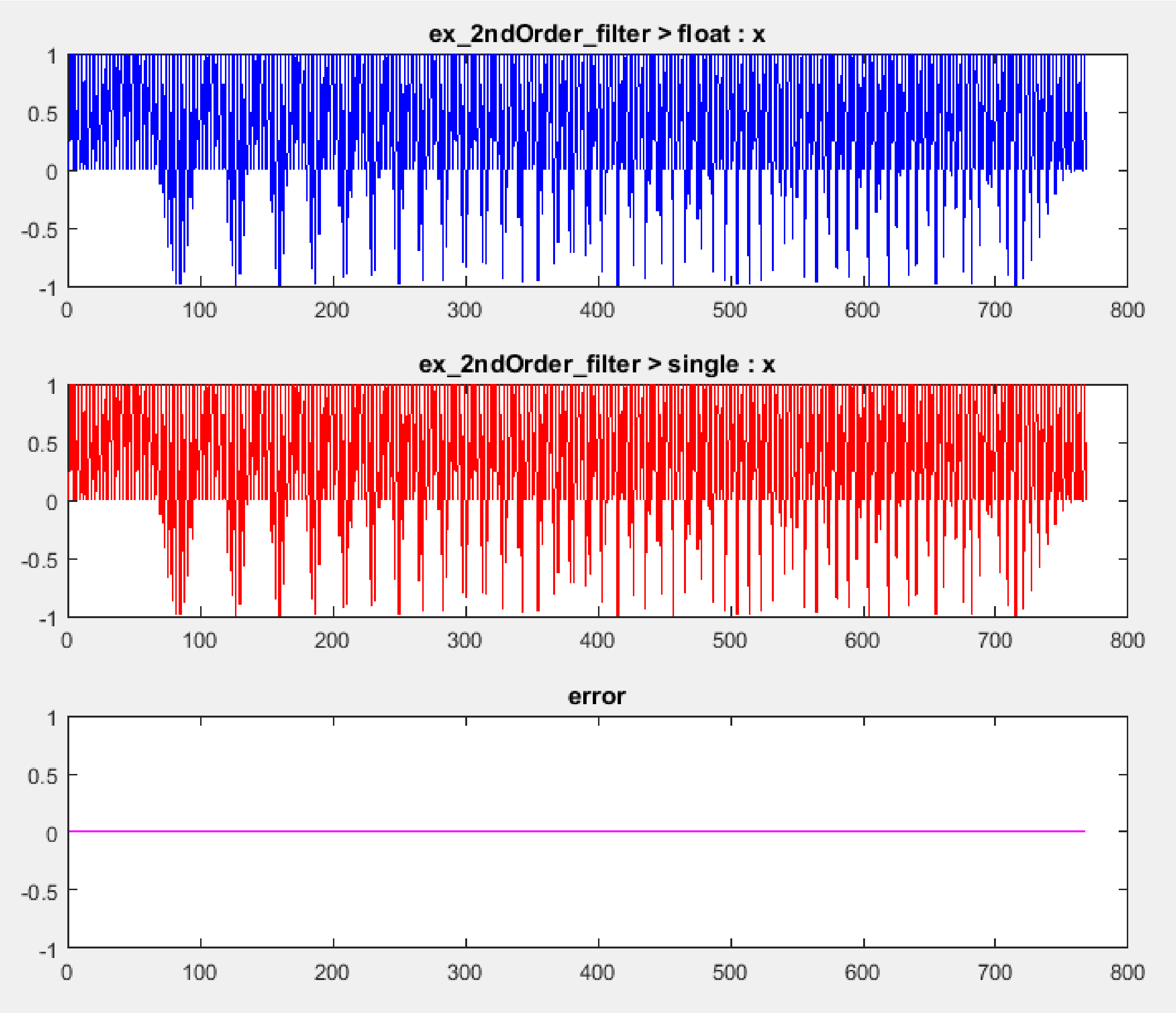
The code generation report displays the single-precision MATLAB code for `ex_2ndOrder_filter`.

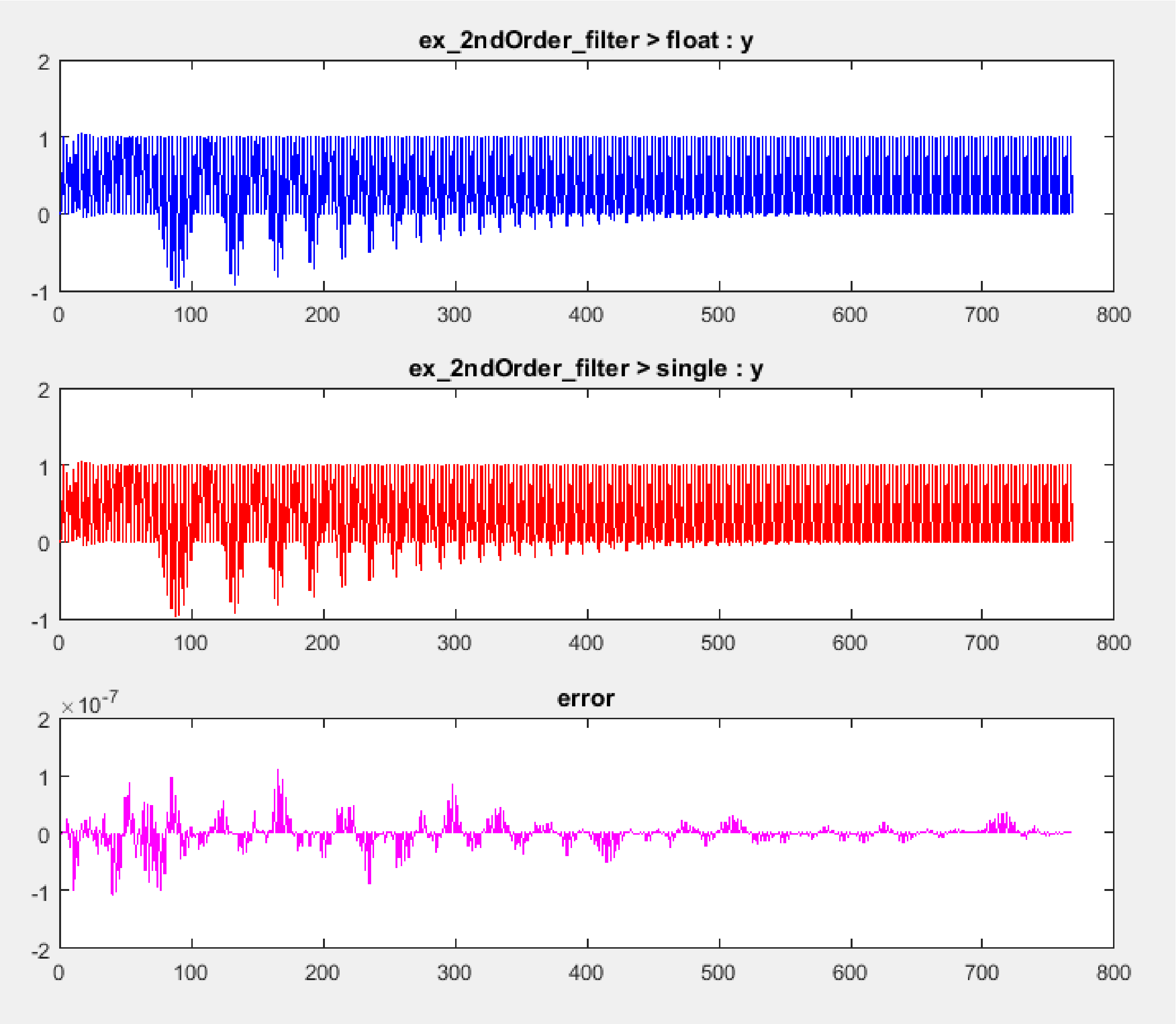
View Potential Data Type Issues

When you generate single-precision code, `codegen` enables highlighting of potential data type issues in code generation reports. If `codegen` cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation. Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

Compare the Double-Precision and Single-Precision Variables

You can see the comparison plots for the input `x` and output `y` because you selected to log inputs and outputs for comparison plots .


ex_2ndOrder_filter > float : x
1
0.5
0
-0.5
-1
0
100
200
300
400
500
600
700
800
ex_2ndOrder_filter > single : x
1
0.5
0
-0.5
-1
0
100
200
300
400
500
600
700
800
error
1
0.5
0
-0.5
-1
0
100
200
300
400
500
600
700
800



Optionally Generate Single-Precision C Code

If you also want to generate single-precision C code, create a code configuration object for C code generation. Use this configuration object with the `-config` option of the `codegen` function. For example:

1. Create a code configuration object for generation of a C static library. Specify 'C99' for the standard math library.

   ```
   cfg = coder.config('lib');
   cfg.TargetLangStandard = 'C99 (ISO)';
   ```
2. Generate the C code. Enable generation of the code generation report.

   ```
   codegen -double2single scfg -config cfg ex_2ndOrder_filter -report
   ```
3. To view the code generation report for the C code generation, click the **View Report** link.

   The code generation report displays the generated code for ex_2ndOrder_filter.c.

   - Double-precision variables have type float in the C code.
   - The index i is an integer.

   When you generate single-precision code, codegen enables highlighting of potential data type issues in the code generation report. If codegen cannot remove a double-precision operation, the report highlights the MATLAB expression that results in the operation.

   Click the **MATLAB code** tab. Under **Highlight**, the report shows that no double-precision operations remain.

See Also

coder.SingleConfig | codegen | coder.config

Related Examples

- "Generate Single-Precision C Code Using the MATLAB Coder App" on page 16-8
- "Generate Single-Precision C Code at the Command Line" on page 16-2

More About

- "Single-Precision Conversion Best Practices" on page 16-24
- "Warnings from Conversion to Single-Precision C/C++ Code" on page 16-28

Choose a Single-Precision Conversion Workflow

The information in the following table helps you to decide which single-precision workflow to use.

Goal	Use
You want to generate single-precision C/C++ code in the most direct way using the `codegen` function.	`codegen` with `-singleC` option. See “Generate Single-Precision C Code at the Command Line” on page 16-2.
You want to generate single-precision C/C++ code in the most direct way using the MATLAB Coder app.	The MATLAB Coder app with **Numeric Conversion** set to `Convert to single precision`. See “Generate Single-Precision C Code Using the MATLAB Coder App” on page 16-8.
You want to generate only single-precision MATLAB code. You want to see the single-precision MATLAB code or use verification options.	`codegen` with the `-double2single` option and a `coder.SingleConfig` object. See “Generate Single-Precision MATLAB Code” on page 16-14.
You want to generate single-precision MATLAB code, and then generate single-precision C/C++ code from the single-precision MATLAB code.	`codegen` with the `-double2single` option and a `coder.SingleConfig` object. Also, use the `-config` object with a code configuration object for the output type that you want. See “Generate Single-Precision MATLAB Code” on page 16-14.

Single-Precision Conversion Best Practices

In this section...

Use Integers for Index Variables

In MATLAB code that you want to convert to single precision, it is a best practice to use integers for index variables. However, if the code does not use integers for index variables, when possible single-precision conversion using `codegen` with `-double2single` tries to detect the index variables and select `int32` types for them.

Limit Use of `assert` Statements

- Do not use `assert` statements to define the properties of input arguments.
- Do not use `assert` statements to test the type of a variable. For example, do not use

```
assert(isa(a, 'double'))
```

Initialize MATLAB Class Properties in Constructor

Do not initialize MATLAB class properties in the `properties` block. Instead, use the constructor to initialize the class properties.

Provide a Test File That Calls Your MATLAB Function

Separate your core algorithm from other code that you use to test and verify the results. Create a test file that calls your double-precision MATLAB algorithm. You can use the test file to:

- Automatically define properties of the top-level function inputs.
- Verify that the double-precision algorithm behaves as you expect. The double-precision behavior is the baseline against which you compare the behavior of the single-precision versions of your algorithm.
- Compare the behavior of the single-precision version of your algorithm to the double-precision baseline.

For best results, the test file must exercise the algorithm over its full operating range.

Prepare Your Code for Code Generation

MATLAB code that you want to convert to single precision must comply with code generation requirements. See “MATLAB Programming for Code Generation”.

To help you identify unsupported functions or constructs in your MATLAB code, add the `%#codegen` pragma to the top of your MATLAB file. When you edit your code in the MATLAB editor, the MATLAB Code Analyzer flags functions and constructs that are not supported for code generation. See “Check Code with the Code Analyzer” on page 18-6. When you use the MATLAB Coder app, the app screens your code for code generation readiness. At the function line, you can use the Code Generation Readiness Tool. See “Check Code by Using the Code Generation Readiness Tool” on page 18-8.

Verify Double-Precision Code Before Single-Precision Conversion

Before you begin the single-precision conversion process, verify that you can successfully generate code from your double-precision MATLAB code. Generate and run a MEX version of your double-precision MATLAB code so that you can:

- Detect and fix compilation issues.
- Verify that the generated single-precision code behaves the same as the double-precision MATLAB code.

See “Why Test MEX Functions in MATLAB?” on page 19-2.

Best Practices for Generation of Single-Precision C/C++ Code

When you generate single-precision C/C++ code by using the MATLAB Coder app or `codegen` with the `-singleC` option, follow these best practices:

Set the Standard Math Library to C99

When you generate C/C++ libraries or executables, by default, the code generator uses the C89 /C90 (ANSI) standard math library. When you generate single-precision C/C++ code using this library, the code generator warns you if a function in this library uses double precision. To avoid this warning, set the standard math library to C99 (ISO). See "Warnings from Conversion to Single-Precision C/C++ Code" on page 16-28.

Cast Large Double Constant to Integer

For a constant greater than `2^24`, in your original double-precision MATLAB function, cast the constant to an integer type that is large enough for the constant value. For example:

```
a = int32(2^24 + 1);
```

Generate and Run Single-Precision MEX Before Generating Single-Precision C/C++ Code

Before you generate single-precision C code, generate and run a single-precision MEX version of your MATLAB code. When you follow this practice, you can detect and fix compiler issues. You can verify that the single-precision MEX function has the same functionality as the MATLAB code.

If you use `codegen` with `-singleC`:

1. Generate the single-precision MEX.
2. Call `coder.runTest` to run your test file, replacing calls to the double-precision MATLAB code with calls to the single-precision MEX code.

If you use the MATLAB Coder app, perform the **Check for Run-Time Issues** step with single-precision conversion enabled.

Best Practices for Generation of Single-Precision MATLAB Code

When you use `codegen` with the `-double2single` option to generate single-precision MATLAB code, follow these best practices:

Use the `-args` Option to Specify Input Properties

When you generate single-precision MATLAB code, if you specify a test file, you do not have to specify argument properties with the `-args` option. In this case, the code generator runs the test file to determine the properties of the input types. However, running the test file can slow the code generation. It is a best practice to determine the input properties one time with `coder.getArgTypes`. Then, pass the properties to the `-args` option. For example:

```
types = coder.getArgTypes('myfun_test', 'myfun');
scfg = coder.config('single');
codegen -double2single scfg -args types myfun -report
```

When you repeat the code generation in the same MATLAB session, this practice saves you time.

Test Numerics and Log I/O Data

When you use the `codegen` function with the `-double2single` option to generate single-precision MATLAB code, enable numerics testing and I/O data logging for comparison plots. To use numerics testing, you must provide a test file that calls your MATLAB function. To enable numerics testing and I/O data logging, create a `coder.SingleConfig` object. Set the `TestBenchName`, `TestNumerics`, and `LogIOForComparisonPlotting` properties. For example:

```
scfg = coder.config('single');
scfg.TestBenchName = 'mytest';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

More About

- “Warnings from Conversion to Single-Precision C/C++ Code” on page 16-28

Warnings from Conversion to Single-Precision C/C++ Code

When you generate single-precision C/C++ code by using the MATLAB Coder app or `codegen` with the `-singleC` option, you can receive the following warnings.

Function Uses Double-Precision in the C89/C90 Standard

If the standard math library is C89/C90, the conversion process warns you when a function uses double-precision code in the C89/C90 standard.

Consider the function `mysine`.

```
function c = mysine(a)
c = sin(a);
end
```

Generate single-precision code for `mysine`.

```
x = -pi:0.01:pi;
codegen -singleC mysine -args {x} -config:lib -report
```

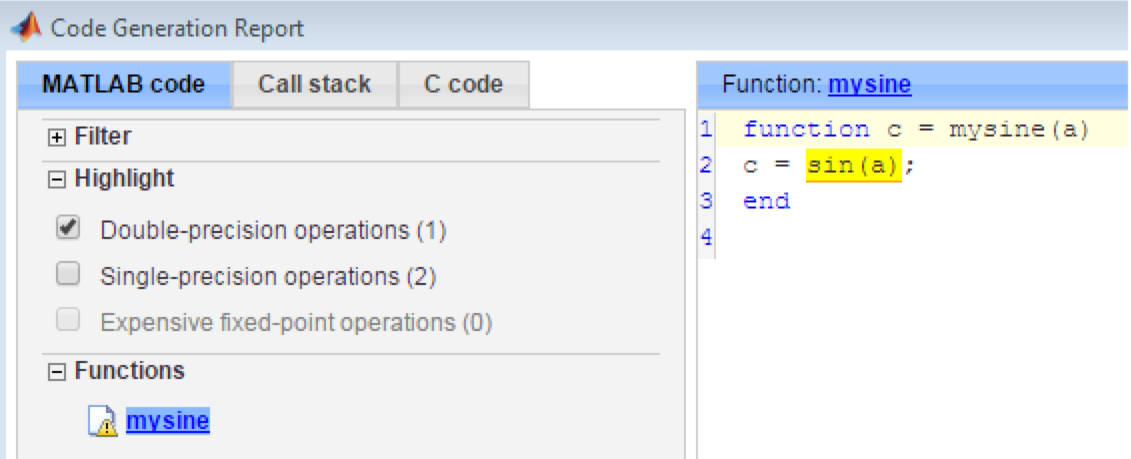
`codegen` warns that `sin` uses double-precision in the C89/C90 (ANSI) standard.

```
Warning: The function sin uses double-precision in the C89/C90 (ANSI) standard. For single-precision
code, consider using the C99 (ISO) standard or use your own function.
```

To open the code generation report, click the **View Report** link.

To see that double-precision operations remain in the converted code, click the **MATLAB code** tab. Under **Highlight**, select the **Double-precision operations** check box. Under **Functions**, click `mysine`.

In the code pane, the report highlights the `sin` call.



☑ Double-precision operations (1)

□ Single-precision operations (2)

□ Expensive fixed-point operations (0)

Function: mysine

```
function c = mysine(a)
c = sin(a);
end
```

To address this warning, specify use of the C99 (ISO) standard math library.

- At the command line:

  ```
  cfg = coder.config('lib');
  cfg.TargetLangStandard = 'C99 (ISO)'
  ```
- In the app, in the project build settings, on the **Custom Code** tab, set **Standard math library** to `C99 (ISO)`.

Built-In Function Is Implemented in Double-Precision

Some built-in MATLAB functions are implemented using double-precision operations. The conversion process warns that the code generated for these functions contains double-precision operations.

Consider the function `geterf` that calls the built-in function `erf`.

```
function y = geterf(x)
y = erf(x);
end
```

Generate single-precision code for`geterf`.

```
codegen -singleC -config:lib -args {1} geterf -report
```

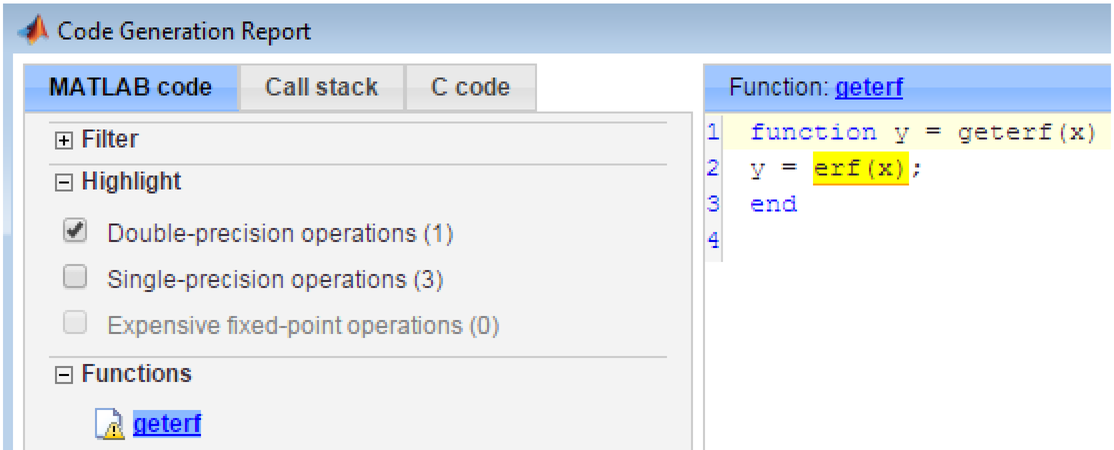
`codegen` warns that `erf` is implemented in double precision.

```
Warning: The builtin function erf is implemented in double-precision. Code generated for this
function will contain doubles.
```

To open the code generation report, click the **View Report** link.

To see that double-precision operations remain in the converted code, click the **MATLAB code** tab. Under **Highlight**, select the **Double-precision operations** check box. Under **Functions**, click `geterf`.

In the code pane, the report highlights the `erf` call.



☑ Double-precision operations (1)

□ Single-precision operations (3)

□ Expensive fixed-point operations (0)

Function: geterf

```
function y = geterf(x)
y = erf(x);
end
```

To address this warning, rewrite your code so that it does not use the function that is implemented in double precision.

Built-In Function Returns Double-Precision

If a built-in MATLAB function returns a double-precision output, the conversion process generates a warning.

Consider the function `mysum` that calls the built-in function `sum`.

```
function y = mysum(x)
y = sum(int32(x));
end
```

Generate single-precision code for`mysum`.

```
A = 1:10;
codegen -singleC -config:lib -args {A} mysum -report
```

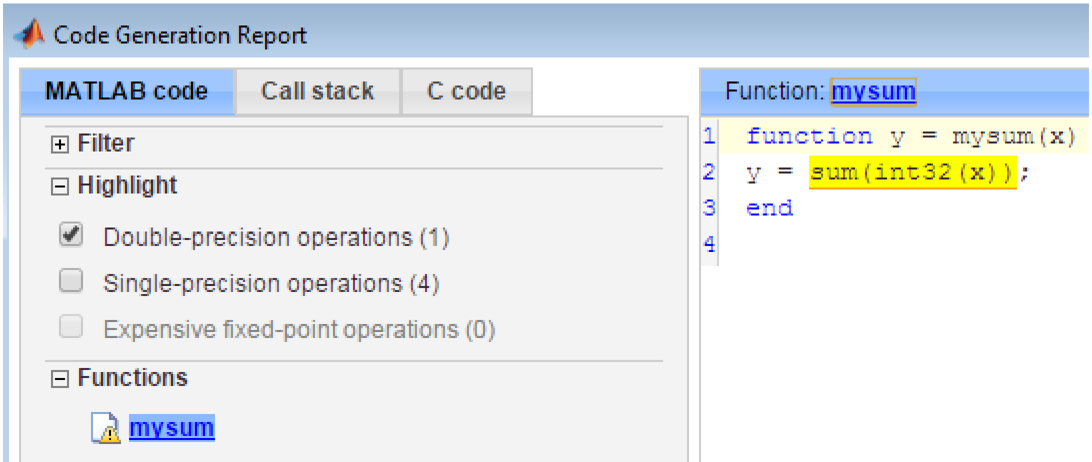
`codegen` warns that `mysum` is implemented in double precision.

```
Warning: The output of builtin function sum is double-precision and has been cast to
single-precision. The code generated for the builtin function may still contain doubles.
```

To open the code generation report, click the **View Report** link.

To see that double-precision operations remain in the converted code, click the **MATLAB code** tab. Under **Highlight**, select the **Double-precision operations** check box. Under **Functions**, click `mysum`.

In the code pane, the report highlights the `sum` call.



☑ Double-precision operations (1)

□ Single-precision operations (4)

□ Expensive fixed-point operations (0)

```
function y = mysum(x)
y = sum(int32(x));
end
```

To address this warning, specify that you want the function to return the `'native'` class.

```
(sum(int32(1), 'native')
```

Using this option causes the function to return the same type as the input.

More About

- "Single-Precision Conversion Best Practices" on page 16-24

Combining Integers and Double-Precision Numbers

MATLAB supports the combination of integers of the same class and scalar double-precision numbers. MATLAB does not support the combination of integers and single-precision numbers. If you use the MATLAB Coder app or `codegen` with the `-singleC` option to generate single-precision C/C++ code, your MATLAB code cannot combine integers and double-precision numbers. Converting an expression that combines integers and doubles results in an illegal MATLAB expression. To work around this limitation, cast the numbers so that the types of the numbers match. Either cast the integer numbers to double-precision or cast the double-precision numbers to the integer class.

For example, consider the function `dut` that returns the sum of `a` and `b`.

```
function c = dut(a,b)
c = a + b;
end
```

Generate single-precision code using `codegen` with the `-singleC` option. Specify that the first argument is double and the second argument is int32.

```
 codegen -singleC -config:lib dut -args {0, int32(2)} -report
```

Code generation fails. The message suggests that you cast the operands so that they have the same types.

Rewrite the code so that it cast `a` to the type of `b`.

```
function c = dut(a,b)
c = int32(a) + b;
end
```

MATLAB Language Features Supported for Single-Precision Conversion

MATLAB Language Features Supported for Single-Precision Conversion

Single-precision conversion supports the following MATLAB language features:

- N-dimensional arrays.
- Matrix operations, including deletion of rows and columns.
- Variable-size data (see "Generate Code for Variable-Size Data" on page 20-108). Comparison plotting does not support variable-size data.
- Subscripting (see "Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation" on page 6-30).
- Complex numbers (see "Code Generation for Complex Data" on page 5-4).
- Numeric classes (see "Supported Variable Types" on page 4-16).
- Program control statements `if`, `switch`, `for`, `while`, and `break`.
- Arithmetic, relational, and logical operators.
- Local functions.
- Global variables.
- Persistent variables.
- Structures.
- Characters.

  Single-precision conversion does not support the complete set of Unicode characters. Characters are restricted to 8 bits of precision in generated code. Many mathematical operations require more than 8 bits of precision. If you intend to convert your MATLAB algorithm to single precision, it is a best practice not to perform arithmetic with characters.

- MATLAB classes. Single-precision conversion supports:
  - Class properties
  - Constructors
  - Methods
  - Specializations

  It does not support class inheritance or packages.
- Function calls (see “Resolution of Function Calls for Code Generation” on page 13-2)
- `varargin` and `varargout` are supported when you use `codegen` with `-singleC`. They are not supported when you use `codegen` with `-double2single`.

MATLAB Language Features Not Supported for Single-Precision Conversion

Single-precision conversion does not support the following features:

- Anonymous functions
- Cell arrays
- Function handles
- Java
- Nested functions
- Recursion
- Sparse matrices
- `try/catch` statements
- `varargin` and `varargout` are not supported when you use `codegen` with `-double2single`. They are supported when you use `codegen` with `-singleC`

Setting Up a MATLAB Coder Project

Set Up a MATLAB Coder Project

1 To open the app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.
2 Create a project or open an existing project. See “Create a Project” on page 17-2 and “Open an Existing Project” on page 17-2.
3 If the app detects code generation readiness issues in your entry-point functions, address these issues.
4 Define the properties of the entry-point function input types. See “Specify Properties of Entry-Point Function Inputs Using the App” on page 17-3.
5 Check for run-time issues. Provide code or a test file that the app can use to test your code. The app generates a MEX function. It runs your test code or test file, replacing calls to your MATLAB function with calls to the MEX function. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.
6 Configure the build settings. Select the build type, language, and production hardware. Optionally, modify other build settings. See “Configure Build Settings” on page 20-26.

   You can now generate code.

Create a Project

On the **Select Source Files** page, specify the MATLAB files from which you want to generate code. An entry-point function is a function that you call from MATLAB. Do not add files that have spaces in their names.

The app creates a project that has the name of the first entry-point function.

Open an Existing Project

1 On the app toolbar, click ☰ and select **Open existing project**.
2 Type or select the project.

The app closes other open projects.

If the project is a Fixed-Point Converter project, and you have a Fixed-Point Designer license, the project opens in the Fixed-Point Converter app.

Specify Properties of Entry-Point Function Inputs Using the App

Why Specify Input Properties?

Because C and C++ are statically typed languages, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. To infer variable properties in MATLAB files, MATLAB Coder must identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs to MATLAB Coder. If your primary function has no input parameters, you do not need to specify properties of inputs to local functions or external functions called by the primary function.

Unless you use the tilde (~) character to specify unused function inputs, you must specify the same number and order of inputs as the MATLAB function . If you use the tilde character, the inputs default to real, scalar doubles.

See Also

- “Properties to Specify” on page 20-46

Specify an Input Definition Using the App

Specify an input definition using one of the following methods:

- Autodefine Input Types
- Define Type
- Define by Example
- Define Constant
- “Define Inputs Programmatically in the MATLAB File” on page 17-28

Automatically Define Input Types by Using the App

If you specify a test file that calls the project entry-point functions, the MATLAB Coder app can infer the input argument types by running the test file. If a test file calls an entry-point function multiple times with different size inputs, the app takes the union of the inputs. The app infers that the inputs are variable size, with an upper bound equal to the size of the largest input.

Before using the app to automatically define function input argument types, you must add at least one entry-point file to your project. You must also specify code that calls your entry-point functions with the expected input types. It is a best practice to provide a test file that calls your entry-point functions. The test file can be either a MATLAB function or a script. The test file must call the entry-point function at least once.

To automatically define input types:

1 On the **Define Input Types** page, specify a test file. Alternatively, you can enter code directly.
2 Click **Autodefine Input Types**.

   The app runs the test file and infers the types for entry-point input arguments. The app displays the inferred types.

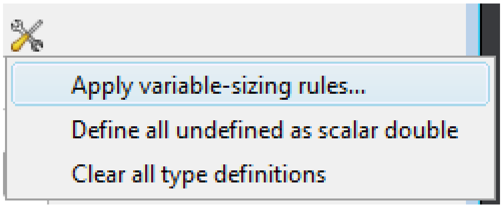
Note: If you automatically define the input types, the entry-point functions must be in a writable folder.

If your test file does not call an entry-point function with different size inputs, the resulting type dimensions are fixed-size. After you define the input types, you can specify and apply rules for making type dimensions variable-size when they meet a size threshold. See “Make Dimensions Variable-Size When They Meet Size Threshold” on page 17-5.

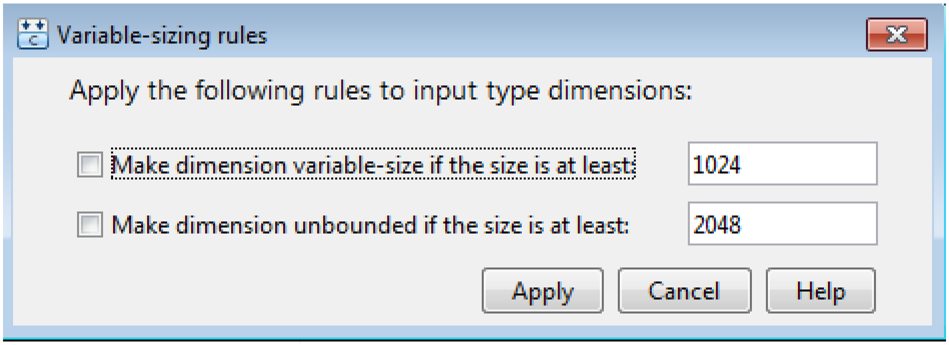
Make Dimensions Variable-Size When They Meet Size Threshold

After you define input types automatically or manually, you can make type dimensions variable-size when they meet a size threshold.

1 From the tools menu, select `Apply variable-sizing rules`.


Apply variable-sizing rules...
Define all undefined as scalar double
Clear all type definitions

2 In the **Variable-sizing rules** dialog box, select the rules that you want to apply.


Variable-sizing rules

Apply the following rules to input type dimensions:

□ Make dimension variable-size if the size is at least: 1024

□ Make dimension unbounded if the size is at least: 2048

- To make a dimension variable-size with an upper bound, select the **Make dimension variable-size if the size is at least** check box. Specify the threshold. If the size of a dimension of an input type is equal to or greater than this threshold, the app makes the dimension variable-size. The upper bound is the original size of the dimension.
- To make a dimension variable-size with no upper bound, select the **Make dimension unbounded if the size is at least** check box. Specify the threshold. If the size of a dimension of an input is equal to or greater than this threshold, the app makes this dimension unbounded.

3 To apply the rules to the current type definitions, click **Apply**. If you change type definitions, the rules do not affect the new definitions unless you apply them.

More About

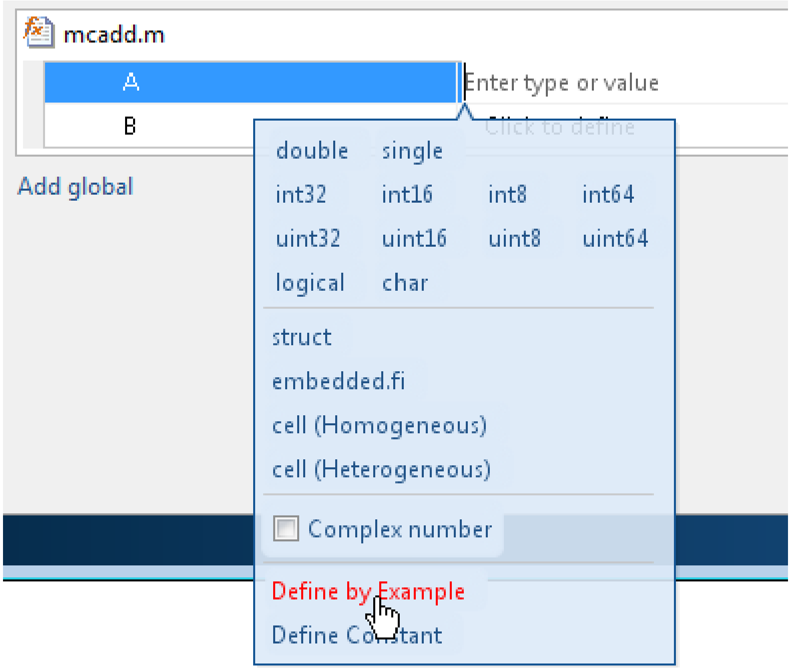
- "Specify Properties of Entry-Point Function Inputs" on page 20-46
- "Code Generation for Variable-Size Arrays" on page 6-2

Define Input Parameter by Example by Using the App

In this section...

Define an Input Parameter by Example

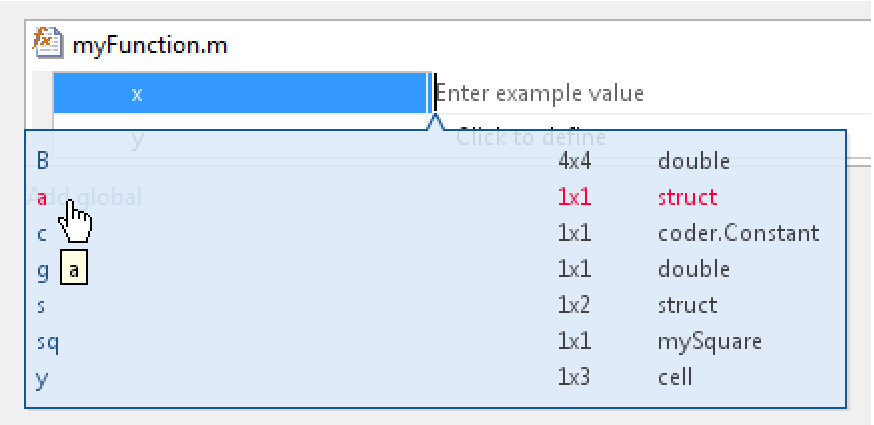
1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.



3 Select **Define by Example**.

4 In the field to the right of the parameter, enter a MATLAB expression. The variable has the class, size, and complexity of the value of the expression.

Alternatively, you can select a variable from the list of workspace variables that displays.



B	4x4	double
a	1x1	struct
c	1x1	coder.Constant
g	1x1	double
s	1x2	struct
sq	1x1	mySquare
y	1x3	cell

Specify Input Parameters by Example

This example shows how to specify a `1-by-4` vector of unsigned 16-bit integers.

1 On the **Define Input Types** page, click **Let me enter input or global types directly**.

2 Click the field to the right of the input parameter that you want to define.

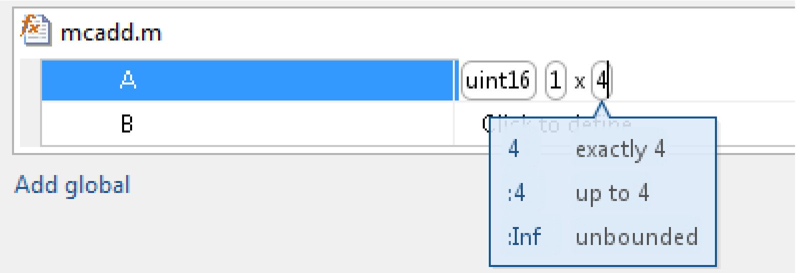
3 Select **Define by Example**.

4 In the field to the right of the parameter, enter:

```
zeros(1,4,'uint16')
```

The input type is `uint16(1x4)`.

5 Optionally, after you specify the input type, you can specify that the input is variable size. For example, select the second dimension.



6 To specify that the second dimension is variable size with an upper bound of `4`, select `:4`. Alternatively, to specify that the second dimension is unbounded, select `:Inf`.

Alternatively, you can specify that the input is variable size by using the `coder.newtype` function. Enter the MATLAB expression:

```
coder.newtype('uint16',[1 4],[0 1])
```

Note: To specify that an input is a double-precision scalar, enter `0`.

Specify a Structure Type Input Parameter by Example

This example shows how to specify a structure with two fields `a` and `b`. The input type of `a` is scalar double. The input type of `b` is scalar char.

1 On the **Define Input Types** page, click **Let me enter input or global types directly**.

2 Click the field to the right of the input parameter that you want to define.

3 Select **Define by Example**.

4 In the field to the right of the parameter, enter:

```
struct('a', 1, 'b', 'x')
```

The type of the input parameter is `struct(1x1)`. The type of field `a` is `double(1x1)`. The type of field `b` is `char(1x1)`

5 For an array of structures, to specify the size of each dimension, click the dimension and specify the size. For example, enter 4 for the first dimension.

6 To specify that the second dimension is variable size with an upper bound of `4`, select `:4`. Alternatively, to specify that the second dimension is unbounded select `:Inf`.

Alternatively, specify the size of the array of structures in the `struct` function call. For example, `struct('a', { 1 2}, 'b', {'x', 'y'})` specifies a 1x2 array of structures with fields `a` and `b`. The type of field `a` is `double(1x1)`. The type of field `b` is `char(1x1)`.

To modify the type definition, see “Specify a Structure Input Parameter” on page 17-19.

Specify a Cell Array Type Input Parameter by Example

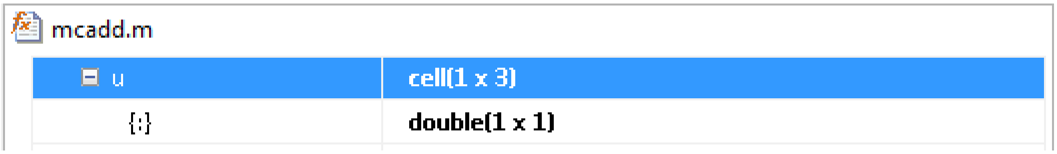
This example shows how to specify a cell array input by example. When you define a cell array by example, the app determines whether the cell array is homogeneous or heterogeneous. See “Code Generation for Cell Arrays” on page 8-2. If you want to control whether the cell array is homogeneous or heterogeneous, specify the cell array by type. See “Specify a Cell Array Input Parameter” on page 17-22.

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.
3. Select **Define by Example**.
4. In the field to the right of the parameter, enter an example cell array.

   - If all cell array elements have the same properties, the cell array is homogeneous. For example, enter:

     ```
     {1 2 3}
     ```

     The input is a 1x3 cell array. The type of each element is `double(1x1)`.

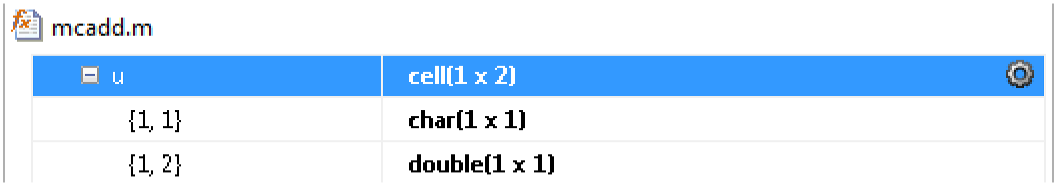
     mcadd.m

⊟ u	cell(1 x 3)
{:}	double(1 x 1)

     The colon inside curly braces`{:}` indicates that all elements have the same properties.
   - If elements of the cell array have different classes, the cell array is heterogeneous. For example, enter:

     ```
     {'a', 1}
     ```

The input is a 1x2 cell array. For a heterogeneous cell array, the app lists each element. The type of the first element is `char(1x1)`. The type of the second element is `double(1x1)`.

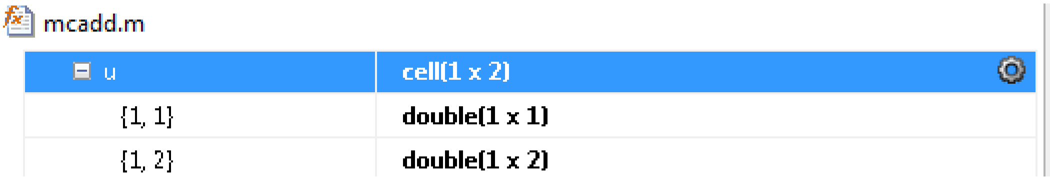


mcadd.m

u	cell(1 x 2)
{1, 1}	char(1 x 1)
{1, 2}	double(1 x 1)

- For some example cell arrays. the classification as homogeneous or heterogeneous is ambiguous. For these cell arrays, the app uses heuristics to determine whether the cell array is homogeneous or heterogeneous. For example, for the example cell array, enter:

  ```
  {1 [2 3]}
  ```
  The elements have the same class, but different sizes. The app determines that the input is a 1x2 heterogeneous cell array. The type of the first element is `double(1x1)`. The type of the second element is `double(1x2)`.



mcadd.m

u	cell(1 x 2)
{1, 1}	double(1 x 1)
{1, 2}	double(1 x 2)

However, the example cell array, `{1 [2 3]}`, can also be a homogeneous cell array whose elements are 1x:2 double. If you want this cell array to be homogeneous, do one of the following:

- Specify the cell array input by type. Specify that the input is a homogeneous cell array. Specify that the elements are 1x:2 double. See “Specify a Cell Array Input Parameter” on page 17-22.
- Right-click the variable. Select **Homogeneous**. Specify that the elements are 1x:2 double.

If you use `coder.typeof` to specify that the example cell array is variable size, the app makes the cell array homogeneous. For example, for the example input, enter:

```
coder.typeof({1 [2 3]}, [1 3], [0 1])
```

The app determines that the input is a 1x:3 homogeneous cell array whose elements are 1x:2 double.

To modify the type definition, see “Specify a Cell Array Input Parameter” on page 17-22.

Specify an Enumerated Type Input Parameter by Example

This example shows how to specify that an input uses the enumerated type `MyColors`.

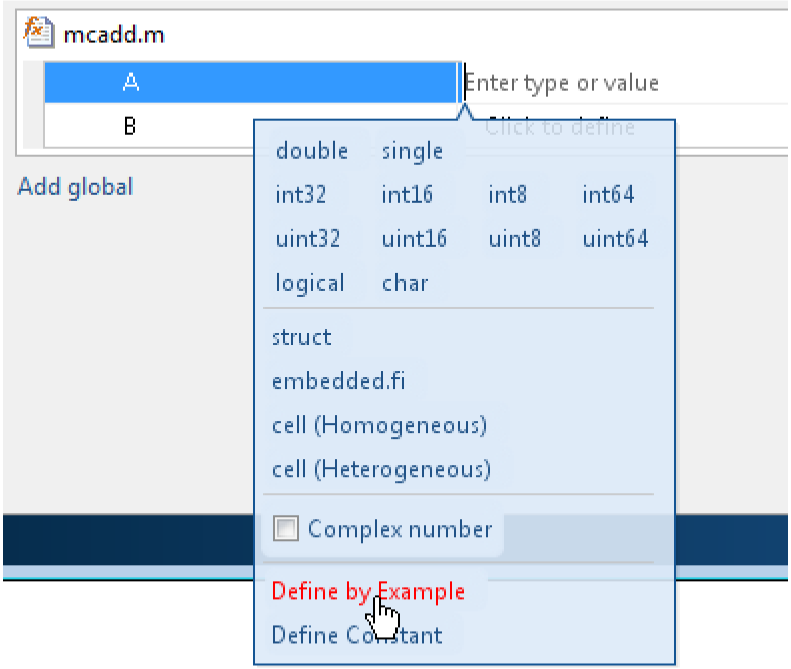
Suppose that `MyColors.m` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

To specify that an input has the enumerated type `MyColors`:

1 On the **Define Input Types** page, click **Let me enter input or global types directly**.

2 Click the field to the right of the input parameter that you want to define.



3 Select **Define by Example**.

4 In the field to the right of the parameter, enter the MATLAB expression:

```
MyColors.red
```

Specify an Object Input Type Parameter by Example

This example shows how to specify the type for an object of a value class `myRectangle`.

```
classdef myRectangle
    properties
        length;
        width;
    end
    methods
        function obj = myRectangle(l,w)
            if nargin > 0
                obj.length = l;
                obj.width = w;
            end
        end
        function area = calcarea(obj)
            area = obj.length * obj.width;
        end
    end
end
```

1 Define a function that takes an object of the value class as an input. For example:

   ```
   function z = getarea(r)
   %#codegen
   z = calcarea(r);
   end
   ```

2 In MATLAB, define an object `rect_obj`.

   ```
   rect_obj = myRectangle(3,4)
   ```

3 In the app, on the **Select Source Files** page, enter `getarea` for the entry-point function.
4 On the **Define Input Types** page, click **Let me enter input or global types directly**.
5 Click the field to the right of `r`.
6 Select **Define by Example**.
7 In the field to the right of `r`, enter `rect_obj` or select it from the list of workspace variables. The app determines that `r` is a class with properties `length` and `width`.

Alternatively, you can provide a `coder.ClassType` object for that class. To define a `coder.ClassType` object, use `coder.typeof`. For example:

1 In MATLAB, define a `coder.ClassType` object that has the same properties as `rect_obj`.

   ```
   t = coder.typeof(rect_obj)
   ```

2 In the app, provide `t` as the example.

To change the size or type of a property, click the field to the right of the property.

When you generate code, the properties that you define in the app must be consistent with the properties in the class definition file. If the class definition file has properties that your code does not use, your type definition in the app does not have to include those properties. The code generator removes properties that your code does not use.

See “Specify Objects as Inputs in the MATLAB Coder App” on page 10-33.

Specify a Fixed-Point Input Parameter by Example

To specify fixed-point inputs, Fixed-Point Designer software must be installed.

This example shows how to specify a signed fixed-point type with a word length of eight bits, and a fraction length of three bits.

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.
3. Select **Define by Example**.
4. In the field to the right of the parameter, enter:

   ```
   fi(10, 1, 8, 3)
   ```

   The app sets the type of input `u` to `fi(1x1)`. By default, if you do not specify a local `fimath`, the app uses the default `fimath`. See “fimath for Sharing Arithmetic Rules” (Fixed-Point Designer).

   Optionally, modify the fixed-point properties or the size of the input. See “Specify a Fixed-Point Input Parameter by Type” on page 17-19 and “Define or Edit Input Parameter Type by Using the App” on page 17-17.

Define or Edit Input Parameter Type by Using the App

In this section...

Define or Edit an Input Parameter Type

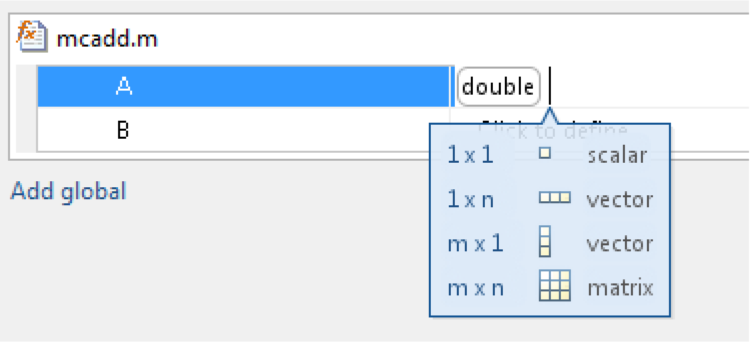
The following procedure shows you how to define or edit `double`, `single`, `int64`, `int32`, `int16`, `int8`, `uint64`, `uint32`, `uint16`, `uint8`, `logical`, and `char` types.

For more information about defining other types, see the following table.

Input Type	Link
A structure (**struct**)	“Specify a Structure Input Parameter” on page 17-19
A cell array (**cell (Homogeneous)** or **cell (Heterogeneous)**)	“Specify a Cell Array Input Parameter” on page 17-22
A fixed-point data type (**embedded.fi**)	“Specify a Fixed-Point Input Parameter by Type” on page 17-19
An input by example (**Define by Example**)	“Define Input Parameter by Example by Using the App” on page 17-7
A constant (**Define Constant**)	“Define Constant Input Parameters Using the App” on page 17-27

1. Click the field to the right of the input parameter name.
2. Optionally, for numeric types, to make the parameter a complex type, select the **Complex number** check box.
3. Select the input type.

   The app displays the selected type. It displays the size options.



4 From the list, select whether your input is a scalar, a `1 x n` vector, a `m x 1` vector, or a `m x n` matrix. By default, if you do not select a size option, the app defines inputs as scalars.

5 Optionally, if your input is not scalar, enter sizes `m` and `n`. You can specify:

- Fixed size, for example, `10`.
- Variable size, up to a specified limit, by using the `:` prefix. For example, to specify that your input can vary in size up to `10`, enter `:10`.
- Unbounded variable size by entering `:Inf`.

You can edit the size of each dimension.

Specify an Enumerated Type Input Parameter by Type

To specify that an input uses the enumerated type `MyColors`:

1 Suppose that the enumeration `MyColors` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

2 On the **Define Input Types** page, click **Let me enter input or global types directly**.

3 In the field to the right of the input parameter, enter `MyColors`.

Specify a Fixed-Point Input Parameter by Type

To specify fixed-point inputs, Fixed-Point Designer software must be installed.

1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
2 Click the field to the right of the input parameter that you want to define.
3 Select **embedded.fi**.
4 Select the size. If you do not specify the size, the size defaults to `1x1`.
5 Specify the input parameter `numerictype` and `fimath` properties.

   If you do not specify a local fimath, the app uses the default fimath. See "Default fimath Usage to Share Arithmetic Rules" (Fixed-Point Designer).

To modify the `numerictype` or `fimath` properties, open the properties dialog box. To open the properties dialog box, click to the right of the fixed-point type definition. Optionally, click ⚙.

Specify a Structure Input Parameter

When a primary input is a structure, the app treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition:

- For each field of an input structure, specify class, size, and complexity.
- For each field that is a fixed-point class, also specify numerictype, and fimath.

Specify Structures by Type

1 On the **Define Input Types** page, click **Let me enter input or global types directly**.
2 Click the field to the right of the input parameter that you want to define.
3 Select **struct**.

   The app displays the selected type, `struct`. The app displays the size options.
4 Specify that your structure is a scalar, `1 x n` vector, `m x 1` vector, or `m x n` matrix. By default, if you do not select a size option, the app defines inputs as scalars.
5 If your input is not scalar, enter sizes for each dimension. Click the dimension. Enter the size. Select from the size options. For example, for size `10`:

- To specify fixed size, select `10`.
- To specify variable size with an upper bound of `10`, select `:10`.
- To specify unbounded variable size, select `:Inf`.

6 Optionally, specify properties for the structure in the generated code. See "Set Structure Properties" on page 17-20.

7 Add fields to the structure. Specify the class, size, and complexity of the fields. See "Add a Field to a Structure" on page 17-22.

Set Structure Properties

1 Click to the right of the structure definition. Optionally, click ⚙.

2 In the dialog box, specify properties for the structure in the generated code.

Property	Description
C type definition name	Name for the structure type in the generated code.
Type definition is externally defined	Default: `No` — type definition is not externally defined. If you select `Yes` to declare an externally defined structure, the app does not generate the definition of the structure type. You must provide it in a custom include file. Dependency: `C type definition name` enables this option.
C type definition header file	Name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`. Specify the path to the file using the **Additional include directories** parameter on the project settings dialog box **Custom Code** tab. By default, the generated code contains `#include` statements for custom header

Property	Description
	files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. If you specify the C type definition header file, the app includes that header file exactly at the point where it is required. Dependency: When `Type definition is externally defined` is set to `Yes`, this option is enabled.
Data alignment boundary	The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. You can take advantage of target-specific function implementations that require aligned data. By default, the structure is not aligned on any specific boundary so it is not matched by CRL functions that require alignment. Alignment must be either `-1` or a power of `2` that is no more than `128`. Default: `0` Dependency: When `Type definition is externally defined` is set to `Yes`, this option is enabled.

Rename a Field in a Structure

Select the name field of the structure that you want to rename. Enter the new name.

Add a Field to a Structure

1. To the right of the structure, click +
2. Enter the field name. Specify the class, size, and complexity of the field.

Insert a Field into a Structure

1. Select the structure field below which you want to add another field.
2. Right-click the structure field.
3. Select **Insert Field Below**.

   The app adds the field after the field that you selected.
4. Enter the field name. Specify the class, size, and complexity of the field.

Remove a Field from a Structure

1. Right-click the field that you want to remove.
2. Select **Remove Field**.

Specify a Cell Array Input Parameter

For code generation, cell arrays are homogeneous or heterogeneous. See “Code Generation for Cell Arrays” on page 8-2. A homogeneous cell array is represented as an array in the generated code. All elements have the same properties. A heterogeneous cell array is represented as a structure in the generated code. Elements can have different properties.

Specify a Homogeneous Cell Array

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.
3. Select **cell (Homogeneous)**.

   The app displays the selected type, `cell`. The app displays the size options.
4. From the list, select whether your input is a scalar, a `1 x n` vector, a `m x 1` vector, or a `m x n` matrix. By default, if you do not select a size option, the app defines inputs as scalars.

5. If your input is not scalar, enter sizes for each dimension. Click the dimension. Enter the size. Select from the size options. For example, for size `10`:

   - To specify fixed size, select `10`.
   - To specify variable size with an upper bound of `10`, select `:10`.
   - To specify unbounded variable size, select `:Inf`.

   Below the cell array variable, a colon inside curly braces `{:}` indicates that the cell array elements have the same properties (class, size, and complexity).
6. To specify the class, size, and complexity of the elements in the cell array, click the field to the right of `{:}`.

Specify a Heterogeneous Cell Array

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.
3. Select **cell (Heterogeneous)**.

   The app displays the selected type, `cell`. The app displays the size options.
4. Specify that your structure is a scalar, `1 x n` vector, `m x 1` vector, or `m x n` matrix. By default, if you do not select a size option, the app defines inputs as scalars.
5. Optionally, if your input is not scalar, enter sizes `m` and `n`. A heterogeneous cell array is fixed size.

   The app lists the cell array elements. It uses indexing notation to specify each element. For example, `{1,2}` indicates the element in row 1, column 2.
6. Specify the class, size, and complexity for each cell array element.
7. Optionally, add elements. See “Add an Element to a Heterogeneous Cell Array” on page 17-26
8. Optionally, specify properties for the structure that represents the cell array in the generated code. See “Set Structure Properties for a Heterogeneous Cell Array” on page 17-23.

Set Structure Properties for a Heterogeneous Cell Array

A heterogeneous cell array is represented as a structure in the generated code. You can specify the properties for the structure that represents the cell array.

1 Click to the right of the cell array definition. Optionally click ⚙.

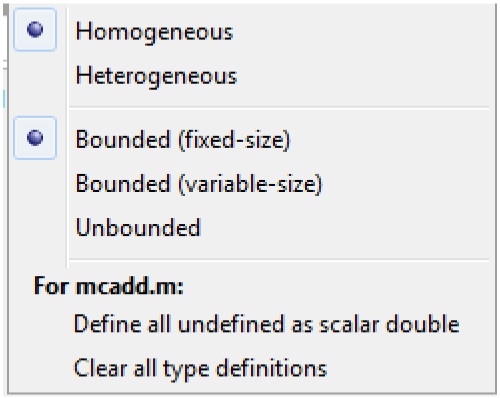
2 In the dialog box, specify properties for the structure in the generated code.

Property	Description
C type definition name	Name for the structure type in the generated code.
Type definition is externally defined	Default: `No` — type definition is not externally defined. If you select `Yes` to declare an externally defined structure, the app does not generate the definition of the structure type. You must provide it in a custom include file. Dependency: `C type definition name` enables this option.
C type definition header file	Name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`. Specify the path to the file using the **Additional include directories** parameter on the project settings dialog box **Custom Code** tab. By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. If you specify the C type definition header file, the app includes that header file exactly at the point where it is required. Dependency: When `Type definition is externally defined` is set to `Yes`, this option is enabled.

Property	Description
Data alignment boundary	The run-time memory alignment of structures of this type in bytes. If you have an Embedded Coder license and use Code Replacement Libraries (CRLs), the CRLs provide the ability to align data objects passed into a replacement function to a specified boundary. You can take advantage of target-specific function implementations that require aligned data. By default, the structure is not aligned on any specific boundary so it is not matched by CRL functions that require alignment. Alignment must be either `-1` or a power of `2` that is no more than `128`. Default: `0` Dependency: When `Type definition is externally defined` is set to `Yes`, this option is enabled.

Change Classification as Homogeneous or Heterogeneous

To change the classification as homogeneous or heterogeneous, right-click the variable. Select **Homogeneous** or **Heterogeneous**.



The app clears the definitions of the elements.

Change the Size of the Cell Array

1. In the definition of the cell array, click a dimension. Specify the size.
2. For a homogeneous cell array, specify whether the dimension is variable size and whether the dimension is bounded or unbounded. Alternatively, right-click the variable. Select **Bounded (fixed-size)**, **Bounded (variable-size)**, or **Unbounded**
3. For a heterogeneous cell array, the app adds elements so that the cell array has the specified size and shape.

Add an Element to a Heterogeneous Cell Array

1. In the definition of the cell array, click a dimension. Specify the size. For example, enter 1 for the first dimension and 4 for the second dimension.

   The app adds elements so that the cell array has the specified size and shape. For example for a 1x4 heterogeneous cell array, the app lists four elements: `{1,1}`, `{1,2}`, `{1,3}`, and `{1,4}`.
2. Specify the properties of the new elements.

Define Constant Input Parameters Using the App

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter name.
3. Select **Define Constant**.
4. In the field to the right of the parameter name, enter the value of the constant or a MATLAB expression that represents the constant.

   The app uses the value of the specified MATLAB expression as a compile-time constant.

Define Inputs Programmatically in the MATLAB File

You can use the MATLAB `assert` function to define properties of entry-point function inputs in your MATLAB entry-point files.

To instruct the MATLAB Coder app to determine input types from the assert statements in your code, on the app toolbar, click ☰. Select **Determine input types from code preconditions**. If you enable this option:

- The app labels all entry-point function inputs as `Deferred`. It determines the input types at compile time.
- In this project, you cannot use other input specification methods to specify input types.

See “Define Input Properties Programmatically in the MATLAB File” on page 20-68.

Note: If you enable fixed-point conversion (requires a Fixed-Point Designer license), the app disables the **Determine input types from code preconditions** option.

Add Global Variables by Using the App

To add global variables to the project:

1 On the **Define Input Types** page, automatically define input types or click **Let me enter input or global types directly**.

  The app displays a table of entry-point inputs.

2 To add a global variable, click **Add global**.

  By default, the app names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, and so on.

3 Under **Global variables**, enter a name for the global variable.

4 After adding a global variable, but before generating code, specify its type and initial value. Otherwise, you must create a variable with the same name in the global workspace. See "Specify Global Variable Type and Initial Value Using the App" on page 17-30.

Specify Global Variable Type and Initial Value Using the App

In this section...

Why Specify a Type Definition for Global Variables?

If you use global variables in your MATLAB algorithm, before building the project, you must add a global type definition and initial value for each global variable. If you do not initialize the global data, the app looks for the variable in the MATLAB global workspace. If the variable does not exist, the app generates an error.

For MEX functions, if you use global data, you must also specify whether to synchronize this data between MATLAB and the MEX function.

Specify a Global Variable Type

1. Specify the type of each global variable using one of the following methods:
   - Define by example
   - Define type
2. Define an initial value for each global variable.

If you do not provide a type definition and initial value for a global variable, create a variable with the same name and suitable class, size, complexity, and value in the MATLAB workspace.

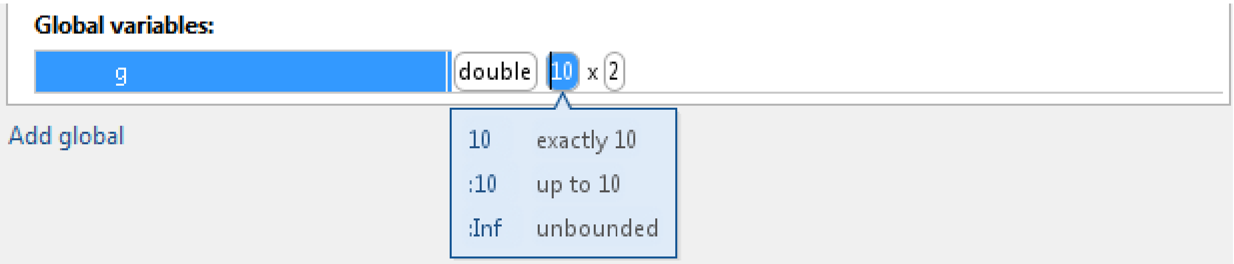
Define a Global Variable by Example

1. Click the field to the right of the global variable that you want to define.

2 Select `Define by Example`.

3 In the field to the right of the global name, enter a MATLAB expression that has the required class, size, and complexity. MATLAB Coder software uses the class, size, and complexity of the value of this expression as the type for the global variable.

4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, `10`.



You can specify:

- Fixed size. In this example, select `10`.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to `10`, select `:10`.
- Unbounded variable size by selecting `:Inf`.

Define or Edit Global Variable Type

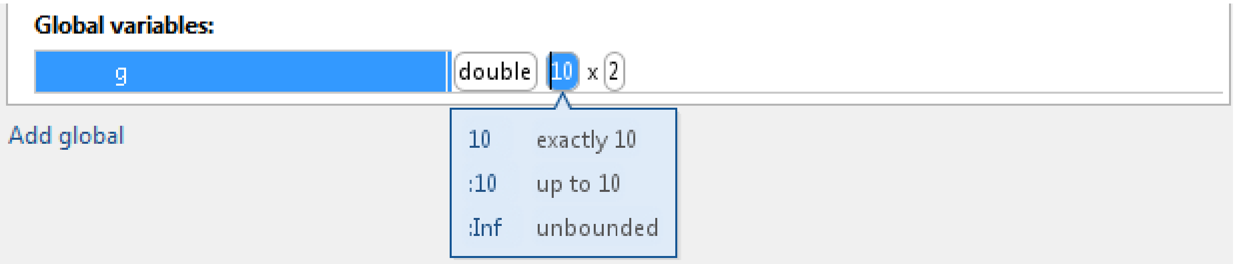
1 Click the field to the right of the global variable that you want to define.

2 Optionally, for numeric types, select **Complex** to make the parameter a complex type. By default, inputs are real.

3 Select the type for the global variable. For example, `double`.

By default, the global variable is a scalar.

4 Optionally, change the size of the global variable. Click the dimension that you want to change and enter the size, for example, `10`.



You can specify:

- Fixed size. In this example, select `10`.
- Variable size, up to a specified limit, by using the `:` prefix. In this example, to specify that your input can vary in size up to `10`, select `:10`.
- Unbounded variable size by selecting `:Inf`.

Define Global Variable Initial Value

- “Define Initial Value Before Defining Type” on page 17-32
- “Define Initial Value After Defining Type” on page 17-33

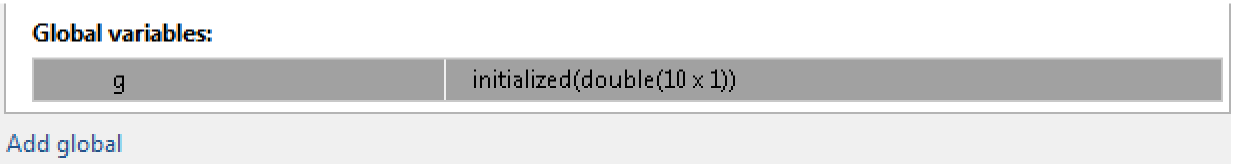
Define Initial Value Before Defining Type

1 Click the field to the right of the global variable.

2 Select `Define Initial Value`.

3 Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable. Because you did not define the type of the global variable before you defined its initial value, MATLAB Coder uses the initial value type as the global variable type.

The project shows that the global variable is initialized.

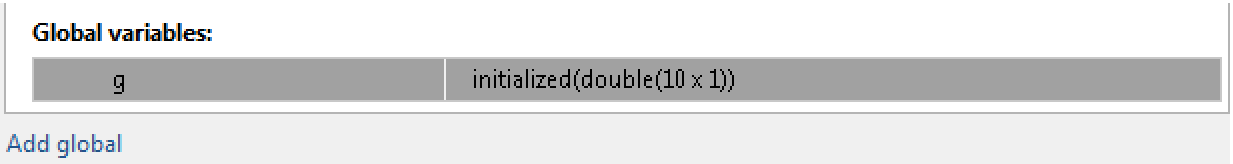


If you change the type of a global variable after defining its initial value, you must redefine the initial value.

Define Initial Value After Defining Type

- Click the type field of a predefined global variable.
- Select `Define Initial Value`.
- Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

  The project shows that the global variable is initialized.



g	initialized(double(10 x 1))

Define Global Variable Constant Value

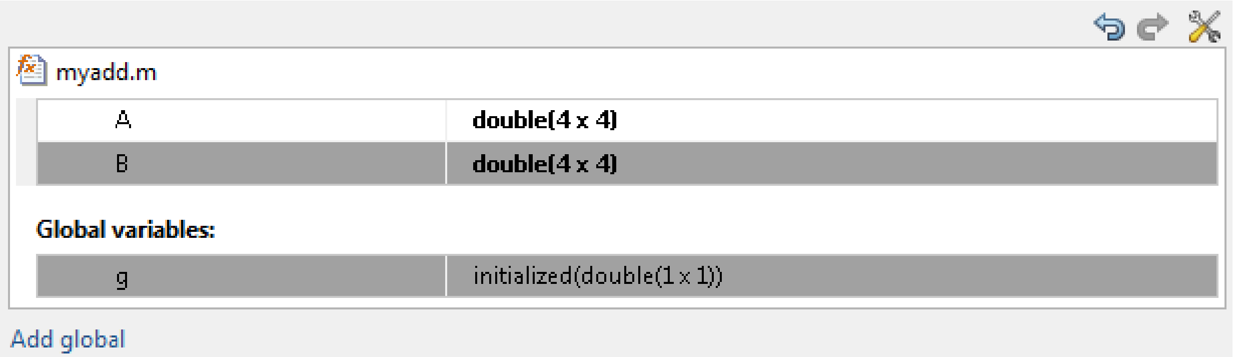
1. Click the field to the right of the global variable.
2. Select `Define Constant Value`.
3. In the field to the right of the global variable, enter a MATLAB expression.

Remove Global Variables

1. Right-click the global variable.
2. From the menu, select **Remove Global**.

Undo and Redo Changes to Type Definitions in the App

To revert or restore changes to input argument or global variable type definitions, above the input arguments table, click ↶ or ↷.



myadd.m

A	double(4 x 4)
B	double(4 x 4)

Global variables:

g	initialized(double(1 x 1))

Add global

Alternatively, use the keyboard shortcuts for Undo and Redo. The shortcuts are defined in your MATLAB preferences. On a Windows platform, the default keyboard shortcuts for Undo and Redo are **Ctrl+Z** and **Ctrl+Y**.

Each undo operation reverts the last change. Each redo operation restores the last change.

Related Examples

- “Define Keyboard Shortcuts” (MATLAB)

Changing Output Type

In this section...

MEX functions use a different set of configuration parameters than libraries and executables use. When you switch the output type between `MEX Function` and `Source Code`, `Static Library`, `Dynamic Library`, or `C/C++ Executable`, verify these settings.

If you enable any of the following parameters when the output type is `MEX Function`, and you want to use the same setting for C/C++ code generation as well, you must enable it again for `C/C++ Static Library`, `C/C++ Dynamic Library`, and `C/C++ Executable`.

Project Settings

Project Settings Dialog Box Tab	Parameter Name
Paths	Working folder
	Build folder
	Search paths
Speed	Saturate on integer overflow
Memory	Enable variable-sizing
	Dynamic memory allocation
	Stack usage max
Code Appearance	Generated file partitioning method
	Include comments
	MATLAB source code as comments
	Reserved names
Debugging	Always create a code generation report
	Automatically launch a report if one is generated
Custom Code	Source file

Project Settings Dialog Box Tab	Parameter Name
	Header file
	Initialize function
	Terminate function
	Additional include directories
	Additional source files
	Additional libraries
	Post-code-generation command
Advanced	Constant folding timeout
	Language
	Inline threshold
	Inline threshold max
	Inline stack limit
	Use memcpy for vector assignment
	Memcpy threshold (bytes)
	Use memset to initialize floats and doubles to 0.0

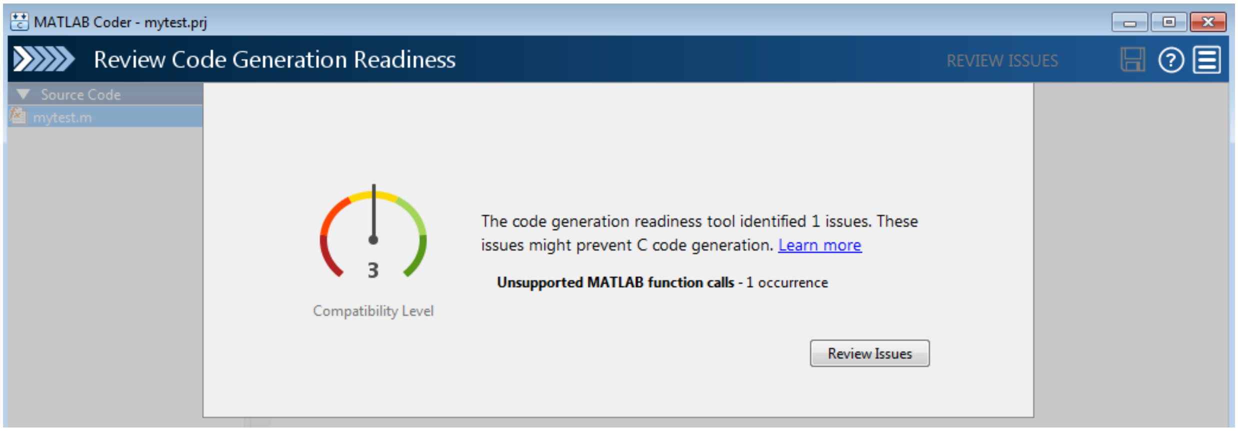
Configuration Object Parameters

- ConstantFoldingTimeout
- CustomHeaderCode
- CustomInclude
- CustomInitializer
- CustomLibrary
- CustomSource
- CustomSourceCode
- CustomTerminator
- DynamicMemoryAllocation
- EnableMemcpy
- EnableVariableSizing

- FilePartitionMethod
- GenCodeOnly
- GenerateComments
- GenerateReport
- InitFltsAndDblsToZero
- InlineStackLimit
- InlineThreshold
- InlineThresholdMax
- LaunchReport
- MATLABSourceComments
- MemcpyThreshold
- PostCodeGenCommand
- ReservedNameArray
- SaturateOnIntegerOverflow
- StackUsageMax
- TargetLang

Code Generation Readiness Screening in the MATLAB Coder App

By default, the MATLAB Coder app screens your MATLAB code for features and functions that code generation does not support. After you enter entry-point functions and click **Next**, if the app detects issues, it opens the **Review Code Generation Readiness** page.



The code generation readiness tool identified 1 issues. These issues might prevent C code generation. Learn more

Unsupported MATLAB function calls - 1 occurrence

If you click **Review Issues**, you can use the app editor to fix issues before you generate code.

If the code generation readiness screening causes slow operations in the app, consider disabling the screening. To disable code generation readiness screening, on the app toolbar, click ☰ and clear **Check code generation readiness**.

If you clear **Check code generation readiness** during or after screening, the app retains the screening results for the current session. If you fix or introduce code generation readiness issues in your code, the app does not update the screening results. To clear screening results after you disable screening, or to update screening results after you reenable screening, close and reopen the project.

For a fixed-point conversion project, code generation readiness screening identifies functions that do not have fixed-point support. The app lists these functions on the **Function Replacements** tab of the **Convert to Fixed Point** page where you can specify function replacement with a custom function or a lookup table. If you disable screening, do not rely on the app to identify functions that you must replace. Manually enter the names of functions on the **Function Replacements** tab. Fixed-point conversion requires a Fixed-Point Designer license.

More About

- “Slow Operations in MATLAB Coder App” on page 17-40
- “Automated Fixed-Point Conversion” on page 14-83

Slow Operations in MATLAB Coder App

By default, the MATLAB Coder app screens your entry-point functions for code generation readiness. For some large entry-point functions, or functions with many calls, screening can take a long time. If the screening takes a long time, certain app or MATLAB operations can be slower than expected or appear to be unresponsive.

To determine if slow operations are due to the code generation readiness screening, disable the screening.

More About

- "Code Generation Readiness Screening in the MATLAB Coder App" on page 17-38

Unable to Open a MATLAB Coder Project

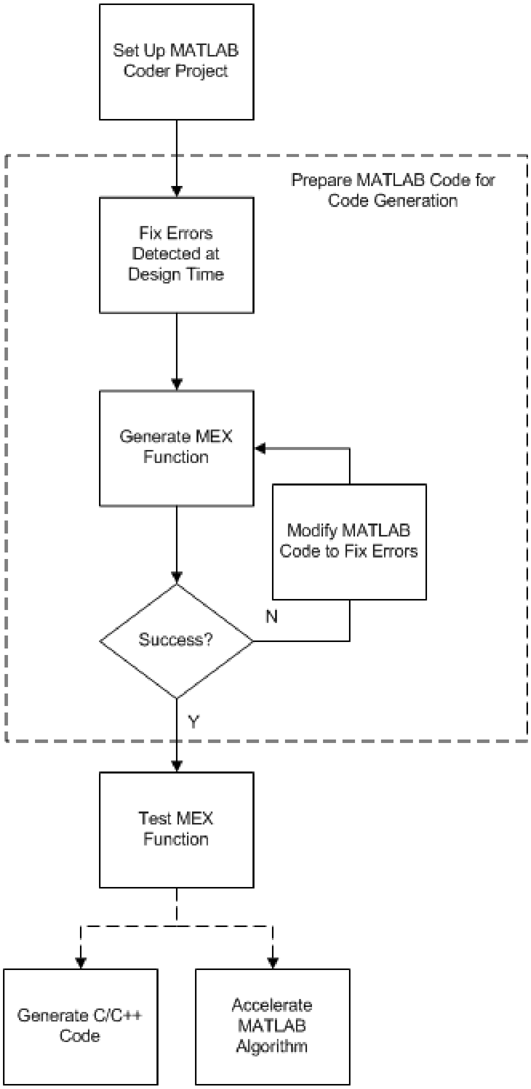
When you open a project from a different release, if necessary, the MATLAB Coder app updates the project file so that the format is compatible with the release that you are using. Before the app updates the project file, it creates a backup file with the name `*project_name*.prj.bak`. For example, the backup file name for `myproject.prj` is `myproject.prj.bak`. If the backup file exists, the app inserts an integer between the `prj` and `bak` extensions to make the file name unique. For example, if `myproject.prj.bak` exists, the app creates the backup file `myproject.prj.2.bak`.

If the project file is from a release before R2015a, the app also displays a message about the project file update and backup. To use the project in a release before R2015a, use the backup project file instead of the updated project file.

To use a backup project file, remove the extensions that follow the `prj` extension. For example, rename `myproject.prj.2.bak` to `myproject.prj`. If you use the backup project file in the release that created it, the project is the same as the original project. If you use the backup project file in a different release than the one that created it, you can possibly lose some information. For example, if you open a project file in a release that does not recognize a setting in the file, that setting is lost. For best results, open the backup project file in the release in which you created it.

Preparing MATLAB Code for C/C++ Code Generation

Workflow for Preparing MATLAB Code for Code Generation


Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Fix Errors Detected at Design Time
Generate MEX Function
Modify MATLAB Code to Fix Errors
N
Success?
Y
Test MEX Function
Generate C/C++ Code
Accelerate MATLAB Algorithm

See Also

- “Set Up a MATLAB Coder Project” on page 17-2
- “Fixing Errors Detected at Design Time” on page 18-4
- “Generate MEX Functions by Using the MATLAB Coder App” on page 18-16
- “Fix Errors Detected at Code Generation Time” on page 18-23
- “Workflow for Testing MEX Functions in MATLAB” on page 19-3
- “C/C++ Code Generation” on page 20-4
- “Accelerate MATLAB Algorithms” on page 25-13

Fixing Errors Detected at Design Time

Use the code analyzer and the code generation readiness tool to detect issues at design time. Before generating code, you must fix these issues.

See Also

- “Check Code with the Code Analyzer” on page 18-6
- “Check Code by Using the Code Generation Readiness Tool” on page 18-8
- “Design Considerations When Writing MATLAB Code for Code Generation” on page 18-24
- “Debugging Strategies” on page 18-27

Using the Code Analyzer

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing compilation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications.

To use the code analyzer to identify warnings and errors specific to MATLAB for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of code generation analyzer messages is available in the MATLAB Code Analyzer preferences. For more information, see "Running the Code Analyzer Report" (MATLAB).

Note: The code analyzer might not detect all MATLAB for code generation issues. After eliminating the errors or warnings that the code analyzer detects, compile your code with MATLAB Coder to determine if the code has other compliance issues.

Check Code with the Code Analyzer

The code analyzer checks your code for problems and recommends modifications. You can use the code analyzer to check your code interactively in the MATLAB Editor while you work.

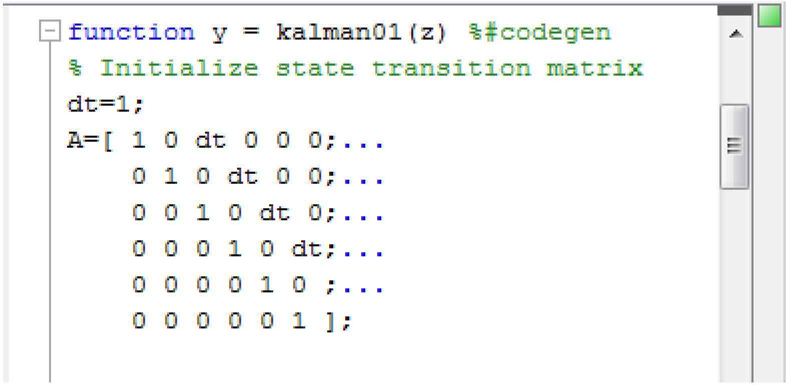
To verify that continuous code checking is enabled:

1 In MATLAB, select the **Home** tab and then click **Preferences**.

2 In the **Preferences** dialog box, select **Code Analyzer**.

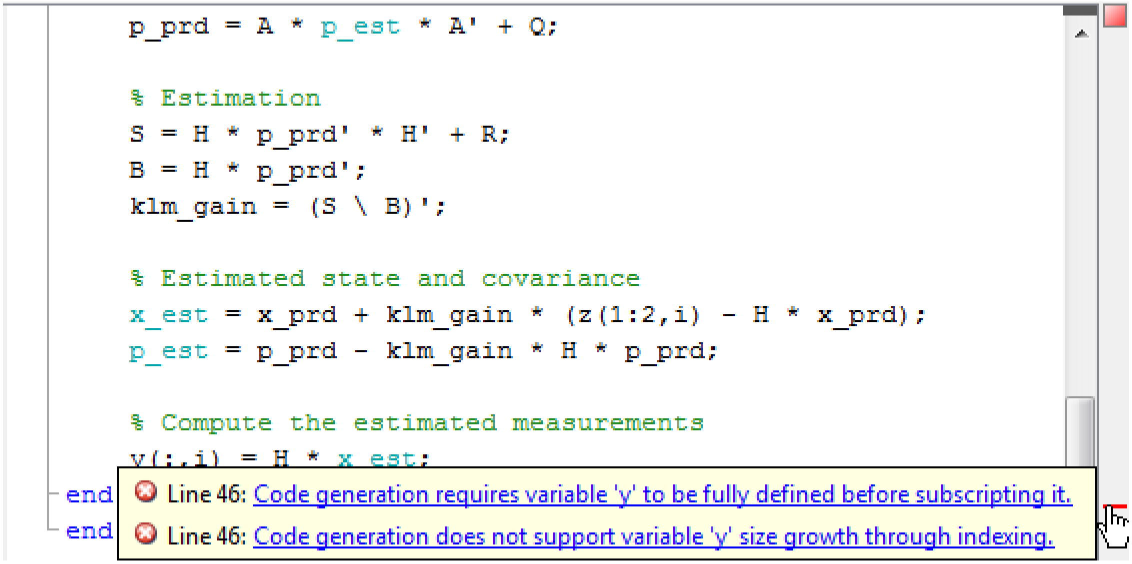
3 In the **Code Analyzer Preferences** pane, verify that **Enable integrated warning and error messages** is selected.

The code analyzer provides an indicator in the top right of the editor window. If the indicator is green, the analyzer did not detect code generation issues.



```
function y = kalman01(z) %#codegen
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

If the indicator is red, the analyzer has detected errors in your code. If it is orange, it has detected warning. When the indicator is red or orange, a red or orange marker appears to the right of the code where the error occurs. Place your pointer over the marker for information about the error. Click the underlined text in the error message for a more detailed explanation and suggested actions to fix the error.



```
    p_prd = A * p_est * A' + Q;

    % Estimation
    S = H * p_prd' * H' + R;
    B = H * p_prd';
    klm_gain = (S \ B)';

    % Estimated state and covariance
    x_est = x_prd + klm_gain * (z(1:2,i) - H * x_prd);
    p_est = p_prd - klm_gain * H * p_prd;

    % Compute the estimated measurements
    y(:,i) = H * x_est;
end
end
```

Line 46: Code generation requires variable 'y' to be fully defined before subscripting it.

Line 46: Code generation does not support variable 'y' size growth through indexing.

Before generating code from your MATLAB code, you must fix the errors detected by the code analyzer.

Check Code by Using the Code Generation Readiness Tool

In this section...

Run Code Generation Readiness Tool at the Command Line

1. Navigate to the folder that contains the file that you want to check for code generation readiness.
2. At the command prompt, enter:

   ```
   coder.screener('filename')
   ```

   The **Code Generation Readiness** tool opens for the file named `filename`. The tool provides a code generation readiness score and lists issues that you must fix prior to code generation.

Run Code Generation Readiness Tool from the Current Folder Browser

1. In the current folder browser, right-click the file that you want to check for code generation readiness.
2. From the context menu, select `Check Code Generation Readiness`.

   The **Code Generation Readiness** tool opens for the selected file. It provides a code generation readiness score and lists issues that you must fix prior to code generation.

Run the Code Generation Readiness Tool Using the MATLAB Coder App

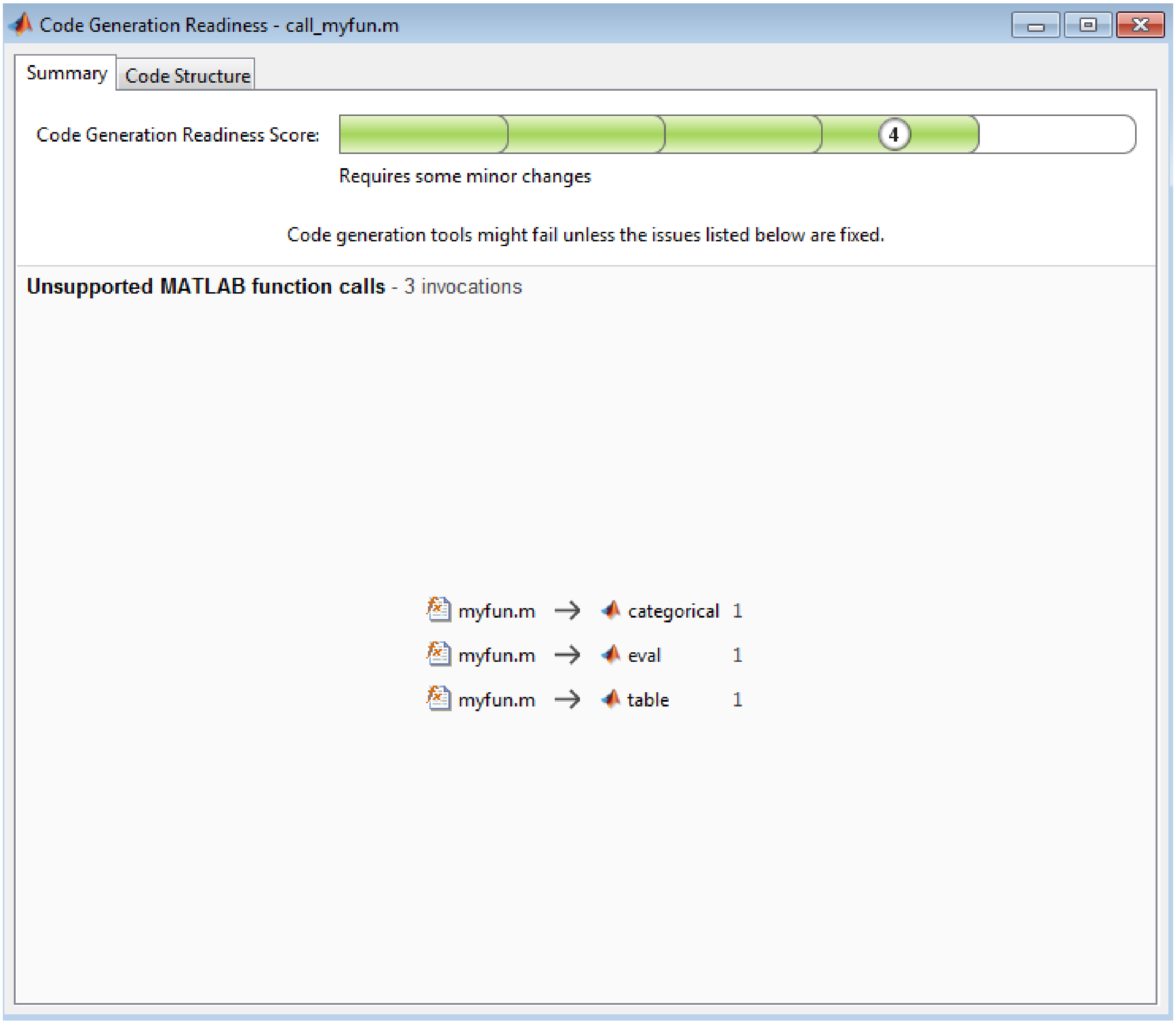
After you add entry-point files to your project, the MATLAB Coder app analyzes the functions for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page. You can review and fix issues.

See "Code Generation Readiness Tool" on page 18-9.

Code Generation Readiness Tool

The code generation readiness tool screens MATLAB code for features and functions that code generation does not support. The tool provides a report that lists the source files that contain unsupported features and functions. The report also indicates the amount of work required to make the MATLAB code suitable for code generation. It is possible that the tool does not detect all code generation issues. Under certain circumstances, it is possible that the tool can report false errors. Therefore, before you generate C code, verify that your code is suitable for code generation by generating a MEX function.

Summary Tab



Code Generation Readiness Score: 4

Requires some minor changes

Code generation tools might fail unless the issues listed below are fixed.

Unsupported MATLAB function calls - 3 invocations

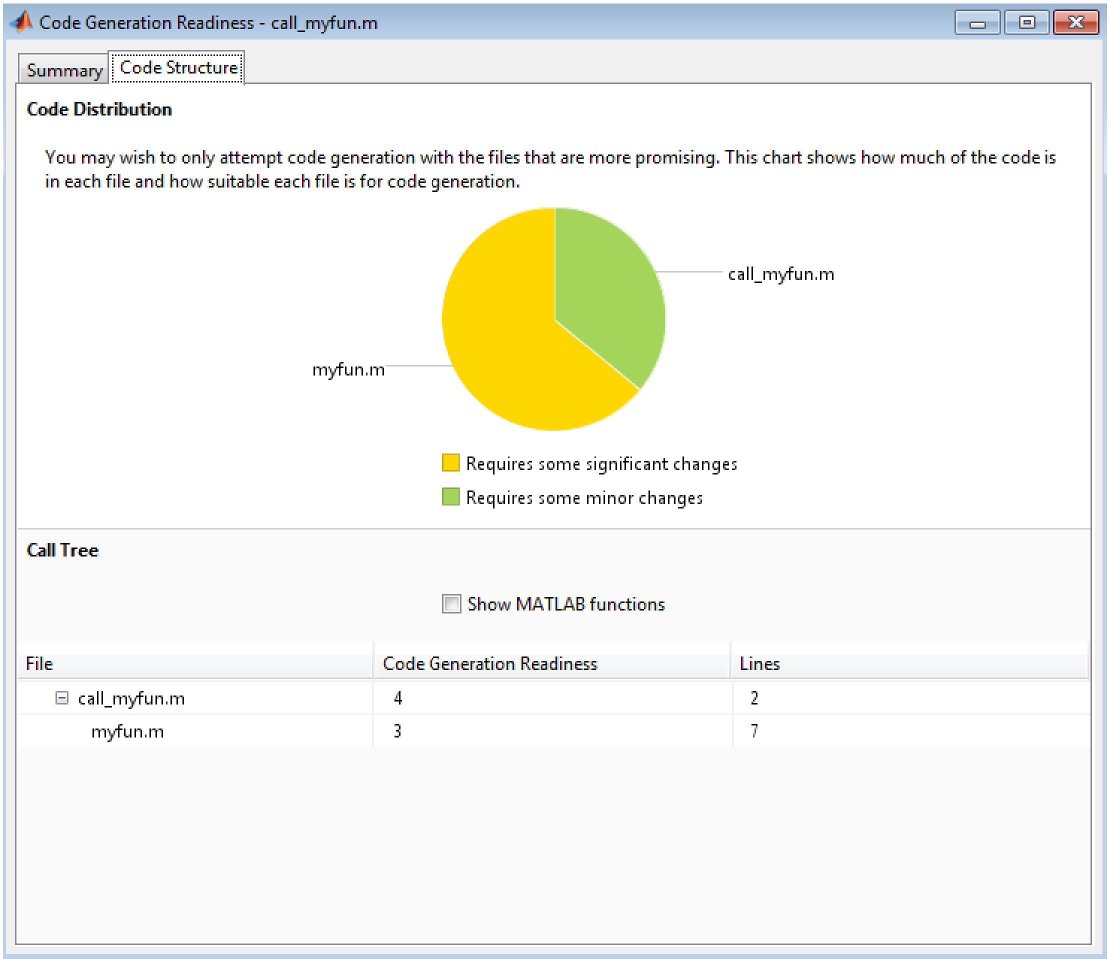
```
myfun.m  →  categorical  1
myfun.m  →  eval         1
myfun.m  →  table        1
```

The **Summary** tab provides a **Code Generation Readiness Score**, which ranges from 1 to 5. A score of 1 indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool does not detect code generation issues; the code is ready to use with minimal or no changes.

On this tab, the tool also displays information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. To learn more about the issues and how to fix them, use the Code Analyzer.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features.
- Unsupported data types.

Code Structure Tab



Code Distribution

You may wish to only attempt code generation with the files that are more promising. This chart shows how much of the code is in each file and how suitable each file is for code generation.

🟨 Requires some significant changes

🟩 Requires some minor changes

Call Tree

□ Show MATLAB functions

File	Code Generation Readiness	Lines
⊟ call_myfun.m	4	2
myfun.m	3	7

If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab displays information about the relative size of each file and how suitable each file is for code generation.

Code Distribution

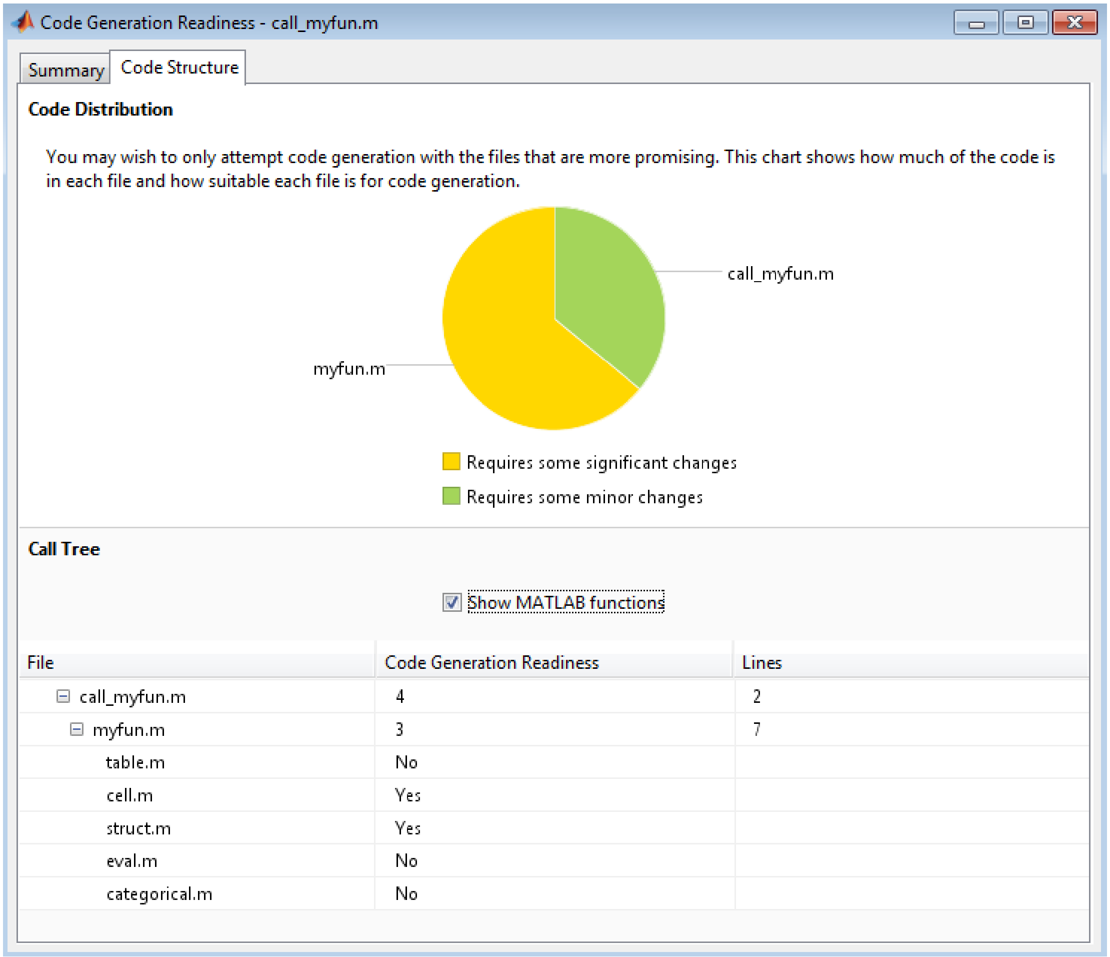
The **Code Distribution** pane displays a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. During the planning phase of a project, you can use this information for estimation and scheduling. If the report indicates that multiple files are not suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane displays information about the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score, which ranges from `1` to `5`. A score of `1` indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of `5` indicates that the tool does not detect code generation issues. The code is ready to use with minimal or no changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions

If you select **Show MATLAB Functions**, the report also lists the MATLAB functions that your function calls. For each of these MATLAB functions, if code generation supports the function, the report sets **Code Generation Readiness** to `Yes`.



Code Generation Readiness - call_myfun.m

Summary | Code Structure

Code Distribution

You may wish to only attempt code generation with the files that are more promising. This chart shows how much of the code is in each file and how suitable each file is for code generation.

🟨 Requires some significant changes

🟩 Requires some minor changes

Call Tree

☑ Show MATLAB functions

File	Code Generation Readiness	Lines
⊟ call_myfun.m	4	2
⊟ myfun.m	3	7
table.m	No	
cell.m	Yes	
struct.m	Yes	
eval.m	No	
categorical.m	No	

Related Examples

- “Check Code by Using the Code Generation Readiness Tool” on page 18-8

Unable to Determine Code Generation Readiness

Sometimes the code generation readiness tool cannot determine whether the entry-point functions in your project are suitable for code generation. The most likely reason is that the tool is unable to find the entry-point files. Verify that your current working folder is set to the folder that contains your entry-point files. If it is not, either make this folder your current working folder or add the folder containing these files to the MATLAB path.

Generate MEX Functions by Using the MATLAB Coder App

In this section...

Workflow for Generating MEX Functions Using the MATLAB Coder App

Step	Action	Details
1	Set up the MATLAB Coder project.	"Set Up a MATLAB Coder Project" on page 17-2
2	Specify the build configuration parameters. Set **Build type** to `MEX`.	"Configure Project Settings" on page 18-19
3	Build the project.	"Build a MATLAB Coder Project" on page 18-19

Generate a MEX Function Using the MATLAB Coder App

This example shows how to generate a MEX function from MATLAB code using the MATLAB Coder app.

Create the Entry-Point Function

In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

Create the Test File

In the same local writable folder, create a MATLAB file, `mcadd_test.m`, that calls `mcadd` with example inputs. The example inputs are scalars with type `int16`.

```
function y = mcadd_test
```

```
y = mcadd(int16(2), int16(3));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

1. On the **Select Source Files** page, type or select the name of the entry-point function `mcadd`.

   The app creates a project with the default name `mcadd.prj`.
2. Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify the test file `mcadd_test.m` that MATLAB Coder uses to automatically define types for `u` and `v`:

1. Enter or select the test file `mcadd_test.m`.
2. Click **Autodefine Input Types**.

   The test file, `mcadd_test.m`, calls the entry-point function, `mcadd`, with the example input types. MATLAB Coder infers that inputs `u` and `v` are `int16(1x1)`.
3. Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow ▼.

   The app populates the test file field with `mcadd_test`, the test file that you used to define the input types.
2. Click **Check for Issues**.

   The app generates a MEX function. It runs the test file replacing calls to `mcadd` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.
3. Click **Next** to go to the **Generate Code** step.

Generate the MEX Function

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. In the **Generate** dialog box, set **Build type** to `MEX` and **Language** to C. Use the default values for the other project build configuration settings.
3. Click **Generate**.

   The app indicates that code generation succeeded. It displays the source MATLAB files and the generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Target Build Log** tab, it displays the build log, including compiler warnings and errors.

   MATLAB Coder builds the project and, by default, generates a MEX function, `mcadd_mex`, in the current folder. MATLAB Coder also generates other supporting files in a subfolder called `codegen/mex/mcadd`. MATLAB Coder uses the name of the MATLAB function as the root name for the generated files. It creates a platform-specific extension for the MEX file. See “Naming Conventions” on page 20-83.
4. To view the code generation report, click **View Report**.
5. Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to the generated output.

Configure Project Settings

To open the project settings dialog box:

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. Click **More Settings**.

To change a project setting, click the tab that contains the setting that you want to change. For example, to change the **Saturate on integer overflow** setting, click the **Speed** tab.

MEX functions use a different set of configuration parameters than libraries and executables. When you change the output type from `MEX Function` to `Source Code Static Library`, `Dynamic Library`, or `Executable`, verify these settings. See “Changing Output Type” on page 17-35.

See Also

- “Enable Code Generation Reports” on page 21-27
- “Using the MATLAB Coder App” on page 20-126
- “How to Disable Inlining Globally Using the MATLAB Coder App” on page 20-136
- “Include MATLAB Source Code as Comments by Using the MATLAB Coder App” on page 21-2
- “Disabling Run-Time Checks Using the MATLAB Coder App” on page 25-16

Build a MATLAB Coder Project

To build a project using the specified settings, on the **Generate Code** page, click **Generate**. As the MATLAB Coder app builds a project, it displays the build progress. When the build is complete, the app provides details about the build on the **Target Build Log** tab.

If the code generation report is enabled or build errors occur, the app generates a report. The report provides detailed information about the most recent build, and provides a link to the report.

To view the report, click the **View report** link. The report provides links to your MATLAB code and generated C/C++ files and compile-time type information for the variables in your MATLAB code. If build errors occur, the report lists errors and warnings.

See Also

- “Configure Build Settings” on page 20-26

Related Examples

- “Configure Build Settings” on page 20-26
- “C Code Generation Using the MATLAB Coder App”

Generate MEX Functions at the Command Line

Command-line Workflow for Generating MEX Functions

Step	Action	Details
1	Install prerequisite products.	"Installing Prerequisite Products"
2	Set up your file infrastructure.	"Paths and File Infrastructure Setup" on page 20-82
3	Fix errors detected by the code analyzer.	"Fixing Errors Detected at Design Time" on page 18-4
4	Specify build configuration parameters.	"Specify Build Configuration Parameters" on page 20-32
5	Specify properties of primary function inputs.	"Specify Properties of Entry-Point Function Inputs" on page 20-46
6	Generate the MEX function using `codegen` with suitable command-line options.	`codegen`

Generate a MEX Function at the Command Line

In this example, you use the `codegen` function to generate a MEX function from a MATLAB file that adds two inputs. You use the `codegen -args` option to specify that both inputs are `int16`.

1 In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

2 Generate a platform-specific MEX function in the current folder. At the command line, specify that the two input parameters are `int16` using the `-args` option. By default, if you do not use the `-args` option, `codegen` treats inputs as real, scalar doubles.

```
codegen mcadd -args {int16(0), int16(0)}
```

`codegen` generates a MEX function, `mcadd_mex`, in the current folder. `codegen` also generates other supporting files in a subfolder called `codegen/mex/mcadd.codegen` uses the name of the MATLAB function as the root name for the generated files and creates a platform-specific extension for the MEX file, as described in “Naming Conventions” on page 20-83.

Related Examples

- “Specify Properties of Entry-Point Function Inputs” on page 20-46
- “MEX Function Generation at the Command Line”

Fix Errors Detected at Code Generation Time

When the code generator detects errors or warnings, it automatically generates an error report. The error report describes the issues and provides links to the MATLAB code with errors.

To fix the errors, modify your MATLAB code to use only those MATLAB features that are supported for code generation. For more information, see “Programming Considerations for Code Generation”. Choose a debugging strategy for detecting and correcting code generation errors in your MATLAB code. For more information, see “Debugging Strategies” on page 18-27.

When code generation is complete, the software generates a MEX function that you can use to test your implementation in MATLAB.

If your MATLAB code calls functions on the MATLAB path, unless the code generator determines that these functions should be extrinsic or you declare them to be extrinsic, it attempts to compile these functions. See “Resolution of Function Calls for Code Generation” on page 13-2. To get detailed diagnostics, add the `%#codegen` directive to each external function that you want `codegen` to compile.

See Also

- “Code Generation Reports” on page 21-9
- “Why Test MEX Functions in MATLAB?” on page 19-2
- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Debugging Strategies” on page 18-27
- “Declaring MATLAB Functions as Extrinsic Functions” on page 13-10

Design Considerations When Writing MATLAB Code for Code Generation

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

- Data types

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

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

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

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

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

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

  To improve the speed of the generated code:

  - Choose a suitable C or /C++ compiler. The default compiler that MathWorks supplies with MATLAB for Windows 64-bit platforms is not a good compiler for performance.

- Consider disabling run-time checks.

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

See Also

- “Programming Considerations for Code Generation”
- “Data Definition”
- “Code Generation for Variable-Size Arrays” on page 6-2
- “Control Run-Time Checks” on page 25-15

Running MEX Functions

When you call a MEX function, pass it the same inputs that you use for the original MATLAB algorithm. Do not pass `coder.Constant` or any of the `coder.Type` classes to a MEX function. You can use these classes with only the `codegen` function.

To run a MEX function generated by MATLAB Coder, you must have licenses for all the toolboxes that the MEX function requires. For example, if you generate a MEX function from a MATLAB algorithm that uses a Computer Vision System Toolbox function or System object, to run the MEX function, you must have a Computer Vision System Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

Debugging MEX Functions

To debug your MEX functions, use the `disp` function to inspect the contents of your MEX function variables. You cannot use `save` to debug MEX function variables because code generation does not support it. Code generation does not support declaration of `save` as extrinsic.

Debugging Strategies

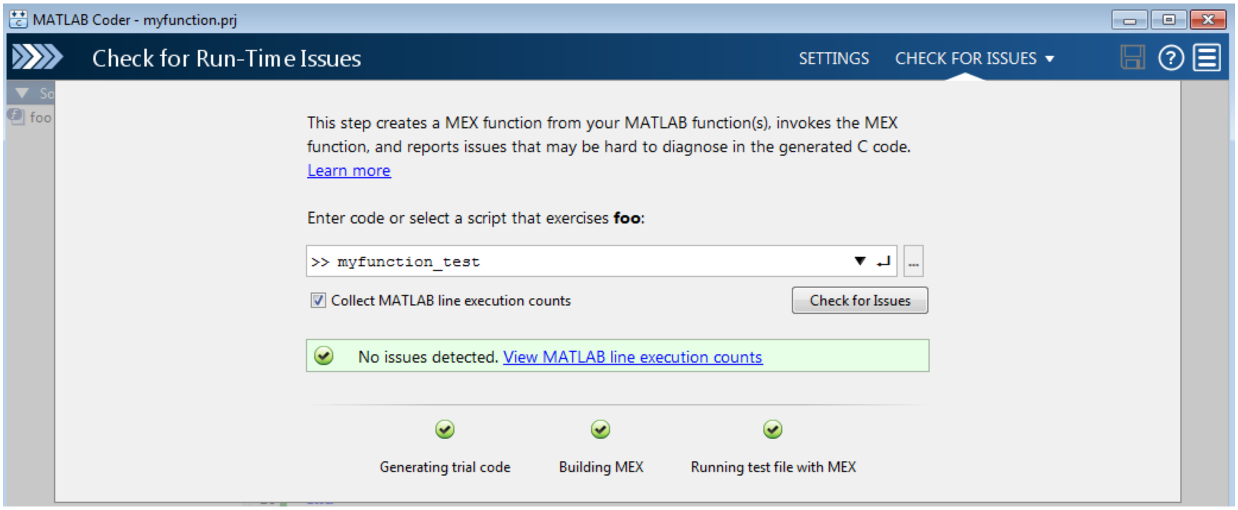
Before you perform code verification, choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other's functions. The following table describes two general strategies, each of which has advantages and disadvantages.

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	**1** Verify that your lowest-level (leaf) functions are compliant. **2** Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function.	• Efficient • Unlikely to cause errors • Easy to isolate code generation syntax violations	Requires application tests that work from the bottom up
Top-down verification	**1** Declare functions called by the top-level function to be extrinsic so that MATLAB Coder does not compile them. See "Declaring MATLAB Functions as Extrinsic Functions" on page 13-10. **2** Verify that your top-level function is compliant. **3** Work your way down the function hierarchy incrementally by removing extrinsic declarations one by one to compile and verify each function, ending with the leaf functions.	You retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: • Extrinsic declarations • Additional assignment statements as required to convert opaque values returned by extrinsic functions to nonopaque values (see "Working with mxArrays" on page 13-15).

Collect and View Line Execution Counts for Your MATLAB Code

When you perform the **Check for Run-Time Issues** step in the MATLAB Coder app, you must provide a test that calls your entry-point functions with representative data. The **Check for Run-Time Issues** step generates a MEX function from your MATLAB functions and runs the test, replacing calls to the MATLAB functions with calls to the MEX function. When running the MEX function, the app counts executions of the MEX code that corresponds to a line of MATLAB code. These line execution counts help you to see how well your test exercises your MATLAB code. You can identify dead code and sections of code that require further testing.

To see the line execution counts, after the **Check for Run-Time Issues** step finishes the test, click **View MATLAB line execution counts**.


MATLAB Coder - myfunction.prj
Check for Run-Time Issues
SETTINGS
CHECK FOR ISSUES ▾
So
foo
Check for Issues
Generating trial code
Building MEX
Running test file with MEX

This step creates a MEX function from your MATLAB function(s), invokes the MEX function, and reports issues that may be hard to diagnose in the generated C code.
Learn more

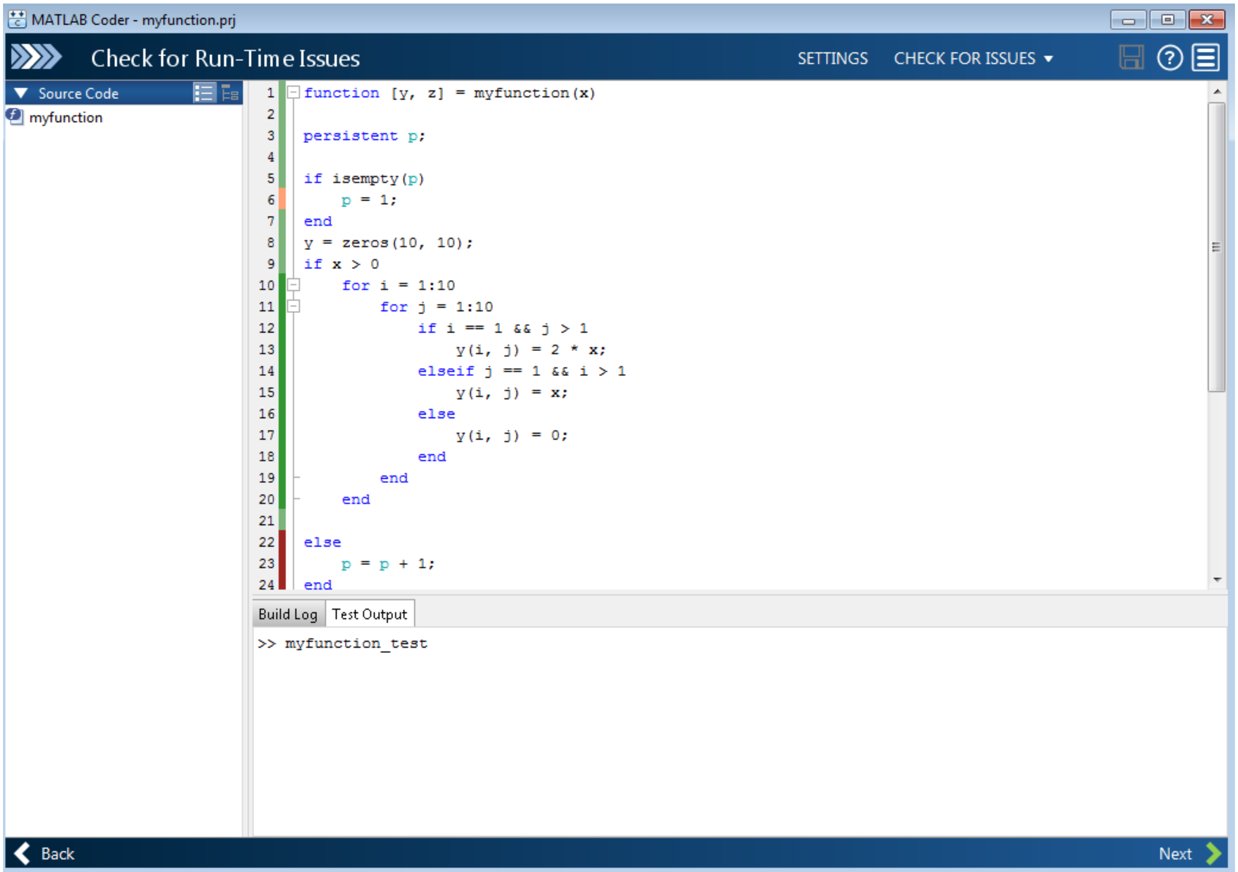
Enter code or select a script that exercises **foo**:

```
>> myfunction_test
```

☑ Collect MATLAB line execution counts

✓ No issues detected. View MATLAB line execution counts

In the app editor, the app displays a color-coded bar to the left of your MATLAB code.



```
function [y, z] = myfunction(x)

persistent p;

if isempty(p)
    p = 1;
end
y = zeros(10, 10);
if x > 0
    for i = 1:10
        for j = 1:10
            if i == 1 && j > 1
                y(i, j) = 2 * x;
            elseif j == 1 && i > 1
                y(i, j) = x;
            else
                y(i, j) = 0;
            end
        end
    end

else
    p = p + 1;
end
```

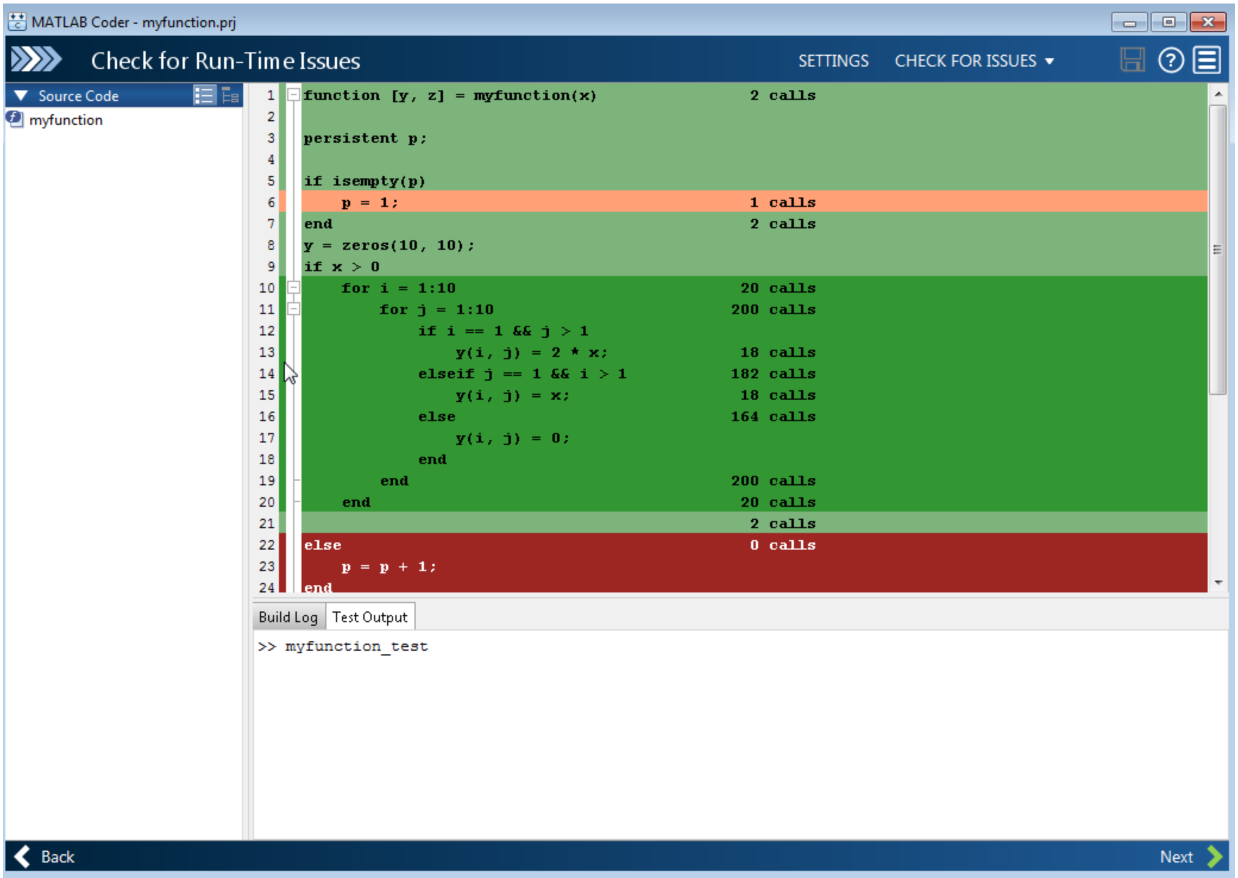
```
>> myfunction_test
```

This table describes the color coding.

Color	Indicates
Green	One of the following situations: • The entry-point function executes multiple times and the code executes more than one time. • The entry-point function executes one time and the code executes one time. Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.

Color	Indicates
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you place your cursor over the bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that the section executes.



```
function [y, z] = myfunction(x)                         2 calls

persistent p;

if isempty(p)
    p = 1;                                              1 calls
end                                                     2 calls
y = zeros(10, 10);
if x > 0
    for i = 1:10                                       20 calls
        for j = 1:10                                  200 calls
            if i == 1 && j > 1
                y(i, j) = 2 * x;                       18 calls
            elseif j == 1 && i > 1                    182 calls
                y(i, j) = x;                           18 calls
            else                                      164 calls
                y(i, j) = 0;
            end
        end                                           200 calls
    end                                                20 calls
                                                        2 calls
else                                                    0 calls
    p = p + 1;
end
```

```
>> myfunction_test
```

Collection of line execution counts is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off line execution counts can speed up the **Check for Run-Time Issues** step. To turn off collection of line executions counts, in the **Check for Run-Time Issues** dialog box, clear the **Collect MATLAB line execution counts** check box.

Related Examples

- “Check for Run-Time Issues by Using the App” on page 19-6

More About

- “Why Test MEX Functions in MATLAB?” on page 19-2

Testing MEX Functions in MATLAB

Why Test MEX Functions in MATLAB?

Before generating C/C++ code for your MATLAB code, it is a best practice to test the MEX function to verify that it provides the same functionality as the original MATLAB code. To do this testing, run the MEX function using the same inputs as you used to run the original MATLAB code and compare the results. For more information about how to test a MEX function using the MATLAB Coder app, see “Check for Run-Time Issues by Using the App” on page 19-6 and “Verify MEX Functions in the MATLAB Coder App” on page 19-8. For more information about how to test a MEX function at the command line, see “Verify MEX Functions at the Command Line” on page 19-9.

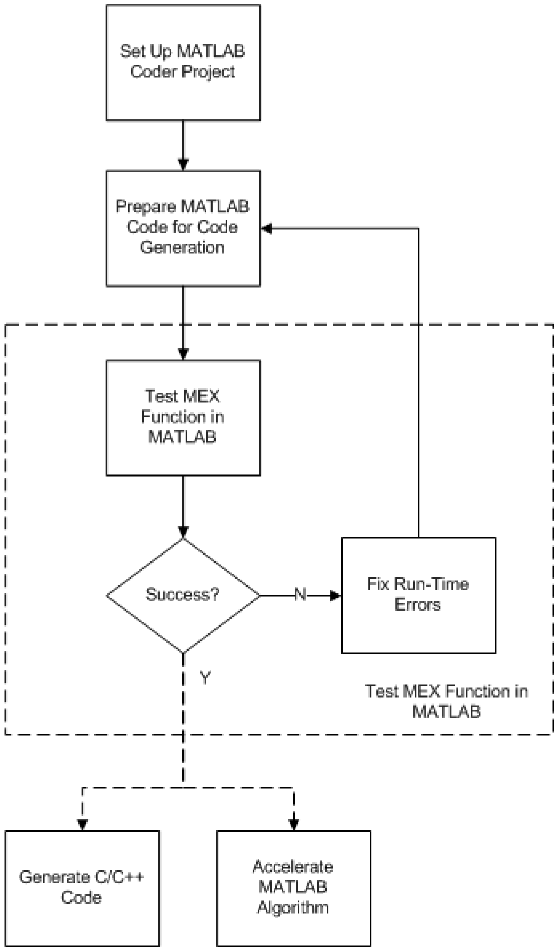
Running the MEX function in MATLAB before generating code enables you to detect and fix run-time errors that are much harder to diagnose in the generated code. If you encounter run-time errors in your MATLAB functions, fix them before generating code. See “Fix Errors Detected at Code Generation Time” on page 18-23 and “Debug Run-Time Errors” on page 19-10.

When you run your MEX function in MATLAB, by default, the following run-time checks execute:

- Memory integrity checks. These checks perform array bounds checking, dimension checking, and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution and provides a diagnostic message.
- Responsiveness checks in code generated for MATLAB functions. These checks enable periodic checks for **Ctrl+C** breaks in code generated for MATLAB functions, allowing you to terminate execution with **Ctrl+C**.

For more information, see “Control Run-Time Checks” on page 25-15.

Workflow for Testing MEX Functions in MATLAB


Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Test MEX Function in MATLAB
Success?
N
Fix Run-Time Errors
Y
Test MEX Function in MATLAB
Generate C/C++ Code
Accelerate MATLAB Algorithm

See Also

- "Set Up a MATLAB Coder Project" on page 17-2
- "Workflow for Preparing MATLAB Code for Code Generation" on page 18-2

- “Why Test MEX Functions in MATLAB?” on page 19-2
- “Debug Run-Time Errors” on page 19-10
- “C/C++ Code Generation” on page 20-4
- “Accelerate MATLAB Algorithms” on page 25-13

Running MEX Functions

When you call a MEX function, pass it the same inputs that you use for the original MATLAB algorithm. Do not pass `coder.Constant` or any of the `coder.Type` classes to a MEX function. You can use these classes with only the `codegen` function.

To run a MEX function generated by MATLAB Coder, you must have licenses for all the toolboxes that the MEX function requires. For example, if you generate a MEX function from a MATLAB algorithm that uses a Computer Vision System Toolbox function or System object, to run the MEX function, you must have a Computer Vision System Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

Debugging MEX Functions

To debug your MEX functions, use the `disp` function to inspect the contents of your MEX function variables. You cannot use `save` to debug MEX function variables because code generation does not support it. Code generation does not support declaration of `save` as extrinsic.

Check for Run-Time Issues by Using the App

Before you generate standalone C/C++ code for your MATLAB code, it is a best practice to generate a MEX function from your entry-point functions. Running the MEX function helps you to detect and fix run-time errors that are much harder to diagnose in the generated code. It also helps you to verify that the MEX provides the same functionality as the original MATLAB code.

In the MATLAB Coder app, to generate and run the MEX function for your MATLAB code, perform the **Check for Run-Time Issues** step.

1. Write a function or script that calls your entry-point functions. You can use the same test file (or files) that you use to automatically define input types in the **Define Input Types** step.
2. Complete the **Select Source Files** and **Define Input Types** steps. On the **Define Input Types** page, click **Next** to go to **Check for Run-Time Issues** step.
3. Specify the test file. In the previous step, if you automatically generated the input types, the app populates the dialog box with that test file. Instead of a test file, you can enter code that calls your entry-point functions. However, it is a best practice to provide a test file.
4. Click **Check for Issues**. The app generates a MEX function from your MATLAB function. It runs the test that you specified, substituting calls to your MATLAB entry-point functions with calls to the generated MEX function. The app reports MEX generation or build errors on the **Build Errors** tab. The app reports MEX run-time errors on the **Test Output** tab.
5. If the app reports errors, to edit the MATLAB code, click **View errors**.
6. After you fix issues, to rerun the test, click **Check for Issues**.

When the app runs the MEX function, it counts executions of the MEX code that corresponds to a line of MATLAB code. If the app does not detect issues, you can view these line execution counts. The line execution counts help you to see how well your test exercises your MATLAB code. You can identify dead code and sections of code that require further testing. See “Collect and View Line Execution Counts for Your MATLAB Code” on page 18-28.

By default, to speed up generation of the MEX function, the app tries to use just-in-time (JIT) compilation. JIT compilation is incompatible with certain code generation features and options such as custom code and use of the OpenMP library. If the app cannot use JIT compilation, it generates a C/C++ MEX function instead. If your code uses `parfor` and the **Enable OpenMP library if possible** setting is `Yes`, the app uses JIT compilation and treats the `parfor`-loops as `for`-loops. If you want the **Check for Run-Time Issues** step to run `for`-loops in parallel, disable JIT compilation:

1. On the **Check for Run-Time Issues** page, click **Settings**.
2. On the **All Settings** tab, set **Use JIT compilation in Check for Run-Time Issues** to `No`.

Related Examples

- "C Code Generation Using the MATLAB Coder App"
- "Fix Errors Detected at Code Generation Time" on page 18-23
- "Collect and View Line Execution Counts for Your MATLAB Code" on page 18-28
- "Control Run-Time Checks" on page 25-15

More About

- "Why Test MEX Functions in MATLAB?" on page 19-2

Verify MEX Functions in the MATLAB Coder App

In the MATLAB Coder app, after you generate a MEX function, you can verify that the generated MEX function has the same functionality as the original MATLAB entry-point function. Provide a test file that calls the original MATLAB entry-point function. The test file can be a MATLAB function or scriptThe test file must be in the same folder as the original entry-point function.

1. On the **Generate Code** page, click **Verify Code**.
2. Type or select the test file name.
3. To run the test file without replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **MATLAB code**. Click **Run Generated Code**.
4. To run the test file, replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **Generated code**. Click **Run Generated Code**.
5. Compare the results of running the original MATLAB function with the results of running the MEX function.

See Also

`codegen` | `coder.runTest`

More About

- "Why Test MEX Functions in MATLAB?" on page 19-2
- "Verify MEX Functions at the Command Line" on page 19-9
- "Unit Test Generated Code with MATLAB Coder" on page 21-33

Verify MEX Functions at the Command Line

If you have a test file that calls your original MATLAB function, you can use `coder.runTest` to verify the MEX function at the command line. `coder.runTest` runs the test file replacing calls to the original MATLAB function with calls to the generated MEX function. For example, here is a call to `coder.runTest` for the test file `myfunction_test` and the function `myfunction`

```
coder.runTest('myfunction_test', 'myfunction')
```

If errors occur during the run with `coder.runTest`, call stack information is available for debugging.

Alternatively, you can use the `codegen -test` option.

```
codegen myfunction -test 'myfunction_test'
```

The test file can be a MATLAB function, script, or class-based unit test.

See Also

`codegen` | `coder.runTest`

More About

- "Why Test MEX Functions in MATLAB?" on page 19-2
- "Check for Run-Time Issues by Using the App" on page 19-6
- "Unit Test Generated Code with MATLAB Coder" on page 21-33

Debug Run-Time Errors

In this section...
"Viewing Errors in the Run-Time Stack" on page 19-10
"Handling Run-Time Errors" on page 19-12

If you encounter run-time errors in your MATLAB functions, the run-time stack appears in the MATLAB command window. Use the error message and stack information to learn more about the source of the error, and then either fix the issue or add error-handling code. For more information, see "Viewing Errors in the Run-Time Stack" on page 19-10"Handling Run-Time Errors" on page 19-12.

Viewing Errors in the Run-Time Stack

About the Run-Time Stack

The run-time stack is enabled by default for MEX code generation from MATLAB. To learn more about the source of the error, use the error message and the following stack information:

- The name of the function that generated the error
- The line number of the attempted operation
- The sequence of function calls that led up to the execution of the function and the line at which each of these function calls occurred

Example Run-Time Stack Trace

This example shows the run-time stack trace for MEX function `mlstack_mex`:

```
mlstack_mex(-1)

Index exceeds matrix dimensions.  Index
value -1 exceeds valid range [1-4] of
array x.

Error in mlstack>mayfail (line 31)
y = x(u);

Error in mlstack>subfcn1 (line 5)
switch (mayfail(u))

Error in mlstack (line 2)
y = subfcn1(u);
```

The stack trace provides the following information:

- The type of error.

  ```
  ??? Index exceeds matrix dimensions.
  Index value -1 exceeds valid range [1-4] of array x.
  ```

- Where the error occurred.

  ```
  Error in ==>mlstack>mayfail at 31
  y = x(u);
  ```

- The function call sequence prior to the failure.

  ```
  Error in ==> mlstack>subfcn1 at 5
  switch (mayfail(u))

  Error in ==> mlstack at 2
  y = subfcn1(u);
  ```

When to Use the Run-Time Stack

To help you find the source of run-time errors, the run-time stack is useful during debugging. However, when the stack is enabled, the generated code contains instructions for maintaining the run-time stack, which might slow the run time. Consider disabling the run-time stack for faster run time.

Disable the Run-Time Stack

You can disable the run-time stack by disabling the memory integrity checks as described in “How to Disable Run-Time Checks” on page 25-16.

Caution: Before disabling the memory integrity checks, verify that all array bounds and dimension checking is unnecessary.

Handling Run-Time Errors

The code generator propagates error IDs. If you throw an error or warning in your MATLAB code, use the `try-catch` statement in your test bench code to examine the error information and attempt to recover, or clean up and abort. For example, for the function in “Example Run-Time Stack Trace” on page 19-11, create a test script containing:

```
try
    mlstack_mex(u)
catch
    % Add your error handling code here
end
```

For more information, see “The try/catch Statement” (MATLAB).

Using MEX Functions That MATLAB Coder Generates

When you specify MEX for the output (build) type, MATLAB Coder generates a binary MATLAB executable (MEX) version of your MATLAB function. You can call the MEX function from MATLAB. See “Call MEX File Functions” (MATLAB).

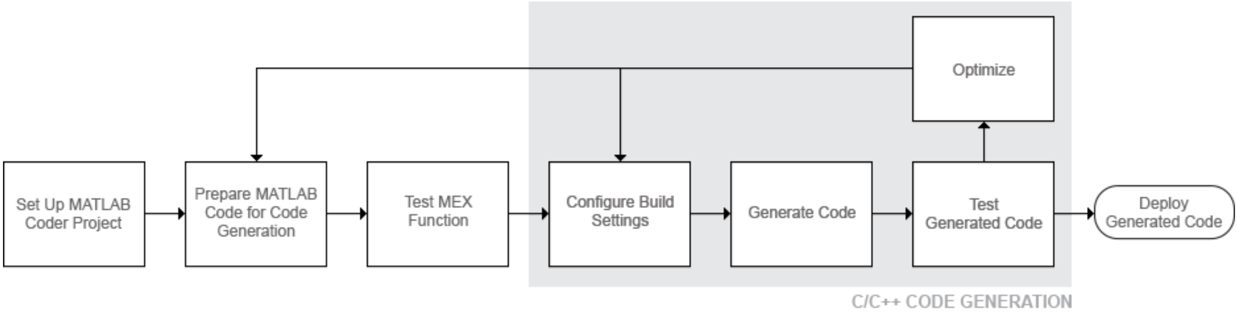
How you use the MEX function depends on your goal.

Goal	See
Accelerate your MATLAB function.	“MATLAB Algorithm Acceleration”
Test generated function for functionality and run-time issues.	“Why Test MEX Functions in MATLAB?” on page 19-2
Debug your MEX function.	“Debug Run-Time Errors” on page 19-10

20

Generating C/C++ Code from MATLAB Code

Code Generation Workflow



See Also

- “Set Up a MATLAB Coder Project” on page 17-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 18-2
- “Workflow for Testing MEX Functions in MATLAB” on page 19-3
- “Configure Build Settings” on page 20-26
- “C/C++ Code Generation” on page 20-4

C/C++ Code Generation

Using MATLAB Coder, you can generate platform-specific MEX functions, C/C++ static and dynamic libraries, and C/C++ executable programs. If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

To learn how to generate...	See...
C/C++ static libraries from your MATLAB code	"Generating C/C++ Static Libraries from MATLAB Code" on page 20-5
C/C++ dynamic libraries from your MATLAB code	"Generating C/C++ Dynamically Linked Libraries from MATLAB Code" on page 20-9
C/C++ executables from your MATLAB code	"Generating Standalone C/C++ Executables from MATLAB Code" on page 20-14
MEX functions from your MATLAB code	"Generate MEX Functions by Using the MATLAB Coder App" on page 18-16

If errors occur, MATLAB Coder does not generate code, but produces an error report and provides a link to this report. For more information, see "Code Generation Reports" on page 21-9.

Specify Custom Files to Build

In addition to your MATLAB file, you can specify the following types of custom *files* to include in the build for standalone C/C++ code generation.

File Extension	Description
`.c`	Custom C file
`.cpp`	Custom C++ file
`.h`	Custom header file
`.o` , `.obj`	Custom object file
`.a` , `.lib`, `.so`, `.dylib`	Library
`.tmf`	Template makefile for custom MATLAB Coder builds

Generating C/C++ Static Libraries from MATLAB Code

In this section...

Generate a C Static Library Using the MATLAB Coder App

This example shows how to generate a C static library from MATLAB code using the MATLAB Coder app.

In this example, you create a MATLAB function that adds two numbers. You use the app to create a MATLAB Coder project and generate a C static library for the MATLAB code.

Create the Entry-Point Function

In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

Create the Test File

In the same local writable folder, create a MATLAB file, `mcadd_test.m`, that calls `mcadd` with example inputs. The example inputs are scalars with type `int16`.

```
function y = mcadd_test
y = mcadd(int16(2), int16(3));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

1 On the **Select Source Files** page, type or select the name of the entry-point function `mcadd`.

The app creates a project with the default name `mcadd.prj` in the current folder.

2. Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify the test file `mcadd_test.m` that MATLAB Coder can use to automatically define types for `u` and `v`:

1. Enter or select the test file `mcadd_test.m`.
2. Click **Autodefine Input Types**.

   The test file, `mcadd_test.m`, calls the entry-point function, `mcadd` with the example input types. MATLAB Coder infers that inputs `u` and `v` are `int16(1x1)`.
3. Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow ▼.

   The app populates the test file field with `mcadd_test`, the test file that you used to define the input types.
2. Click **Check for Issues**.

   The app generates a MEX function. It runs the test file replacing calls to `mcadd` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

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

Generate C Code

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. In the **Generate** dialog box, set **Build type** to `Static Library (.lib)` and **Language** to C. Use the default values for the other project build configuration settings.
3. Click **Generate**.

   The app indicates that code generation succeeded. It displays the source MATLAB files and generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Target Build Log** tab, it displays the build log, including compiler warnings and errors. By default, in the code window, the app displays the C source code file, `mcadd.c`. To view a different file, in the **Source Code** or **Output Files** pane, click the file name.

   MATLAB Coder generates a standalone C static library `mcadd` in the `codegen\lib\mcadd` folder. It generates the minimal set of `#include` statements for header files required by the selected code replacement library.
4. To view the code generation report, click **View Report**.
5. Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to generated output.

Generate a C Static Library at the Command Line

This example shows how to generate a C static library from MATLAB code at the command line using the `codegen` function.

1. In a local writable folder, create a MATLAB file, `mcadd.m`, that contains:

   ```
   function y = mcadd(u,v) %#codegen
   ```

```
y = u + v;
```

2 Using the `config:lib` option, generate C library files. Using the `-args` option, specify that the first input is a `1-by-4` vector of unsigned 16-bit integers and that the second input is a double-precision scalar.

```
codegen -config:lib mcadd -args {zeros(1,4,'uint16'),0}
```

MATLAB Coder generates a C static library with the default name, `mcadd`, and supporting files in the default folder, `codegen/lib/mcadd`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Generating C/C++ Dynamically Linked Libraries from MATLAB Code

In this section...

Dynamic Libraries Generated by MATLAB Coder

By default, when MATLAB Coder generates a dynamic library (DLL):

- The DLL is suitable for the platform that you are working on.
- The DLL uses the release version of the C run-time library.
- The DLL linkage conforms to the target language, by default, C. If you set the target language to C++, the linkage conforms to C++.
- When the target language is C, the generated header files explicitly declare the exported functions to be `extern "C"` to simplify integration of the DLL into C++ applications.
- When an executable that uses the DLL runs, the DLL must be on the system path so that the executable can access it.

If you generate a DLL that uses dynamically allocated variable-size data, MATLAB Coder provides exported utility functions to interact with this data in the generated code. For more information, see “Utility Functions for Creating emxArray Data Structures” on page 6-18.

Generate a C Dynamically Linked Library Using the MATLAB Coder App

This example shows how to generate a C DLL from MATLAB code using the MATLAB Coder app.

Create the Entry-Point Functions

Write two MATLAB functions, `ep1` and `ep2`. `ep1` takes one input, a single scalar. `ep2` takes two inputs that are double scalars. In a local writable folder:

1. Create a MATLAB file, `ep1.m`, that contains:

   ```
   function y = ep1(u) %#codegen
   y = u;
   ```

2. Create a MATLAB file, `ep2.m`, that contains:

   ```
   function y = ep2(u, v) %#codegen
   y = u + v;
   ```

Create the Test File

In the folder that contains `ep1.m` and `ep2.m`, create a MATLAB file, `ep_test.m`, that calls `ep1` and `ep2` with example inputs.

```
function [y, y1] = ep_test
y = ep1(single(2));
y1 = ep2(double(3), double(4));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

1. On the **Select Source Files** page, type or select the name of the entry-point function `ep1`.

   The app creates a project with the default name `ep1.prj` in the current folder.
2. To add `ep2` to the list of entry-point functions, click **Add Entry-Point Function**. Type or select the name of the entry-point function `ep2`.
3. Click **Next** to go to the **Define Input Types** step. The app analyzes the functions for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify a test file that MATLAB Coder can use to automatically define types:

1. Enter or select the test file `ep_test.m`.
2. Click **Autodefine Input Types**.

   The test file, `eps_test.m`, calls the entry-point functions `ep1` and `ep2` with the example input types. MATLAB Coder infers that for `ep1`, input `u` is `single(1x1)`. For `ep2`, `u` and `v` are `double(1x1)`.
3. Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow ▼.

   The app populates the test file field with `ep_test`, the test file that you used to define the input types.
2. Click **Check for Issues**.

   The app generates a MEX function named `ep1_mex` for `ep1` and `ep2`. It runs the test file `ep_test` replacing calls to `ep1` and `ep2` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.
3. Click **Next** to go to the **Generate Code** step.

Generate C Code

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. In the **Generate** dialog box, set **Build type** to `Dynamic Library` and **Language** to C. Use the default values for the other project build configuration settings.

3 Click **Generate**.

   The app indicates that code generation succeeded. It displays the source MATLAB files and generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Target Build Log** tab, it displays the build log, including compiler warnings and errors. By default, in the code window, the app displays the C source code file, `ep1.c`. To view a different file, in the **Source Code** or **Output Files** pane, click the file name.

   On Microsoft® Windows systems, MATLAB Coder generates a C dynamic library, `ep1.dll`, and supporting files, in the default folder, `codegen\dll\ep1`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library. On Linux®, it generates a shared object (.so) file. On Mac, it generates a dynamic library (.dylib) file. The DLL linkage conforms to the target language, in this example, C. If you set the target language to C++, the linkage conforms to C++. MATLAB Coder generates a standalone C static library `mcadd` in the `codegen\lib\mcadd` folder.

4 To view the code generation report, click **View Report**.

5 Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to generated output.

Generate a C Dynamic Library at the Command Line

This example shows how to generate a C dynamic library from MATLAB code at the command line using the `codegen` function.

1 Write two MATLAB functions, `ep1` takes one input, a single scalar, and `ep2` takes two inputs, both double scalars. In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

   In the same local writable folder, create a MATLAB file, `ep2.m`, that contains:

```
function y = ep2(u, v) %#codegen
y = u + v;
```

2 Generate the C dynamic library.

```
codegen -config:dll ep1 -args single(0) ep2 -args {0,0}
```

On Microsoft Windows systems, `codegen` generates a C dynamic library, `ep1.dll`, and supporting files, in the default folder, `codegen/dll/ep1`. It generates the minimal set of `#include` statements for header files required by the selected code replacement library. On Linux, it generates a shared object (.so) file. On Mac, it generates a dynamic library (.dylib) file. The DLL linkage conforms to the target language, in this example, C. If you set the target language to C++, the linkage conforms to C++.

Note: The default target language is C. To change the target language to C++, see "Specify a Language for Code Generation" on page 20-28.

Generating Standalone C/C++ Executables from MATLAB Code

In this section...

Generate a C Executable Using the MATLAB Coder App

This example shows how to generate a C executable from MATLAB code using the MATLAB Coder app. In this example, you generate an executable for a MATLAB function that generates a random scalar value. Using the app, you:

1. Generate a C `main` function that calls the generated library function.
2. Copy and modify the generated `main.c` and `main.h`.
3. Modify the project settings so that the app can find the modified `main.c` and `main.h`.
4. Generate the executable.

Create the Entry-Point Function

In a local writable folder, create a MATLAB function, `coderand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1):

```
function r = coderand() %#codegen
r = rand();
```

Create the Test File

In the same local writable folder, create a MATLAB file, `coderand_test.m`, that calls `coderand`.

```
function y = coderand_test()
y = coderand();
```

Open the MATLAB Coder app

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

1. On the **Select Source Files** page, type or select the name of the entry-point function `coderand`.

   The app creates a project with the default name `coderand.prj` in the current folder.
2. Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

In this example, the function `coderand` does not have inputs.

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

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow ▼.

   Select or enter the test file `coderand_test`.
2. Click **Check for Issues**.

   The app generates a MEX function for `coderand`. It runs the test file replacing calls to `coderand` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages.

Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

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

Generate a C `main` Function

When you generate an executable, you must provide a C/C++ function. By default, when you generate C/C++ source code, static libraries, dynamically linked libraries, or executables, MATLAB Coder generates a `main` function. This generated main function is a template that you modify for your application. See “Incorporate Generated Code Using an Example Main Function” on page 24-20. After you copy and modify the generated main function, you can use it for generation of the C/C++ executable. Alternatively, you can write your own main function.

Before you generate the executable for `coderand`, generate a `main` function that calls `coderand`.

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. In the **Generate** dialog box, set **Build type** to `Source Code` and **Language** to C. Use the default values for the other project build configuration settings.
3. Click **More Settings**.
4. On the **All Settings** tab, under **Advanced**, verify that **Generate example main** is set to `Generate, but do not compile, an example main function`. Click **Close**.
5. Click **Generate**.

   MATLAB Coder generates a `main.c` file and a `main.h` file. The app indicates that code generation succeeded.
6. Click **Next** to open the **Finish Workflow** page.

   On the **Finish Workflow** page, under **Generated Output**, you see that `main.c` is in the subfolder `coderand\codegen\lib\coderand\examples`.

Copy the Generated Example Main Files

Because subsequent code generation can overwrite the generated example files, before you modify these files, copy them to a writable folder outside of the `codegen` folder. For this example, copy `main.c` and `main.h` from the subfolder `coderand\codegen\lib\coderand\examples` to a writable folder, for example, `c:\myfiles`.

Modify the Generated Example Main Files

1 In the folder that contains a copy of the example main files, open `main.c`.

Generated main.c

```
/*************************************************************************/
/* This automatically generated example C main file shows how to call    */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly.     */
/* Instead, make a copy of this file, modify it, and integrate it into   */
/* your development environment.                                         */
/*                                                                       */
/* This file initializes entry-point function arguments to a default     */
/* size and value before calling the entry-point functions. It does      */
/* not store or use any values returned from the entry-point functions.  */
/* If necessary, it does pre-allocate memory for returned values.        */
/* You can use this file as a starting point for a main function that    */
/* you can deploy in your application.                                   */
/*                                                                       */
/* After you copy the file, and before you deploy it, you must make the  */
/* following changes:                                                    */
/* * For variable-size function arguments, change the example sizes to   */
/* the sizes that your application requires.                             */
/* * Change the example values of function arguments to the values that  */
/* your application requires.                                            */
/* * If the entry-point functions return values, store these values or   */
/* otherwise use them as required by your application.                   */
/*                                                                       */
/*************************************************************************/
/* Include Files */
#include "rt_nonfinite.h"
#include "coderand.h"
#include "main.h"
#include "coderand_terminate.h"
#include "coderand_initialize.h"

/* Function Declarations */
static void main_coderand(void);

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
static void main_coderand(void)
{
  double r;

  /* Call the entry-point 'coderand'. */
  r = coderand();
}

/*
 * Arguments    : int argc
 *                const char * const argv[]
 * Return Type  : int
 */
int main(int argc, const char * const argv[])
{
  (void)argc;
  (void)argv;

  /* Initialize the application.
     You do not need to do this more than one time. */
  coderand_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_coderand();

  /* Terminate the application.
     You do not need to do this more than one time. */
  coderand_terminate();
  return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */
```

2 Modify `main.c` so that it prints the results of a `coderand` call:

- In `main_coderand`, delete the line

  ```
  double r;
  ```

- In `main_coderand`, replace

  ```
  r = coderand()
  ```
  with

  ```
  printf("coderand=%g\n", coderand());
  ```
- For this example, `main` does not have arguments. In `main`, delete the lines:

  ```
  (void)argc;
  (void)argv;
  ```

  Change the definition of `main` to

  ```
  int main()
  ```

Modified main.c

```
/* Include Files */
#include "rt_nonfinite.h"
#include "coderand.h"
#include "main.h"
#include "coderand_terminate.h"
#include "coderand_initialize.h"

/* Function Declarations */
static void main_coderand(void);

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
static void main_coderand(void)
{
    /* Call the entry-point 'coderand'. */
  printf("coderand=%g\n", coderand());
}

/*
 * Arguments    : int argc
 *                const char * const argv[]
 * Return Type  : int
 */
```

```
int main()
{

  /* Initialize the application.
     You do not need to do this more than one time. */
  coderand_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_coderand();

  /* Terminate the application.
     You do not need to do this more than one time. */
  coderand_terminate();
  return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */
```

3 Open `main.h`

Generated main.h

```
*************************************************************************/
/* This automatically generated example C main file shows how to call    */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly.     */
/* Instead, make a copy of this file, modify it, and integrate it into   */
/* your development environment.                                         */
/*                                                                       */
/* This file initializes entry-point function arguments to a default     */
/* size and value before calling the entry-point functions. It does      */
/* not store or use any values returned from the entry-point functions.  */
/* If necessary, it does pre-allocate memory for returned values.        */
/* You can use this file as a starting point for a main function that    */
/* you can deploy in your application.                                   */
/*                                                                       */
/* After you copy the file, and before you deploy it, you must make the  */
/* following changes:                                                    */
/* * For variable-size function arguments, change the example sizes to   */
/* the sizes that your application requires.                             */
/* * Change the example values of function arguments to the values that  */
/* your application requires.                                            */
/* * If the entry-point functions return values, store these values or   */
/* otherwise use them as required by your application.                   */
/*                                                                       */
/*************************************************************************/
#ifndef __MAIN_H__
#define __MAIN_H__

/* Include Files */

#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include "rtwtypes.h"
#include "coderand_types.h"

/* Function Declarations */
extern int main(int argc, const char * const argv[]);

#endif

/*
 * File trailer for main.h
 *
 * [EOF]
 */
```

4 Modify `main.h`:

- Add `stdio` to the include files:

  ```
  #include <stdio.h>
  ```

- Change the declaration of main to

  ```
  extern int main()
  ```

Modified main.h

```
#ifndef __MAIN_H__
#define __MAIN_H__

/* Include Files */
```

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include "rtwtypes.h"
#include "coderand_types.h"

/* Function Declarations */
extern int main();

#endif

/*
 * File trailer for main.h
 *
 * [EOF]
 */
```

Generate the Executable

1. To open the **Generate Code** page, expand the workflow steps »» and click **Generate**
2. To open the **Generate** dialog box, click the **Generate** arrow ▼.
3. Set **Build type** to `Executable (.exe)`.
4. Click **More Settings**.
5. On the **Custom Code** tab, in **Additional source files**, enter `main.c`
6. On the **Custom Code** tab, in **Additional include directories**, enter the location of the modified `main.c` and `main.h` files. For example, `c:\myfiles`. Click **Close**.
7. To generate the executable, click **Generate**.

   The app indicates that code generation succeeded.
8. Click **Next** to go to the **Finish Workflow** step.
9. Under **Generated Output**, you can see the location of the generated executable `coderand.exe`.

Run the Executable

To run the executable in MATLAB on a Windows platform:

```
system('coderand')
```

Generate a C Executable at the Command Line

In this example, you create a MATLAB function that generates a random scalar value and a main C function that calls this MATLAB function. You then specify types for the function input parameters, specify the main function, and generate a C executable for the MATLAB code.

1 Write a MATLAB function, `coderand`, that generates a random scalar value from the standard uniform distribution on the open interval (0,1):

```
function r = coderand() %#codegen
r = rand();
```

2 Write a main C function, `c:\myfiles\main.c`, that calls `coderand`. For example:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_initialize.h"
#include "coderand_terminate.h"

int main()
{
    coderand_initialize();

    printf("coderand=%g\n", coderand());

    coderand_terminate();

    return 0;
}
```

Note: In this example, because the default file partitioning method is to generate one file for each MATLAB file, you include “`coderand_initialize.h`” and “`coderand_terminate.h`”. If your file partitioning method is set to generate one file for all functions, do **not** include “`coderand_initialize.h`” and “`coderand_terminate.h`”.

3 Configure your code generation parameters to include the main C function and then generate the C executable:

```
cfg = coder.config('exe');
cfg.CustomSource = 'main.c';
cfg.CustomInclude = 'c:\myfiles';
codegen -config cfg coderand
```

`codegen` generates a C executable, `coderand.exe`, in the current folder. It generates supporting files in the default folder, `codegen/exe/coderand`. `codegen` generates the minimal set of `#include` statements for header files required by the selected code replacement library.

Specifying main Functions for C/C++ Executables

When you generate an executable, you must provide a `main` function. For a C executable, provide a C file, `main.c`. For a C++ executable, provide a C++ file, `main.cpp`. Verify that the folder containing the main function has only one main file. Otherwise, `main.c` takes precedence over `main.cpp`, which causes an error when generating C++ code. You can specify the main file from the project settings dialog box, the command line, or the Code Generation dialog box.

By default, when you generate C/C++ source code, static libraries, dynamically linked libraries, or executables, MATLAB Coder generates a `main` function. This generated main function is a template that you modify for your application. See "Incorporate Generated Code Using an Example Main Function" on page 24-20. After you copy and modify the generated main function, you can use it for generation of the C/C++ executable. Alternatively, you can write your own main function.

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder generates an initialize function and a terminate function.

- If your file partitioning method is set to generate one file for each MATLAB file, you must include the initialize and terminate header functions in `main.c`. Otherwise, do not include them in `main.c`.
- You must call these functions along with the C/C++ function. For more information, see "Calling Initialize and Terminate Functions" on page 24-9.

Specify main Functions

Specifying main Functions Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Custom Code** tab, set:

   a. **Additional source files** to the name of the C/C++ source file that contains the `main` function. For example, `main.c`. For more information, see "Specifying main Functions for C/C++ Executables" on page 20-24.

   b. **Additional include directories** to the location of `main.c`. For example, `c:\myfiles`.

Specifying main Functions at the Command Line

Set the `CustomSource` and `CustomInclude` properties of the code generation configuration object (see "Working with Configuration Objects" on page 20-33). The `CustomInclude` property indicates the location of C/C++ files specified by `CustomSource`.

1. Create a configuration object for an executable:

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

2. Set the `CustomSource` property to the name of the C/C++ source file that contains the `main` function. (For more information, see "Specifying main Functions for C/C++ Executables" on page 20-24.) For example:

   ```
   cfg.CustomSource = 'main.c';
   ```

3. Set the `CustomInclude` property to the location of `main.c`. For example:

   ```
   cfg.CustomInclude = 'c:\myfiles';
   ```

4. Generate the C/C++ executable using the command-line options. For example, if `myFunction` takes one input parameter of type `double`:

   ```
   codegen -config cfg  myMFunction -args {0}
   ```

   MATLAB Coder compiles and links the main function with the C/C++ code that it generates from `myMFunction.m`.

Configure Build Settings

In this section...

Specify Build Type

Build Types

MATLAB Coder can generate code for the following output types:

- MEX function
- Standalone C/C++ code
- Standalone C/C++ code and compile it to a static library
- Standalone C/C++ code and compile it to a dynamically linked library
- Standalone C/C++ code and compile it to an executable

Note: When you generate an executable, you must provide a C/C++ file that contains the `main` function, as described in “Specifying main Functions for C/C++ Executables” on page 20-24.

Location of Generated Files

By default, MATLAB Coder generates files in output folders based on your output type. For more information, see “Generated Files and Locations” on page 20-132.

Note: Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

Specify the Build Type Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to one of the following.

   - `Source Code`
   - `MEX`
   - `Static Library`
   - `Dynamic Library`
   - `Executable`

If you select `Source Code`, MATLAB Coder does not invoke the make command or generate compiled object code. When you iterate between modifying MATLAB code and generating C/C++ code and you want to inspect the generated code, this option can save you time. This option is equivalent to `Static Library` with the **Generate code only** box selected.

Code generation uses a different set of configuration parameters for MEX functions than it uses for the other build types. . When you switch the output type between `MEX Function` and `Source`, `Static Library`, `Dynamic Library`, or `Executable`, verify these settings. For more information, see “Changing Output Type” on page 17-35.

Specifying the Build Type at the Command Line

Call `codegen` with the `-config` option. For example, suppose that you have a primary function `foo` that takes no input parameters. The following table shows how to specify different output types when compiling `foo`. If a primary function has input parameters, you must specify these inputs. For more information, see “Specify Properties of Entry-Point Function Inputs” on page 20-46.

Note: C is the default language for code generation with MATLAB Coder. To generate C++ code, see “Specify a Language for Code Generation” on page 20-28.

To Generate:	Use This Command:
MEX function using the default code generation options	`codegen foo`

To Generate:	Use This Command:

MEX function specifying code generation options	`cfg = coder.config('mex');`
	`% Set configuration parameters, for example,`
	`% enable a code generation report`
	`cfg.GenerateReport=true;`
	`% Call codegen, passing the configuration`
	`% object`
	`codegen -config cfg foo`

Standalone C/C++ code and compile it to a library using the default code generation options	`codegen -config:lib foo`

Standalone C/C++ code and compile it to a library specifying code generation options	`cfg = coder.config('lib');`
	`% Set configuration parameters, for example,`
	`% enable a code generation report`
	`cfg.GenerateReport=true;`
	`% Call codegen, passing the configuration`
	`% object`
	`codegen -config cfg foo`

Standalone C/C++ code and compile it to an executable using the default code generation options and specifying the `main.c` file at the command line	`codegen -config:exe main.c foo` **Note:** You must specify a `main` function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 20-24

Standalone C/C++ code and compile it to an executable specifying code generation options	`cfg = coder.config('exe');`
	`% Set configuration parameters, for example,`
	`%  specify main file`
	`cfg.CustomSource = 'main.c';`
	`cfg.CustomInclude = 'c:\myfiles';`
	`codegen -config cfg foo`
	Note: You must specify a `main` function for generating a C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 20-24

Specify a Language for Code Generation

- “Specify the Language Using the MATLAB Coder App” on page 20-29

- “Specifying the Language Using the Command-Line Interface” on page 20-29

MATLAB Coder can generate C or C++ libraries and executables. C is the default language. You can specify a language explicitly from the project settings dialog box or at the command line.

Specify the Language Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Language** to `C` or `C++`.

Note: If you specify `C++`, MATLAB Coder wraps the `C` code into `.cpp` files. You can use a `C++` compiler and interface with external `C++` applications. MATLAB Coder does not generate `C++` classes.

Specifying the Language Using the Command-Line Interface

1. Select a suitable compiler for your target language.
2. Create a configuration object for code generation. For example, for a library:

   ```
   cfg = coder.config('lib');
   ```
3. Set the `TargetLang` property to `'C'` or `'C++'`. For example:

   ```
   cfg.TargetLang = 'C++';
   ```

Note: If you specify `C++`, MATLAB Coder wraps the `C` code into `.cpp` files. You can then use a `C++` compiler and interface with external `C++` applications. MATLAB Coder does not generate `C++` classes.

See Also

- “Working with Configuration Objects” on page 20-33
- “Setting Up the C or C++ Compiler”

Specify Output File Name

Specify Output File Name Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. In the **Output file name** field, enter the file name.

Note: Do not put spaces in the file name.

By default, if the name of the first entry-point MATLAB file is *`fcn1`*, the output file name is:

- *`fcn1`* for C/C++ libraries and executables.
- *`fcn1_mex`* for MEX functions.

By default, MATLAB Coder generates files in the folder *`project_folder`*`/codegen/target/fcn1`:

- *`project_folder`* is your current project folder
- `target` is:
  - `mex` for MEX functions
  - `lib` for static C/C++ libraries
  - `dll` for dynamic C/C++ libraries
  - `exe` for C/C++ executables

Command-Line Alternative

Use the `codegen` function `-o` option.

Specify Output File Locations

Specify Output File Location Using the MATLAB Coder App

The output file location must not contain:

- Spaces (Spaces can lead to code generation failures in certain operating system configurations).
- Tabs
- `\`, `$`, `#`, `*`, `?`
- Non-7-bit ASCII characters, such as Japanese characters.

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
3. Click **More Settings**.
4. Click the **Paths** tab.

   The default setting for the **Build folder** field is `A subfolder of the project folder`. By default, MATLAB Coder generates files in the folder *`project_folder`*`/codegen/target/fcn1`:

   - `fcn1` is the name of the alphabetically first entry-point file.
   - `target` is:

     - `mex` for MEX functions
     - `lib` for static C/C++ libraries
     - `dll` for dynamically linked C/C++ libraries
     - `exe` for C/C++ executables
5. To change the output location, you can either:

   - Set **Build Folder** to `A subfolder of the current MATLAB working folder`

     MATLAB Coder generates files in the *`MATLAB_working_folder`*`/codegen/target/fcn1` folder
   - Set **Build Folder** to `Specified folder`. In the **Build folder name** field, provide the path to the folder.

Command-Line Alternative

Use the `codegen` function `-d` option.

Parameter Specification Methods

If you are using	Use	Details
The MATLAB Coder app	The project settings dialog box.	"Specify Build Configuration Parameters MATLAB Coder App" on page 20-32
`codegen` at the command line and want to specify a few parameters	Configuration objects	"Specify Build Configuration Parameters at the Command Line Using Configuration Objects" on page 20-33
`codegen` in build scripts		
`codegen` at the command line and want to specify many parameters	Configuration object dialog boxes	"Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes" on page 20-37

Specify Build Configuration Parameters

- "Specify Build Configuration Parameters MATLAB Coder App" on page 20-32
- "Specify Build Configuration Parameters at the Command Line Using Configuration Objects" on page 20-33
- "Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes" on page 20-37

You can specify build configuration parameters from the MATLAB Coder project settings dialog box, the command line, or configuration object dialog boxes.

Specify Build Configuration Parameters MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
3. Click **More Settings**.

   The project settings dialog box provides the set of configuration parameters applicable to the output type that you select. Code generation uses a different set of configuration parameters for MEX functions than it uses for the other build types.

When you switch the output type between `MEX Function` and `Source Code`, `Static Library`, `Dynamic Library`, or `Executable`, verify these settings. See "Changing Output Type" on page 17-35.

4 Modify the parameters as required. For more information about parameters on a tab, click **Help**.

Changes to the parameter settings take place immediately.

Specify Build Configuration Parameters at the Command Line Using Configuration Objects

Types of Configuration Objects

The `codegen` function uses configuration objects to customize your environment for code generation. The following table lists the available configuration objects.

Configuration Object	Description
coder.CodeConfig	If no Embedded Coder license is available or you disable use of the Embedded Coder license, specifies parameters for C/C++ library or executable generation. For more information, see the class reference information for `coder.CodeConfig`.
coder.EmbeddedCodeConfig	If an Embedded Coder license is available, specifies parameters for C/C++ library or executable generation. For more information, see the class reference information for `coder.EmbeddedCodeConfig`.
coder.HardwareImplementation	Specifies parameters of the target hardware implementation. If not specified, `codegen` generates code that is compatible with the MATLAB host computer. For more information, see the class reference information for `coder.HardwareImplementation`.
coder.MexCodeConfig	Specifies parameters for MEX code generation. For more information, see the class reference information for `coder.MexCodeConfig`.

Working with Configuration Objects

To use configuration objects to customize your environment for code generation:

1. In the MATLAB workspace, define configuration object variables, as described in “Creating Configuration Objects” on page 20-35.

   For example, to generate a configuration object for C static library generation:

   ```
   cfg = coder.config('lib');
   % Returns a coder.CodeConfig object if no
   % Embedded Coder license available.
   % Otherwise, returns a coder.EmbeddedCodeConfig object.
   ```
2. Modify the parameters of the configuration object as required, using one of these methods:

   - Interactive commands, as described in “Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 20-33
   - Dialog boxes, as described in “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 20-37
3. Call the `codegen` function with the `-config` option. Specify the configuration object as its argument.

   The `-config` option instructs `codegen` to generate code for the target, based on the configuration property values. In the following example, `codegen` generates a C static library from a MATLAB function, `foo`, based on the parameters of a code generation configuration object, `cfg`, defined in the first step:

   ```
   codegen -config cfg foo
   ```

   The `-config` option specifies the type of output that you want to build — in this case, a C static library. For more information, see `codegen`.

Creating Configuration Objects

You can define a configuration object in the MATLAB workspace.

To Create...	Use a Command Such As...
MEX configuration object coder.MexCodeConfig	`cfg = coder.config('mex');`

Code generation configuration object for generating a standalone C/C++ library or executable coder.CodeConfig	`% To generate a static library`
	`cfg = coder.config('lib');`
	`% To generate a dynamic library`
	`cfg = coder.config('dll')`
	`% To generate an executable`
	`cfg = coder.config('exe');`
	Note: If an Embedded Coder license is available, creates a `coder.EmbeddedCodeConfig` object.
	If you use concurrent licenses, to disable the check out of an Embedded Coder license, use one of the following commands:
	`cfg = coder.config('lib', 'ecoder', false)`
	`cfg = coder.config('dll', 'ecoder', false)`
	`cfg = coder.config('exe', 'ecoder', false)`

Code generation configuration object for generating a standalone C/C++ library or executable for an embedded target coder.EmbeddedCodeConfig	`% To generate a static library`
	`cfg = coder.config('lib');`
	`% To generate a dynamic library`
	`cfg = coder.config('dll')`
	`% To generate an executable`
	`cfg = coder.config('exe');`
	Note: Requires an Embedded Coder license; otherwise creates a `coder.CodeConfig` object.

Hardware implementation configuration object coder.HardwareImplementation	`hwcfg = coder.HardwareImplementation`

Each configuration object comes with a set of parameters, initialized to default values. You can change these settings, as described in “Modifying Configuration Objects at the Command Line Using Dot Notation” on page 20-36.

Modifying Configuration Objects at the Command Line Using Dot Notation

You can use dot notation to modify the value of one configuration object parameter at a time. Use this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties:

- To specify a `main` function during C/C++ code generation:

  ```
  cfg = coder.config('exe');
  cfg.CustomInclude = 'c:\myfiles';
  cfg.CustomSource = 'main.c';
  codegen -config cfg foo
  ```
- To automatically generate and launch code generation reports after generating a C/C+ + static library:

  ```
  cfg = coder.config('lib');
  cfg.GenerateReport= true;
  cfg.LaunchReport = true;
  codegen -config cfg  foo
  ```

Saving Configuration Objects

Configuration objects do not automatically persist between MATLAB sessions. Use one of the following methods to preserve your settings:

Save a configuration object to a MAT-file and then load the MAT-file at your next session

For example, assume that you create and customize a MEX configuration object `mexcfg` in the MATLAB workspace. To save the configuration object, at the MATLAB prompt, enter:

```
save mexcfg.mat mexcfg
```

The `save` command saves `mexcfg` to the file `mexcfg.mat` in the current folder.

To restore `mexcfg` in a new MATLAB session, at the MATLAB prompt, enter:

```
load mexcfg.mat
```

The load command loads the objects defined in mexcfg.mat to the MATLAB workspace.

Write a script that creates the configuration object and sets its properties.

You can rerun the script whenever you need to use the configuration object again.

Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes

1. Create a configuration object as described in "Creating Configuration Objects" on page 20-35.

   For example, to create a coder.MexCodeConfig configuration object for MEX code generation:

   ```
   mexcfg = coder.config('mex');
   ```
2. Open the property dialog box using one of these methods:

   - In the MATLAB workspace, double-click the configuration object variable.
   - At the MATLAB prompt, issue the open command, passing it the configuration object variable, as in this example:

     ```
     open mexcfg
     ```
3. In the dialog box, modify configuration parameters as required, then click **Apply**.
4. Call the codegen function with the -config option. Specify the configuration object as its argument:

   ```
   codegen -config mexcfg foo
   ```

   The -config option specifies the type of output that you want to build. For more information, see codegen.

Specify Data Types Used in Generated Code

In this section...

MATLAB Coder can use built-in C data types or predefined types from `rtwtypes.h` in generated code. By default, when the generated code declares variables, it uses built-in C types.

You can explicitly specify the data types used in generated code in the project settings dialog box or at the command line.

Specify Data Type Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
3. Click **More Settings**.
4. To use built-in C types, on the **Code Appearance** tab, set **Data Type Replacement** to `Use built-in C data types in the generated code`. To use predefined types from `rtwtypes.h`, set **Data Type Replacement** to `Use MathWorks typedefs in the generated code`.

Specify Data Type at the Command Line

1. Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`,`'dll'`, or `'exe'` (depending on your requirements). For example:

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

2. To use built-in C types, set the `DataTypeReplacement` property to `'CBuiltIn'`.

   ```
   cfg.DataTypeReplacement = 'CBuiltIn';
   ```

   To use predefined types from `rtwtypes.h`, set the `DataTypeReplacement` property to `'CoderTypedefs'`.

Change the Standard Math Library

For calls to math operations, the code generator uses the standard math library that you specify in the build settings. The default standard math library depends on the language that you select. For C, it is C89/C90 (ANSI). For C++, it is C++03 (ISO).

You can change the standard math library to one of these libraries.

Library Name	Language Support	Standard
C89/C90 (ANSI)	C, C++	ISO/IEC 9899:1990
C99 (ISO)	C, C++	ISO/IEC 9899:1999
C++03 (ISO)	C++	ISO/IEC 14882:2003

The C++03 (ISO) math library is available only if the language is C++.

To change the library:

- In the project build settings, on the **Custom Code** tab, set the **Standard math library** parameter.
- In a code configuration object, set the `TargetLangStandard` parameter.

Verify that your compiler supports the library that you want to use. If you select a library that your compiler does not support, compiler errors can occur.

More About

- "Specify Build Configuration Parameters MATLAB Coder App" on page 20-32
- "Specify Build Configuration Parameters at the Command Line Using Configuration Objects" on page 20-33

Share Build Configuration Settings

To share build configuration settings between multiple projects or between the project and command-line workflow, you can export settings to and import settings from a code generation configuration object.

Export Settings

To export the current project settings to a code generation configuration object stored in the base workspace:

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`), or `Executable` (depending on your requirements).
3. Click **More Settings**.
4. Click **Import/Export Settings**.
5. In the **Variable name** field, specify a name for the configuration object.
6. Click **Export to Variable**.

   MATLAB Coder saves the project settings information in a configuration object with the specified name in the base workspace.

Project Output Type	Configuration Object
`MEX Function`	coder.MexCodeConfig
`C/C++ Static Library`	Without an Embedded Coder license: coder.CodeConfig With an Embedded Coder license: coder.EmbeddedCodeConfig
`C/C++ Dynamic Library`	
`C/C++ Executable`	

You can then either import these settings into another project or use the configuration object with the `codegen` function `-config` option to generate code at the command line.

Import Settings

To import the settings saved in a code generation configuration object stored in the base workspace:

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
3. Click **More Settings**.
4. Click **Import/Export Settings**.
5. In the **Variable name** field, specify the name of the configuration object.
6. Click **Import from Variable**.

See Also

- "Configure Build Settings" on page 20-26
- `coder.config`
- "Convert MATLAB Coder Project to MATLAB Script" on page 20-42

Convert MATLAB Coder Project to MATLAB Script

After you define input types, you can convert a MATLAB Coder project to the equivalent script of MATLAB commands. The script reproduces the project in a configuration object and runs the `codegen` command. You can:

- Move from a project workflow to a command-line workflow.
- Save the project as a text file that you can share.

You can convert a project using the MATLAB Coder app or the command-line interface.

Project to script conversion does not support entry-point function inputs that are value classes.

Convert a Project Using the MATLAB Coder App

1. On the app toolbar, click ☰, and then select **Convert to script**.
2. Specify the script name and click **Save**.

Convert a Project Using the Command-Line Interface

To convert a project to a script using the command-line interface, use the `-tocode` option of the `coder` command. The project file must be on the search path.

For example, to convert the project, `myproject.prj` to the script named `myscript.m` use this command:

```
coder -tocode myproject -script myscript.m
```

The `coder` command overwrites a file that has the same name as the script. If you omit the `-script` option, the `coder` command writes the script to the Command Window.

For more information about the `-tocode` option, see `coder`.

Run the Script

1. Make sure that the entry-point functions that are arguments to `codegen` in the script are on the search path.

2 Run the script. For example:

```
myscript
```

The following variables appear in the base workspace.

Variable	For
`cfg`	Configuration object
`ARGS`	Types of input arguments, if the project has entry-point function inputs
`ARG`	Types of cell array elements, if the project has cell array inputs. A script can reuse `ARG` for different cell array elements
`GLOBALS`	Types and initial values of global variables, if the project has global variables

cfg, ARGS, ARG, and GLOBALS appear in the workspace only after you run the script. The type of configuration object depends on the project output type.

Project Output Type	Configuration Object
`MEX Function`	coder.MexCodeConfig
`C/C++ Static Library`	Without an Embedded Coder license: coder.CodeConfig With an Embedded Coder license: coder.EmbeddedCodeConfig
`C/C++ Dynamic Library`	
`C/C++ Executable`	

You can import the settings from the configuration object `cfg` into a project. See “Share Build Configuration Settings” on page 20-40.

For a project that includes fixed-point conversion, project to script conversion generates a pair of scripts for fixed-point conversion and fixed-point code generation. For an example, see “Convert Fixed-Point Conversion Project to MATLAB Scripts” on page 14-104.

P

Preserve Variable Names in Generated Code

If code readability is more important than reduced memory usage, specify that you want the code generator to preserve your variable names rather than reuse them in the generated code.

By default, when possible, variables share names and memory in the generated code. The code generator reuses your variable names for other variables or reuses other variable names for your variables. For example, for code such as:

```
if (s>0)
    myvar1 = 0;
    ...
else
    myvar2 = 0;
    ...
end
```

the generated code can look like this code:

```
 if (s > 0.0) {
   myvar2 = 0.0;
    ...
 } else {
   myvar2 = 0.0;
   ...
 }
```

When the code generator preserves your variable names, the generated code can look like this code:

```
 if (s > 0.0) {
   myvar1 = 0.0;
    ...
 } else {
   myvar2 = 0.0;
   ...
 }
```

To specify that you want the code generator to preserve your variable names:

- In a code generation configuration object, set the `PreserveVariableNames` parameter to `'UserNames'`.

- In the MATLAB Coder app, set **Preserve variable names** to `User names`.

Preservation of variable names does not prevent an optimization from removing them from the generated code or prevent the C/C++ compiler from reusing them in the generated binary code.

More About

- "Reuse Large Arrays and Structures" on page 28-56
- "Configure Build Settings" on page 20-26

Specify Properties of Entry-Point Function Inputs

In this section...

Why You Must Specify Input Properties

Because C and C++ are statically typed languages, MATLAB Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, MATLAB Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, to MATLAB Coder. If your primary function has no input parameters, MATLAB Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the primary function.

If you use the tilde (~) character to specify unused function inputs:

- In MATLAB Coder projects, if you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.
- When generating code with `codegen`, you must specify the type of these inputs using the `-args` option.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For	Specify properties				
	Class	**Size**	**Complexity**	**numerictype**	**fimath**

For	Specify properties				
Fixed-point inputs	✓	✓	✓	✓	✓
Each field in a structure input	**Specify properties for each field according to its class** When a primary input is a structure, the code generator treats each field as a separate input. Therefore, you must specify properties for allfields of a primary structure input in the order that they appear in the structure definition: • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify `numerictype`, and `fimath`.				
Other inputs	✓	✓	✓		

Default Property Values

MATLAB Coder assigns the following default values for properties of primary function inputs.

Property	Default
class	`double`
size	`scalar`
complexity	`real`
`numerictype`	No default
`fimath`	MATLAB default `fimath` object

Specifying Default Values for Structure Fields

In most cases, when you do not explicitly specify values for properties, MATLAB Coder uses defaults except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you might need to specify default values for properties of structure fields. For examples, see “Specifying Class and Size of Scalar Structure” on page 20-78 and “Specifying Class and Size of Structure Array” on page 20-79.

Specifying Default fimath Values for MEX Functions

MEX functions generated with MATLAB Coder use the default `fimath` value in effect at compile time. If you do not specify a default `fimath` value, MATLAB Coder uses the MATLAB default `fimath`. The MATLAB factory default has the following properties:

```
RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
CastBeforeSum: true
```

For more information, see “fimath for Sharing Arithmetic Rules” (Fixed-Point Designer).

When running MEX functions that depend on the default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time warning, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose that you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` relies on the default `fimath` object in effect at compile time. At the MATLAB prompt, generate the MEX function `text_mex` to use the factory setting of the MATLAB default `fimath`:

```
codegen test
% codegen generates a MEX function, test_mex,
% in the current folder
```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```
test_mex

ans =

     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Now create a local MATLAB `fimath` value. so you no longer use the default setting:

```
F = fimath('RoundingMethod','Floor');
```

Finally, clear the MEX function from memory and rerun it:

```
clear test_mex
test_mex
```

The mismatch is detected and causes an error:

```
??? This function was generated with a different default
fimath than the current default.

Error in ==> test_mex
```

Supported Classes

The following table presents the class names supported by MATLAB Coder.

Class Name	Description
`logical`	Logical array of true and false values
`char`	Character array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64–bit unsigned integer array
`single`	Single-precision floating-point or fixed-point number array
`double`	Double-precision floating-point or fixed-point number array
`struct`	Structure array
`embedded.fi`	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules:

- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature. For example, the first element in the cell array defines the properties of the first primary function input.
- To generate fewer arguments than those arguments that occur in the MATLAB function, specify properties for only the number of arguments that you want in the generated function.
- If the MATLAB function has input arguments, to generate a function that has no input arguments, pass an empty cell array to `-args`.
- For each primary function input whose class is fixed point (`fi`), specify the input `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages
"Specify Properties of Entry-Point Function Inputs Using the App" on page 17-3	• If you are working in a MATLAB Coder project, easy to use • Does not alter original MATLAB code • MATLAB Coder saves the definitions in the project file	• Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Define Input Properties by Example at the Command Line" on page 20-51 **Note:** If you define input properties programmatically in the MATLAB file, you cannot use this method	• Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a few primary inputs	• Must be specified at the command line every time you invoke `codegen` (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
"Define Input Properties Programmatically in	• Integrated with MATLAB code; no need to redefine properties	• Uses complex syntax

Method	Advantages	Disadvantages
the MATLAB File" on page 20-68	each time you invoke MATLAB Coder • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures	• MATLAB Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.

Define Input Properties by Example at the Command Line

- "Command-Line Option -args" on page 20-51
- "Rules for Using the -args Option" on page 20-51
- "Specifying Properties of Primary Inputs by Example at the Command Line" on page 20-52
- "Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line" on page 20-53

Command-Line Option -args

The `codegen` function provides a command-line option `-args` for specifying the properties of primary (entry-point) function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you generate code for the MATLAB function with `codegen` .

If you have a test function or script that calls the entry-point MATLAB function with the required types, you can use `coder.getArgTypes` to determine the types of the function inputs. `coder.getArgTypes` returns a cell array of coder.Type objects that you can pass to `codegen` using the `-args` option. See "Specifying General Properties of Primary Inputs" on page 20-75 for `codegen`.

For information about specifying cell array inputs, see "Specify Cell Array Inputs at the Command Line" on page 20-57.

Rules for Using the -args Option

When using the `-args` command-line option to define properties by example, follow these rules:

- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature. For example, the first element in the cell array defines the properties of the first primary function input.
- To generate fewer arguments than those arguments that occur in the MATLAB function, specify properties for only the number of arguments that you want in the generated function.
- If the MATLAB function has input arguments, to generate a function that has no input arguments, pass an empty cell array to `-args`.
- For each primary function input whose class is fixed point (`fi`), specify the input `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, specify the properties of each of its fields in the order that they appear in the structure definition.

Specifying Properties of Primary Inputs by Example at the Command Line

Consider a MATLAB function that adds its two inputs:

```
function y = mcf(u,v)
%#codegen
y = u + v;
```

The following examples show how to specify different properties of the primary inputs `u` and `v` by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

  ```
  codegen mcf -args {0,0}
  ```
- Use a literal cell array of constants to specify that input `u` is an unsigned 16-bit, 1-by-4 vector and input `v` is a scalar double:

  ```
  codegen  mcf -args {zeros(1,4,'uint16'),0}
  ```
- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

  ```
  a = uint8([1;2;3;4])
  b = uint8([5;6;7;8])
  ex = {a,b}
  codegen mcf -args ex
  ```

Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line

To generate a MEX function or C/C++ code for fixed-point MATLAB code, you must install Fixed-Point Designer software.

Consider a MATLAB function that calculates the square root of a fixed-point number:

```
%#codegen
function y = sqrtfi(x)
y = sqrt(x);
```

To specify the properties of the primary fixed-point input `x` by example, follow these steps:

1. Define the `numerictype` properties for `x`, for example:

   ```
   T = numerictype('WordLength',32,...
                   'FractionLength',23,...
                   'Signed',true);
   ```
2. Define the `fimath` properties for `x`, for example:

   ```
   F = fimath('SumMode','SpecifyPrecision',...
              'SumWordLength',32,...
              'SumFractionLength',23,...
              'ProductMode','SpecifyPrecision',...
              'ProductWordLength',32,...
              'ProductFractionLength',23);
   ```
3. Create a fixed-point variable with the `numerictype` and `fimath` properties that you defined, for example:

   ```
   myeg = { fi(4.0,T,F) };
   ```
4. Compile the function `sqrtfi` using the `codegen` command, passing the variable `myeg` as the argument to the `-args` option, for example:

   ```
   codegen sqrtfi -args myeg;
   ```

Specify Constant Inputs at the Command Line

If you know that your primary inputs do not change at run time, you can reduce overhead in the generated code by specifying that the primary inputs are constant values. Constant inputs are commonly used for flags that control how an algorithm executes and values that specify the sizes or types of data.

To specify that inputs are constants, use the `-args` command-line option with a `coder.Constant` object. To specify that an input is a constant with the size, class, complexity, and value of `constant_input`, use the following syntax:

```
-args {coder.Constant(constant_input)}
```

Calling Functions with Constant Inputs

The code generator compiles constant function inputs into the generated code. In the generated C or C++ code, function signatures do not contain the constant inputs. By default, MEX function signatures contain the constant inputs. When you call a MEX function, you must provide the compile-time constant values. The constant input values must match the compile-time values. You can control whether a MEX function signature includes constant inputs and whether the constant input values must match the compile-time values. See “Control Constant Inputs in MEX Function Signatures” on page 20-64.

Specifying a Structure as a Constant Input

Suppose that you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
y = zeros(p.rows,p.cols) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
codegen rowcol -args {0,coder.Constant(tmp)}
```

Specify Variable-Size Inputs at the Command Line

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. You can define inputs to have one or more variable-size dimensions — and specify their upper bounds — using the `-args` option and `coder.typeof` function:

-args {coder.typeof(*example_value*, *size_vector*, *variable_dims*}
Specifies a variable-size input with:

- Same class and complexity as *example_value*
- Same size and upper bounds as *size_vector*
- Variable dimensions specified by *variable_dims*

When you enable dynamic memory allocation, you can specify Inf in the size vector for dimensions with unknown upper bounds at compile time.

When *variable_dims* is a scalar, it is applied to all the dimensions, with the following exceptions:

- If the dimension is 1 or 0, which are fixed.
- If the dimension is unbounded, which is always variable size.

For more information, see coder.typeof and "Generate Code for Variable-Size Data" on page 20-108.

Specifying a Variable-Size Vector Input

1 Write a function that computes the average of every n elements of a vector A and stores them in a vector B:

```
function B = nway(A,n) %#codegen
% Compute average of every N elements of A and put them in B.

coder.extrinsic('error');
if ((mod(numel(A),n) == 0) && (n>=1 && n<=numel(A)))
    B = ones(1,numel(A)/n);
    k = 1;
    for i = 1 : numel(A)/n
         B(i) = mean(A(k + (0:n-1)));
         k = k + n;
    end
else
    B = zeros(1,0);
    error('n <= 0 or does not divide number of elements evenly');
end
```

2 Specify the first input A as a vector of double values. Its first dimension stays fixed in size and its second dimension can grow to an upper bound of 100. Specify the second input n as a double scalar.

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

3 As an alternative, assign the `coder.typeof` expression to a MATLAB variable, then pass the variable as an argument to `-args`:

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

More About

- "Specify Objects as Inputs at the Command Line" on page 10-29
- "Specify Cell Array Inputs at the Command Line" on page 20-57
- "Specify Number of Entry-Point Function Input or Output Arguments to Generate" on page 12-3

Specify Cell Array Inputs at the Command Line

To specify cell array inputs at the command line, use the same methods that you use for other types of inputs. You can:

- Provide an example cell array input to the `-args` option of the `codegen` command.
- Provide a `coder.CellType` object to the `-args` option of the `codegen` command. To create a `coder.CellType` object, use `coder.typeof`.
- Use `coder.Constant` to specify a constant cell array input.

For code generation, cell arrays are classified as homogeneous or heterogeneous. See "Code Generation for Cell Arrays" on page 8-2. When you provide an example cell array to `codegen` or `coder.typeof`, the function determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `coder.CellType` `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

If you have a test file, you can use `coder.getArgTypes` to determine input types. In the output cell array of types, for cell array inputs, `coder.getArgTypes` returns a `coder.CellType` object. If you want a different classification (homogeneous or heterogeneous), use the `makeHomogeneous` or `makeHeterogeneous` methods.

Specify Cell Array Inputs by Example

To specify a cell array input by example, provide an example cell array in the `-args` option of the `codegen` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

  ```
  codegen myfunction -args {{1 2 3}} -report
  ```

  The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.
- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

  ```
  codegen myfunction -args {{'a', 1}} -report
  ```

  The input argument is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.

Specify the Type of the Cell Array Input

To specify the type of a cell array input, use `coder.typeof` to create a `coder.CellType` object. Pass the `coder.CellType` object to the `-args` option of the `codegen` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

  ```
  t = coder.typeof({1 2 3});
  codegen myfunction -args {t} -report
  ```

  The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.
- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

  ```
  t = coder.typeof({'a', 1});
  codegen myfunction -args {t}
  ```

  The input argument is a 1x2 heterogeneous cell array whose first element is a 1x1 char and whose second element is a 1x1 double.

You can also use the advanced function `coder.newtype` to create a `coder.CellType` object.

Make a Homogeneous Copy of a Type

If `coder.typeof` returns a heterogeneous cell array type, but you want a homogeneous type, use the `makeHomogeneous` method to make a homogeneous copy of the type.

The following code creates a heterogeneous type.

```
t = coder.typeof({1 [2 3]})
```

```
t = 

coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 double
      f1: 1x2 double
```

To make a homogeneous copy of the type, use:

```
t = makeHomogeneous(t)
```

```
t = 

coder.CellType
   1×2 locked homogeneous cell
      base: 1×:2 double
```

Alternatively, use this notation:

```
t = makeHomogeneous(coder.typeof({1 [2 3]}))
```

```
t = 

coder.CellType
   1×2 locked homogeneous cell
      base: 1×:2 double
```

The classification as homogeneous is locked (permanent). You cannot later use the `makeHeterogeneous` method to make a heterogeneous copy of the type.

If the elements of a type have different classes, such as char and double, you cannot use `makeHomogeneous` to make a homogeneous copy of the type.

If you use `coder.cstructname` to specify a name for the structure type that represents a type in the generated code, you cannot create a homogeneous copy of the type.

Make a Heterogeneous Copy of a Type

If `coder.typeof` returns a homogeneous cell array type, but you want a heterogeneous type, use the `makeHeterogeneous` method to make a heterogeneous copy of the type.

The following code creates a homogeneous type.

```
t = coder.typeof({1 2 3})
```

```
t =

coder.CellType
   1x3 homogeneous cell
      base: 1x1 double
```

To make the type heterogeneous, use:

```
t = makeHeterogeneous(t)
```

```
t =

coder.CellType
   1×3 locked heterogeneous cell
      f1: 1×1 double
      f2: 1×1 double
      f3: 1×1 double
```

Alternatively, use this notation:

```
t = makeHeterogeneous(coder.typeof({1 2 3}))
```

```
t =

coder.CellType
   1×3 locked heterogeneous cell
      f1: 1×1 double
      f2: 1×1 double
      f3: 1×1 double
```

The classification as heterogeneous is locked (permanent). You cannot later use the `makeHomogeneous` method to make a homogeneous copy of the type.

If a type is variable size, you cannot use `makeHeterogeneous` to make a heterogeneous copy of it.

Specify Variable-Size Cell Array Inputs

You can specify variable-size cell array inputs in the following ways:

- In the `coder.typeof` call.

  For example, to specify a variable-size cell array whose first dimension is fixed and whose second dimension has an upper bound of 5:

  ```
  t = coder.typeof({1}, [1 5], [0 1])

  t = 

  coder.CellType
     1x:5 homogeneous cell 
        base: 1x1 double
  ```

  For elements with the same classes, but different sizes, you can the use `coder.typeof` size and variable dimensions arguments to create a variable-size homogeneous cell array type. For example, the following code does not use the size and variable dimensions arguments. This code creates a type for a heterogeneous cell array.

  ```
  t = coder.typeof({1 [2 3]})

  t = 

  coder.CellType
     1x2 heterogeneous cell 
        f0: 1x1 double
        f1: 1x2 double
  ```

  The following code, that uses the size and dimensions arguments, creates a type for a variable-size homogeneous type cell array:

  ```
  t = coder.typeof({1 [2 3]}, [1 5], [0 1])

  t = 

  coder.CellType
     1×:5 locked homogeneous cell
        base: 1×:2 double
  ```

- Use `coder.resize`.

For example, to specify a variable-size cell array whose first dimension is fixed and whose second dimension has an upper bound of 5:

```
t = coder.typeof({1});
t = coder.resize(t, [1 5], [0,1])

t = 

coder.CellType
   1x5 homogeneous cell 
      base: 1x1 double
```

You cannot use `coder.resize` with a heterogeneous cell array type.

Specify Type Name for Heterogeneous Cell Array Inputs

A heterogeneous cell array is represented in the generated code as a structure. To specify the name of the structure type in the generated code, use `coder.cstructname`.

For example, to specify the name `myname` for the cell array type in the generated code:

```
t = coder.typeof({'a', 1})
t = coder.cstructname(t, 'myname')

t = 

coder.CellType
   1×2 locked heterogeneous cell myname
      f1: 1×1 char
      f2: 1×1 double
```

If you use `coder.cstructname` with a homogeneous cell array type, `coder.cstructname` returns a heterogeneous copy of the type. However, it is a best practice to use the `makeHeterogeneous` method of the `coder.CellType` object to make a heterogeneous copy of a homogeneous cell array type. Then, you can use `coder.cstructname` with the heterogeneous copy of the type.

Specify Constant Cell Array Inputs

To specify that a cell array input is constant, use the `coder.Constant` function with the `-args` option of the `codegen` command. For example:

```
codegen myfunction -args {coder.Constant({'red', 1  'green', 2,  'blue', 3})} -report
```

The input is a 1x6 heterogeneous cell array. The sizes and classes of the elements are:

- 1x3 char
- 1x1 double
- 1x5 char
- 1x1 double
- 1x4 char
- 1x1 double

See Also

coder.CellType | `coder.getArgTypes` | `coder.newtype` | `coder.resize` | `coder.typeof`

Related Examples

- “Define Input Properties by Example at the Command Line” on page 20-51
- “Specify Constant Inputs at the Command Line” on page 20-53

More About

- “Code Generation for Cell Arrays” on page 8-2

Control Constant Inputs in MEX Function Signatures

In this section...

You can control whether a generated MEX function signature includes constant inputs. If you want to use the same test file to run the original MATLAB function and the MEX function, then the MEX function signature must contain the constant inputs. You can also control whether the run-time values of the constant inputs must match the compile-time values. Checking that the values match can slow down execution speed.

Control MEX Function Signature Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `MEX`.
3. Click **More Settings**.
4. On the **All Settings** tab, set **Constant Inputs** to one of the menu options. See “Options for Controlling Constant Inputs in MEX Function Signatures” on page 20-65.

Control MEX Function Signature at the Command-Line Interface

1. Create a code configuration object for MEX code generation.

   ```
   mexcfg = coder.config('mex');
   ```
2. Set the `ConstantInputs` parameter to `'CheckValues'`, `'IgnoreValues'`, or `'Remove'` For example:

   ```
   mexcfg.ConstantInputs = 'IgnoreValues';
   ```

   For a description of the options, see “Options for Controlling Constant Inputs in MEX Function Signatures” on page 20-65

Options for Controlling Constant Inputs in MEX Function Signatures

The following table lists the options for the:

- **Constant Inputs** setting in a project with **Output Type** set to `MEX`.
- `ConstantInputs` property in a configuration object for MEX code generation.

Constant Inputs (Project)	ConstantInputs (Configuration Object)	Description
`Check values at run time (default)`	`'CheckValues'`	• The MEX function signature includes the constant inputs. When you call the function, you must provide the constant inputs. • The run-time values of the constant inputs must match the compile-time values. When you call the function, you must provide the value that was used at compile-time. • Allows you to use the same test file to run the original MATLAB algorithm and the MEX function. • Slows down execution of the MEX function. • This setting is the default.
`Ignore input value`	`'IgnoreValues'`	• The MEX function signature includes the constant inputs. When you call the function, you must provide the constant inputs. • The run-time values of the constant inputs can differ from the compile-time values.

Constant Inputs (Project)	ConstantInputs (Configuration Object)	Description
		- Allows you to use the same test file to run the original MATLAB algorithm and the MEX function.
`Remove from MEX signature`	`'Remove'`	The MEX function signature does not include the constant inputs. When you call the function, you do not provide the constant inputs.

Call MEX Function with a Constant Input

This example shows how to call MEX functions that have constant inputs. It shows how to use the `ConstantInputs` parameter to control whether the MEX function signature includes constant inputs and whether the constant input values must match the compile-time values.

Write a function `identity` that copies its input to its output.

```
function y = identity(u) %#codegen
y = u;
```

Create a code configuration object for MEX code generation.

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

Generate a MEX function `identity_mex` with the constant input `42`.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call `identity_mex`. You must provide the input `42`.

```
identity_mex(42)
```

```
ans =

    42
```

Configure `ConstantInputs` so that the MEX function does not check that the input value matches the compile-time value.

```
cfg.ConstantInputs = 'IgnoreValues';
```

Generate identity_mex with the new configuration.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call identity_mex with a constant input value other than 42 .

```
identity_mex(50)
```

```
ans =

    42
```

The MEX function ignored the input value 50.

Configure ConstantInputs so that the MEX function does not include the constant input.

```
cfg.ConstantInputs = 'Remove';
```

Generate identity_mex with the new configuration.

```
codegen identity -config cfg -args {coder.Constant(42)}
```

Call identity_mex. Do not provide the input value .

```
identity_mex()
```

```
ans =

    42
```

See Also

- “Specify Constant Inputs at the Command Line” on page 20-53
- “Define Constant Input Parameters Using the App” on page 17-27

Define Input Properties Programmatically in the MATLAB File

For code generation, you can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

How to Use assert with MATLAB Coder

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

When specifying input properties using the `assert` function, use one of the following methods. Use the exact syntax that is provided; do not modify it.

- "Specify Any Class" on page 20-68
- "Specify fi Class" on page 20-69
- "Specify Structure Class" on page 20-69
- "Specify Cell Array Class" on page 20-70
- "Specify Fixed Size" on page 20-70
- "Specify Scalar Size" on page 20-70
- "Specify Upper Bounds for Variable-Size Inputs" on page 20-71
- "Specify Inputs with Fixed- and Variable-Size Dimensions" on page 20-71
- "Specify Size of Individual Dimensions" on page 20-72
- "Specify Real Input" on page 20-72
- "Specify Complex Input" on page 20-72
- "Specify numerictype of Fixed-Point Input" on page 20-73
- "Specify fimath of Fixed-Point Input" on page 20-73
- "Specify Multiple Properties of Input" on page 20-73

Specify Any Class

```
assert ( isa ( param, 'class_name') )
```

Sets the input parameter *`param`* to the MATLAB class *`class_name`*. For example, to set the class of input `U` to a 32-bit signed integer, call:

```
...
assert(isa(U,'int32'));
...
```

Specify fi Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class `fi` (fixed-point numeric object). For example, to set the class of input `U` to `fi`, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U,'embedded.fi'));
...
```

You must specify both the `fi` class and the `numerictype`. See "Specify numerictype of Fixed-Point Input" on page 20-73. You can also set the `fimath` properties, see "Specify fimath of Fixed-Point Input" on page 20-73. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.

Specify Structure Class

```
assert ( isstruct ( param ) )
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter `param` to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U, 'struct'));
```

```
...
```

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order that they appear in the structure definition.

Specify Cell Array Class

```
assert(iscell( param))
assert(isa(param, 'cell'))
```

Sets the input parameter *`param`* to the MATLAB class `cell` (cell array). For example, to set the class of input `C` to a `cell`, call:

```
...
assert(iscell(C));
...
```

or

```
...
assert(isa(C, 'cell'));
...
```

To specify the properties of cell array elements, see “Specifying Properties of Cell Arrays” on page 20-76.

Specify Fixed Size

```
assert ( all ( size (param) == [dims ] ) )
```

Sets the input parameter *`param`* to the size that dimensions *`dims`* specifies. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size

```
assert ( isscalar (param ) )
assert ( all ( size (param) == [ 1 ] ) )
```

Sets the size of input parameter *`param`* to scalar. To set the size of input `U` to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
assert(all(size(U)== [1]));
...
```

Specify Upper Bounds for Variable-Size Inputs

```
assert ( all(size(param)<=[N0 N1 ...]));
assert ( all(size(param)<[N0 N1 ...]));
```

Sets the upper-bound size of each dimension of input parameter *param*. To set the upper-bound size of input U to be less than or equal to a 3-by-2 matrix, call:

```
assert(all(size(U)<=[3 2]));
```

Note: You can also specify upper bounds for variable-size inputs using coder.varsize.

Specify Inputs with Fixed- and Variable-Size Dimensions

```
assert ( all(size(param)>=[M0 M1 ...]));
assert ( all(size(param)<=[N0 N1 ...]));
```

When you use assert(all(size(*param*)>=[M0 M1 ...])) to specify the lower-bound size of each dimension of an input parameter:

- You must also specify an upper-bound size for each dimension of the input parameter.
- For each dimension, k, the lower-bound Mk must be less than or equal to the upper-bound Nk.
- To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
- Bounds must be nonnegative.

To fix the size of the first dimension of input U to 3 and set the second dimension as variable size with upper bound of 2, call:

```
assert(all(size(U)>=[3 0]));
assert(all(size(U)<=[3 2]));
```

Specify Size of Individual Dimensions

```
assert (size(param, k)==Nk);
assert (size(param, k)<=Nk);
assert (size(param, k)<Nk);
```

You can specify individual dimensions and all dimensions simultaneously. You can also specify individual dimensions instead of specifying all dimensions simultaneously. The following rules apply:

- You must specify the size of each dimension at least once.
- The last dimension specification takes precedence over earlier specifications.

Sets the upper-bound size of dimension `k` of input parameter *`param`*. To set the upper-bound size of the first dimension of input `U` to 3, call:

```
assert(size(U,1)<=3)
```

To fix the size of the second dimension of input `U` to 2, call:

```
assert(size(U,2)==2)
```

Specify Real Input

```
assert ( isreal (param ) )
```

Specifies that the input parameter *`param`* is real. To specify that input `U` is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input

```
assert ( ~isreal (param ) )
```

Specifies that the input parameter *`param`* is complex. To specify that input `U` is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the `numerictype` properties of `fi` input parameter *`fiparam`* to the `numerictype` object *`T`*. For example, to specify the `numerictype` property of fixed-point input `U` as a signed `numerictype` object `T` with 32-bit word length and 30-bit fraction length, use the following code:

```
%#codegen
...
% Define the numerictype object.
T = numerictype(1, 32, 30);

% Set the numerictype property of input U to T.
assert(isequal(numerictype(U),T));
...
```

Specifying the `numerictype` for a variable does not automatically specify that the variable is fixed point. You must specify both the `fi` class and the `numerictype`.

Specify fimath of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *`fiparam`* to the `fimath` object *`F`*. For example, to specify the `fimath` property of fixed-point input `U` so that it saturates on integer overflow, use the following code:

```
%#codegen
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

If you do not specify the `fimath` properties using `assert`, `codegen` uses the MATLAB default `fimath` value.

Specify Multiple Properties of Input

```
assert ( function1 ( params ) &&
         function2 ( params ) &&
        function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
%#codegen
...
assert(isa(U,'double') &&
       ~isreal(U) &&
       all(size(U) == [3 3]) &&
       isa(V,'uint16'));
...
```

Rules for Using assert Function

When using the `assert` function to specify the properties of primary function inputs, follow these rules:

- Call `assert` functions at the beginning of the primary function, before control-flow operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- For a fixed-point input, you must specify both the `fi` class and the `numerictype`. See "Specify numerictype of Fixed-Point Input" on page 20-73. You can also set the `fimath` properties. See "Specify fimath of Fixed-Point Input" on page 20-73. If you do not set the `fimath` properties, `codegen` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order that they appear in the structure definition.
- When you use `assert(all(size(`*`param`*`)>=[M0 M1 ...]))` to specify the lower-bound size of each dimension of an input parameter:
  - You must also specify an upper-bound size for each dimension of the input parameter.
  - For each dimension, `k`, the lower-bound `Mk` must be less than or equal to the upper-bound `Nk`.
  - To specify a fixed-size dimension, set the lower and upper bound of a dimension to the same value.
  - Bounds must be nonnegative.

- If you specify individual dimensions, the following rules apply:
  - You must specify the size of each dimension at least once.
  - The last dimension specification takes precedence over earlier specifications.

Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `mcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs.

Input	Property	Value
`pennywhistle`	class	`int16`
	size	220500-by-1 vector
	complexity	`real` (by default)
`win`	class	`double`
	size	1024-by-1 vector
	complexity	`real` (by default)

```
%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...
```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands:

```
%#codegen
function y = mcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...
```

Specifying Properties of Primary Fixed-Point Inputs

To specify fixed-point inputs, you must install Fixed-Point Designer software.

In the following example, the primary MATLAB function `mcsqrtfi` takes one fixed-point input `x`. The code specifies the following properties for this input.

Property	Value
class	`fi`
`numerictype`	`numerictype` object `T`, as specified in the primary function
`fimath`	`fimath` object `F`, as specified in the primary function
size	`scalar`
complexity	`real` (by default)

```
function y = mcsqrtfi(x) %#codegen
T = numerictype('WordLength',32,'FractionLength',23,...
                'Signed',true);
F = fimath('SumMode','SpecifyPrecision',...
           'SumWordLength',32,'SumFractionLength',23,...
           'ProductMode','SpecifyPrecision',...
           'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numerictype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

You must specify both the `fi` class and the `numerictype`.

Specifying Properties of Cell Arrays

To specify the MATLAB class `cell` (cell array), use one of the following syntaxes:

```
assert(iscell(param))
assert(isa( param, 'cell'))
```

For example, to set the class of input `C` to `cell`, use:

```
...
assert(iscell(C));
```

```
...
```

or

```
...
assert(isa(C, 'cell'));
...
```

You can also specify the size of the cell array and the properties of the cell array elements. The number of elements that you specify determines whether the cell array is homogeneous or heterogeneous. See “Code Generation for Cell Arrays” on page 8-2.

If you specify the properties of the first element only, the cell array is homogeneous. For example, the following code specifies that `C` is a 1x3 homogeneous cell array whose elements are 1x1 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1  3]));
assert(isa(C{1}, 'double'));
...
```

If you specify the properties of the first element only, but also assign a structure type name to the cell array, the cell array is heterogeneous. Each element has the properties of the first element. For example, the following code specifies that `C` is a 1x3 heterogeneous cell array. Each element is a 1x1 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1  3]));
assert(isa(C{1}, 'double'));
coder.cstructname(C, 'myname');
...
```

If you specify the properties of each element, the cell array is heterogeneous. For example, the following code specifies a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x3 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1  2]));
assert(isa(C{1}, 'char'));
assert(all(size(C{2}) == [1 3]));
assert(isa(C{2}, 'double'));
```

```
...
```

If you specify more than one element, you cannot specify that the cell array is variable size, even if all elements have the same properties. For example, the following code specifies a variable-size cell array. Because the code specifies the properties of the first and second elements, code generation fails.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1  2]));
assert(isa(C{1}, 'double'));
assert(isa(C{2}, 'double'));
...
```

In the previous example, if you specify the first element only, you can specify that the cell array is variable-size. For example:

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1  2]));
assert(isa(C{1}, 'double'));
...
```

Specifying Class and Size of Scalar Structure

Suppose that you defineS as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

The following code specifies the properties of the function input S and its fields:

```
function y = fcn(S)  %#codegen


% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8');
...
```

In most cases, when you do not explicitly specify values for properties, MATLAB Coder uses defaults—except for structure fields. The only way to name a field in a structure is to set at least one of its properties. At a minimum, you must specify the class of a structure field.

Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume that you have defined S as the following 1-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input S by using the first element of the array:

```
%#codegen
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [1 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. At a minimum, you must specify the class of all fields.

Speed Up Compilation by Generating Only Code

To speed up compilation, you can generate only code. When you generate only code, MATLAB Coder does not invoke the make command or generate compiled object code. When you iterate between modifying MATLAB code and generating C/C++ code, and you want to inspect the generated code, using this option saves time.

To select this option in the MATLAB Coder app:

1. On the **Generate Code** page, click the **Generate** arrow ▼ to open the **Generate** dialog box.
2. Set **Build Type** to `Static Library`, `Dynamic Library`, or `Executable`.
3. Select the **Generate code only** check box.

To set this option at the command line, use the `codegen -c` option. For example, to generate only code for a function `foo`:

```
codegen -c foo
```

See Also

`codegen`

More About

- “Speed Up MEX Generation by Using JIT Compilation” on page 28-63

Disable Creation of the Code Generation Report

If you disable creation of the code generation report, you can speed up code generation, unless an error occurs. If an error occurs, the code generator creates a report even if you disabled creation of the report.

To disable creation of the code generation report:

- In the MATLAB Coder app, in the project build settings, on the **Debugging** tab, clear the **Always create a code generation report** check box.
- At the command line, when you generate code, do not use the `-report` option. If you specify a code configuration object, make sure that the `GenerateReport` property is set to `false`.

By default, creation of the code generation report is disabled.

Related Examples

- "Enable Code Generation Reports" on page 21-27

More About

- "Configure Build Settings" on page 20-26
- "Code Generation Reports" on page 21-9

Paths and File Infrastructure Setup

In this section...

Compile Path Search Order

MATLAB Coder resolves MATLAB functions by searching first on the *code generation path* and then on the MATLAB path. The code generation path contains the current folder and the code generation libraries. By default, unless MATLAB Coder determines that a function should be extrinsic or you explicitly declare the function to be extrinsic, MATLAB Coder tries to compile and generate code for functions it finds on the path. MATLAB Coder does not compile extrinsic functions, but rather dispatches them to MATLAB for execution. See “Resolution of Function Calls for Code Generation” on page 13-2.

Specify Folders to Search for Custom Code

If you want to integrate custom code — such as source, header, and library files — with the generated code, you can specify additional folder to search. The following table describes how to specify these search paths. The path should not contain:

- Spaces (Spaces can lead to code generation failures in certain operating system configurations)
- Tabs
- \, $, #, *, ?
- Non-7-bit ASCII characters, such as Japanese characters

To specify additional folders	Do this
Using the MATLAB Coder app	**1** To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼. **2** Click **More Settings**.

To specify additional folders	Do this
	3 On the **Paths** tab, in the **Search paths** field, either browse to add a folder to the search path or enter the full path. The search path must not contain spaces.
At the command line	Use the `codegen` function `-I` option.

Naming Conventions

MATLAB Coder enforces naming conventions for MATLAB functions and generated files.

- "Reserved Prefixes" on page 20-83
- "Reserved Keywords" on page 20-83
- "Conventions for Naming Generated Files" on page 20-87

Reserved Prefixes

MATLAB Coder reserves the prefix `eml` for global C/C++ functions and variables in generated code. For example, MATLAB for code generation run-time library function names begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C/C++ functions or primary MATLAB functions with the prefix `eml`.

Reserved Keywords

- "C Reserved Keywords" on page 20-84
- "C++ Reserved Keywords" on page 20-84
- "Reserved Keywords for Code Generation" on page 20-85
- "MATLAB Coder Code Replacement Library Keywords" on page 20-86

MATLAB Coder software reserves certain words for its own use as keywords of the generated code language. MATLAB Coder keywords are reserved for use internal to MATLAB Coder software and should not be used in MATLAB code as identifiers or function names. C reserved keywords should also not be used in MATLAB code as identifiers or function names. If your MATLAB code contains reserved keywords that the code generator cannot rename, the code generation build does not complete and an error message is displayed. To address this error, modify your code to use identifiers or names that are not reserved.

If you are generating C++ code using the MATLAB Coder software, in addition, your MATLAB code must not contain the "C++ Reserved Keywords" on page 20-84.

C Reserved Keywords

assert	extern	setjmp	string
auto	fenv	short	struct
break	float	signal	switch
case	for	signed	tgmath
char	goto	sizeof	threads
const	if	static	time
complex	int	stdalign	typedef
continue	inttypes	stdarg	uchar
ctype	iso646	stdatomic	union
default	limits	stdbool	unsigned
do	locale	stddef	void
double	long	stdint	volatile
else	math	stdio	wchar
enum	register	stdlib	wctype
errno	return	stdnoreturn	while

C++ Reserved Keywords

algorithm	cstddef	iostream	sstream
any	cstdint	istream	stack
array	cstdio	iterator	static_cast
atomic	cstdlib	limits	stdexcept
bitset	cstring	list	streambuf
cassert	ctgmath	locale	string_view
catch	ctime	map	strstream
ccomplex	cuchar	memory	system_error

cctype	cwchar	memory_resource	template
cerrno	cwctype	mutable	this
cfenv	delete	mutex	thread
cfloat	deque	namespace	throw
chrono	dynamic_cast	new	try
cinttypes	exception	numeric	tuple
ciso646	execution	operator	typeid
class	explicit	optional	type_traits
climits	export	ostream	typeindex
clocale	filesystem	private	typeinfo
cmath	foreward_list	protected	typename
codecvt	friend	public	unordered_map
complex	fstream	queue	unordered_set
condition_variable	functional	random	using
const_cast	future	ratio	utility
csetjmp	initializer_list	regex	valarray
csignal	inline	reinterpret_cast	vector
cstdalign	iomanip	scoped_allocator	virtual
cstdarg	ios	set	wchar_t
cstdbool	iosfwd	shared_mutex	

Reserved Keywords for Code Generation

abs	fortran	localZCE	rtNaN
asm	HAVESTDIO	localZCSV	SeedFileBuffer
bool	id_t	matrix	SeedFileBufferLen
boolean_T	int_T	MODEL	single
byte_T	int8_T	MT	TID01EQ
char_T	int16_T	NCSTATES	time_T
cint8_T	int32_T	NULL	true

cint16_T	int64_T	NUMST	TRUE
cint32_T	INTEGER_CODE	pointer_T	uint_T
creal_T	LINK_DATA_BUFFER_SIZE	PROFILING_ENABLED	uint8_T
creal32_T	LINK_DATA_STREAM	PROFILING_NUM_SAMPLES	uint16_T
creal64_T	localB	real_T	uint32_T
cuint8_T	localC	real32_T	uint64_T
cuint16_T	localDWork	real64_T	UNUSED_PARAMETER
cuint32_T	localP	RT	USE_RTMODEL
ERT	localX	RT_MALLOC	VCAST_FLUSH_DATA
false	localXdis	rtInf	vector
FALSE	localXdot	rtMinusInf	

MATLAB Coder Code Replacement Library Keywords

The list of code replacement library (CRL) reserved keywords for your development environment varies depending on which CRLs currently are registered. Beyond the default ANSI, ISO, and GNU® CRLs provided with MATLAB Coder software, additional CRLs might be registered and available for use if you have installed other products that provide CRLs (for example, a target product), or if you have used Embedded Coder APIs to create and register custom CRLs.

To generate a list of reserved keywords for the CRLs currently registered in your environment, use the following MATLAB function:

```
crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers()
```

This function returns a cell array of character vectors that contain CRL keywords. Specifying the return argument is optional.

Note: To list the CRLs currently registered in your environment, use the MATLAB command `crviewer`.

To generate a list of reserved keywords for the CRL that you are using to generate code, call the function passing the name of the CRL as displayed in the **Code replacement library** menu on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box. For example,

```
crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

Here is a partial example of the function output:

```
>> crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')

crl_ids =

    'exp10'
    'exp10f'
    'acosf'
    'acoshf'
    'asinf'
    'asinhf'
    'atanf'
    'atanhf'
...
    'rt_lu_cplx'
    'rt_lu_cplx_sgl'
    'rt_lu_real'
    'rt_lu_real_sgl'
    'rt_mod_boolean'
    'rt_rem_boolean'
    'strcpy'
    'utAssert'
```

Note: Some of the returned keywords appear with the suffix `$N`, for example, `'rt_atan2$N'`. `$N` expands into the suffix `_snf` only if nonfinite numbers are supported. For example, `'rt_atan2$N'` represents `'rt_atan2_snf'` if nonfinite numbers are supported and `'rt_atan2'` if nonfinite numbers are not supported. As a precaution, you should treat both forms of the keyword as reserved.

Conventions for Naming Generated Files

The following table describes how MATLAB Coder names generated files. MATLAB Coder follows MATLAB conventions by providing platform-specific extensions for MEX files.

Platform	MEX File Extension	MATLAB Coder Extension for Static Library	MATLAB Coder Extension for Shared Library	MATLAB Coder Executable Extension
Linux (64-bit)	`.mexa64`	`.a`	`.so`	None
Apple Mac (64-bit)	`.mexmaci64`	`.a`	`.dylib`	None

Platform	MEX File Extension	MATLAB Coder Extension for Static Library	MATLAB Coder Extension for Shared Library	MATLAB Coder Executable Extension
Windows (64-bit)	`.mexw64`	`.lib`	`.dll` Also, generates an import library with a `.lib` extension that is required for linking against the `.dll`.	`.exe`

Generate Code for Multiple Entry-Point Functions

In this section...

Advantages of Generating Code for Multiple Entry-Point Functions

Generating a single C/C++ library for more than one entry-point MATLAB function allows you to:

- Create C/C++ libraries containing multiple, compiled MATLAB files to integrate with larger C/C++ applications.
- Share code efficiently between library functions.
- Communicate between library functions using shared memory.

Generating a MEX function for more than one entry-point function allows you to validate entry-point interactions in MATLAB before creating a C/C++ library.

Generate Code for More Than One Entry-Point Function Using the MATLAB Coder App

This example shows how to generate code for multiple entry-point functions using the MATLAB Coder app.

Create the Entry-Point Functions

1. In a local writable folder, create a MATLAB file, `ep1.m`, that contains:

```
function y = ep1(u) %#codegen
y = u;
```

2. In the same local writable folder, create a MATLAB file, `ep2.m`, that contains:

   ```
   function y = ep2(u, v) %#codegen
   y = u + v;
   ```

Create the Test File

In the folder that contains `ep1.m` and `ep2.m`, create a MATLAB file, `ep_test.m`, that calls `ep1` and `ep2` with example inputs.

```
function [y, y1] = ep_test
y = ep1(single(2));
y1 = ep2(double(3), double(4));
```

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

Specify Source Files

1. On the **Select Source Files** page, type or select the name of the entry-point function `ep1`.

   The app creates a project with the default name `ep1.prj` in the current folder.
2. To add `ep2` to the list of entry-point functions, click **Add Entry-Point Function**. Type or select the name of the entry-point function `ep2`.
3. Click **Next** to go to the **Define Input Types** step. The app analyzes the functions for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, MATLAB Coder must determine the properties of all variables in the MATLAB files. You must specify the properties of all entry-point function inputs. From the properties of the entry-point function inputs, MATLAB Coder can infer the properties of all variables in the MATLAB files.

Specify a test file that MATLAB Coder can use to automatically define types:

1. Enter or select the test file `ep_test.m`.

2. Click **Autodefine Input Types**.

   The test file, `ep_test.m`, calls the entry-point functions `ep1` and `ep2` with the example input types. MATLAB Coder infers that for `ep1`, input `u` is `single(1x1)`. For `ep2`, `u` and `v` are `double(1x1)`.
3. Click **Next** to go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow ▼.

   The app populates the test file field with `ep_test`, the test file that you used to define the input types.
2. Click **Check for Issues**.

   The app generates a MEX function named `ep1_mex` for `ep1` and `ep2`. It runs the test file `ep_test` replacing calls to `ep1` and `ep2` with calls to the MEX function. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.
3. Click **Next** to go to the **Generate Code** step.

Generate MEX Function

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. Set **Build type** to `MEX`.
3. Verify that the **Output file name** is `ep1_mex`. By default, the app uses the name of the alphabetically first entry-point function.
4. Click **Generate**.

   MATLAB Coder builds the project. It generates a MEX function, `ep1_mex`, in the current folder. MATLAB Coder also generates other supporting files in a subfolder called `codegen/mex/ep1_mex`. MATLAB Coder uses the name of the MATLAB function as the root name for the generated files. It creates a platform-specific extension for the MEX file, as described in "Naming Conventions" on page 20-83.

You can now test your MEX function in MATLAB. See "How to Call an Entry-Point Function in a MEX Function" on page 20-93.

If you generate a static library for `ep1` and `ep2`, MATLAB Coder builds the project and generates a C library, `ep1`, and supporting files in the default folder, `codegen/lib/ep1`.

Generating Code for More Than One Entry-Point Function at the Command Line

To generate code for more than one entry-point function, use the following syntax, where `global_options` applies to functions, `fun_1` through `fun_n`, and `options_n` applies only to the preceding function `fun_n`.

```
codegen -global_options fun_1 -options_1 ... fun_n -options_n
```

By default, `codegen`:

- Generates a MEX function in the current folder. `codegen` names the MEX function, *`fun`*`_mex`. *`fun`* is the name of the alphabetically first entry-point function.

  Stores generated files in the subfolder `codegen/mex/`*`fun_1`*. *`fun_1`* is the name of the first entry-point function.

You can specify the output file name and subfolder name using the `-o` option.

```
codegen -o out_fun  fun_1 -options_1 ... fun_n -options_n
```

In this case, `codegen`:

- Generates a MEX function named `out_fun_mex` in the current folder.
- Stores generated files in the subfolder `codegen/mex/out_fun`.

For more information on setting build options at the command line, see `codegen`.

Generating a MEX Function with Two Entry-Point Functions at the Command Line

Generate a MEX function with two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Using the `-o` option, name the generated MEX function `sharedmex`.

```
codegen -o sharedmex ep1 -args single(0) ep2 -args { 0, zeros(1,1024) }
```

`codegen` generates a MEX function named `sharedmex.mex` in the current folder and stores generated files in the subfolder `codegen/mex/sharedmex`.

Generating a C/C++ Static Library with Two Entry-Point Functions at the Command Line

Generate standalone C/C++ code and compile it to a library for two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Use the `-config:lib` option to specify that the target is a library. Using the `-o` option, name the generated library `sharedlib`.

```
codegen -config:lib -o sharedlib ep1 -args single(0) ep2 ...
   -args { 0, zeros(1,1024) }
```

`codegen` generates C/C++ library code in the `codegen/lib/sharedlib` folder.

For information on viewing entry-point functions in the code generation report, see “Code Generation Reports” on page 21-9.

How to Call an Entry-Point Function in a MEX Function

To call an entry-point function in a MEX function that has more than one entry point, use this syntax:

```
MEX_Function('entry_point_function_name',
         ... entry_point_function_param1,
         ... , entry_point_function_paramn)
```

Calling an Entry-Point Function in a MEX Function

Consider a MEX function, `sharedmex`, that has entry-point functions `ep1` and `ep2`. Entry-point function `ep1` takes one single scalar input and `ep2` takes two inputs, a double scalar and a double vector.

To call `ep1` with an input parameter `u`, enter:

```
sharedmex('ep1', u)
```

To call `ep2` with input parameters `u` and `v`, enter:

```
sharedmex('ep2', u, v)
```

How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code

To call an entry-point function in a C/C++ library function from C/C++ code, write a `main` function in C/C++ that:

- Includes the generated header files, which contain the function prototypes for the entry-point functions.
- Calls the initialize function before calling the entry-point functions for the first time.
- Calls the terminate function after calling the entry-point functions for the last time.
- Configures your target to integrate this custom C/C++ main function with your generated code, as described in “Specify External File Locations” on page 24-14.
- Generates the C/C++ executable using `codegen`.

See the example, “Call a Generated C Static Library Function from C Code” on page 24-4.

Generate Code for Global Data

In this section...

Workflow

To generate C/C++ code from MATLAB code that uses global data:

1. Declare the variables as global in your code.
2. Before using the global data, define and initialize it.

   For more information, see "Define Global Data" on page 20-96.
3. Generate code using the MATLAB Coder app or using `codegen`.

If you use global data, you must also specify whether you want to synchronize this data between MATLAB and the generated MEX function. For more information, see "Synchronizing Global Data with MATLAB" on page 20-97.

Declare Global Variables

When using global data, you must first declare the global variables in your MATLAB code. Consider the `use_globals` function that uses two global variables AR and B:

```
function y = use_globals(u)
%#codegen
% Turn off inlining to make
% generated code easier to read
coder.inline('never');
% Declare AR and B as global variables
global AR;
global B;
AR(1) = u + B(1);
```

```
y = AR * 2;
```

Define Global Data

You can define global data in the MATLAB global workspace, in a MATLAB Coder project, or at the command line. If you do not initialize global data in the project or at the command line, MATLAB Coder looks for the variable in the MATLAB global workspace. If the variable does not exist, MATLAB Coder generates an error.

Defining Global Data in the MATLAB Global Workspace

To generate a MEX function for the `use_globals` function described in “Declare Global Variables” on page 20-95 using `codegen`:

1. In the MATLAB workspace, define and initialize the global data. At the MATLAB prompt, enter:

   ```
   global AR B;
   AR = ones(4);
   B = [1 2 3];
   ```

2. Generate a MEX file.

   ```
   codegen use_globals -args {0}
   % Use the -args option to specify that the input u
   % is a real, scalar, double
   % By default, codegen generates a MEX function,
   % use_globals_mex, in the current folder
   ```

Defining Global Data Using the MATLAB Coder App

1. On the **Define Input Types** page, automatically define input types or click **Let me enter input or global types directly**.

   The app displays a table of entry-point inputs.
2. To add a global variable, click **Add global**.

   By default, the app names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, and so on.
3. Under **Global variables**, enter a name for the global variable.
4. Click the field to the right of the global variables name. Specify the type and initial value of the global variable. See “Specify Global Variable Type and Initial Value Using the App” on page 17-30.

If you do not specify the type, you must create a variable with the same name in the global workspace.

Defining Global Data at the Command Line

To define global data at the command line, use the `codegen -globals` option. For example, to compile the `use_globals` function described in “Declare Global Variables” on page 20-95, specify two global inputs `AR` and `B` at the command line. Use the `-args` option to specify that the input `u` is a real, scalar double. By default, codegen generates a MEX function, `use_globals_mex`, in the current folder.

```
codegen -globals {'AR',ones(4),'B',[1 2 3]} use_globals -args {0}
```

Alternatively, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For cell arrays, you must use this format. See “Specify Global Cell Arrays at the Command Line” on page 20-105.

Defining Variable-Size Global Data

To provide initial values for variable-size global data, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For example, to specify a global variable `g1` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
codegen foo -globals {'g1', {coder.typeof(0, [2 2],1),[1 1]}}
```

For a detailed explanation of the syntax, see `coder.typeof`.

Synchronizing Global Data with MATLAB

Why Synchronize Global Data?

The generated MEX function and MATLAB each have their own copies of global data. To make these copies consistent, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ. The level of interaction determines when to synchronize global data. For more information, see “When to Synchronize Global Data” on page 20-98.

When global data is constant, you cannot synchronize the global data with MATLAB. By default, the MEX function tests for consistency between the compile-time constant global values and the MATLAB values at function entry and after extrinsic function calls. If the MATLAB values differ from the compile-time constant global values, the MEX function ends with an error. For information about controlling when the MEX function tests for consistency between the compile-time constant global values and the MATLAB values, see “Consistency Between MATLAB and Constant Global Data” on page 20-103.

When to Synchronize Global Data

By default, synchronization between the MEX function's global data and MATLAB occurs at MEX function entry and exit and for extrinsic calls. Use this synchronization method for maximum consistency between the MEX function and MATLAB.

To improve performance, you can:

- Select to synchronize only at MEX function entry and exit points.
- Disable synchronization when the global data does not interact.
- Choose whether to synchronize before and after each extrinsic call.

The following table summarizes which global data synchronization options to use. To learn how to set these options, see “How to Synchronize Global Data” on page 20-99.

Global Data Synchronization Options

If you want to	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Have maximum consistency when all extrinsic calls modify global data.	`At MEX-function entry, exit and extrinsic calls` (default)	Yes. Default behavior.
Have maximum consistency when most extrinsic calls modify global data, but a few do not.	`At MEX-function entry, exit and extrinsic calls` (default)	Yes. Use the `coder.extrinsic -sync:off` option to turn off synchronization for the extrinsic calls that do not change global data.
Have maximum consistency when most extrinsic calls do not modify global data, but a few do.	`At MEX-function entry and exit`	Yes. Use the `coder.extrinsic -sync:on` option to synchronize only the calls that modify global data.
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	`At MEX-function entry and exit`	No.
Communicate between generated MEX functions only. No interaction between MATLAB and MEX function global data.	`Disabled`	No.

How to Synchronize Global Data

To control global data synchronization, set the global data synchronization mode and select whether to synchronize extrinsic functions. For guidelines on which options to use, see “When to Synchronize Global Data” on page 20-98.

You can control the global data synchronization mode from the project settings dialog box, the command line, or a MEX configuration dialog box. You control the synchronization of data with extrinsic functions using the `coder.extrinsic -sync:on` and `-sync:off` options.

Controlling the Global Data Synchronization Mode Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `MEX`.
3. Click **More Settings**.
4. On the **Memory** tab, set **Global data synchronization mode** to `At MEX-function entry and exit` or `Disabled`, as applicable.

Controlling the Global Data Synchronization Mode from the Command Line

1. In the MATLAB workspace, define the code generation configuration object. At the MATLAB command line, enter:

   ```
   mexcfg = coder.config('mex');
   ```
2. At the MATLAB command line, set the `GlobalDataSyncMethod` property to `SyncAtEntryAndExits` or `NoSync`, as applicable. For example:

   ```
   mexcfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
   ```
3. When compiling your code, use the `mexcfg` configuration object. For example, to generate a MEX function for function `foo` that has no inputs:

   ```
   codegen -config mexcfg foo
   ```

Controlling Synchronization for Extrinsic Function Calls

To control whether synchronization between MATLAB and MEX function global data occurs before and after you call an extrinsic function, use the `coder.extrinsic-sync:on` and `-sync:off` options.

By default, global data is:

- Synchronized before and after each extrinsic call, if the global data synchronization mode is `At MEX-function entry, exit and extrinsic calls`. If you are sure that certain extrinsic calls do not change global data, turn off synchronization for these calls using the `-sync:off` option. For example, if functions `foo1` and `foo2` do not change global data, turn off synchronization for these functions:

  ```
  coder.extrinsic('-sync:off', 'foo1', 'foo2');
  ```
- Not synchronized, if the global data synchronization mode is `At MEX-function entry and exit`. If the code has a few extrinsic calls that change global data,

turn on synchronization for these calls using the `-sync:on` option. For example, if functions `foo1` and `foo2` change global data, turn on synchronization for these functions:

```
coder.extrinsic('-sync:on', 'foo1', 'foo2');
```

- Not synchronized, if the global data synchronization mode is `Disabled`. When synchronization is disabled, you cannot use the `-sync:on` option to control the synchronization for specific extrinsic calls.

Define Constant Global Data

If you know that the value of a global variable does not change at run time, you can reduce overhead in the generated code by specifying that the global variable has a constant value. You cannot write to the constant global variable.

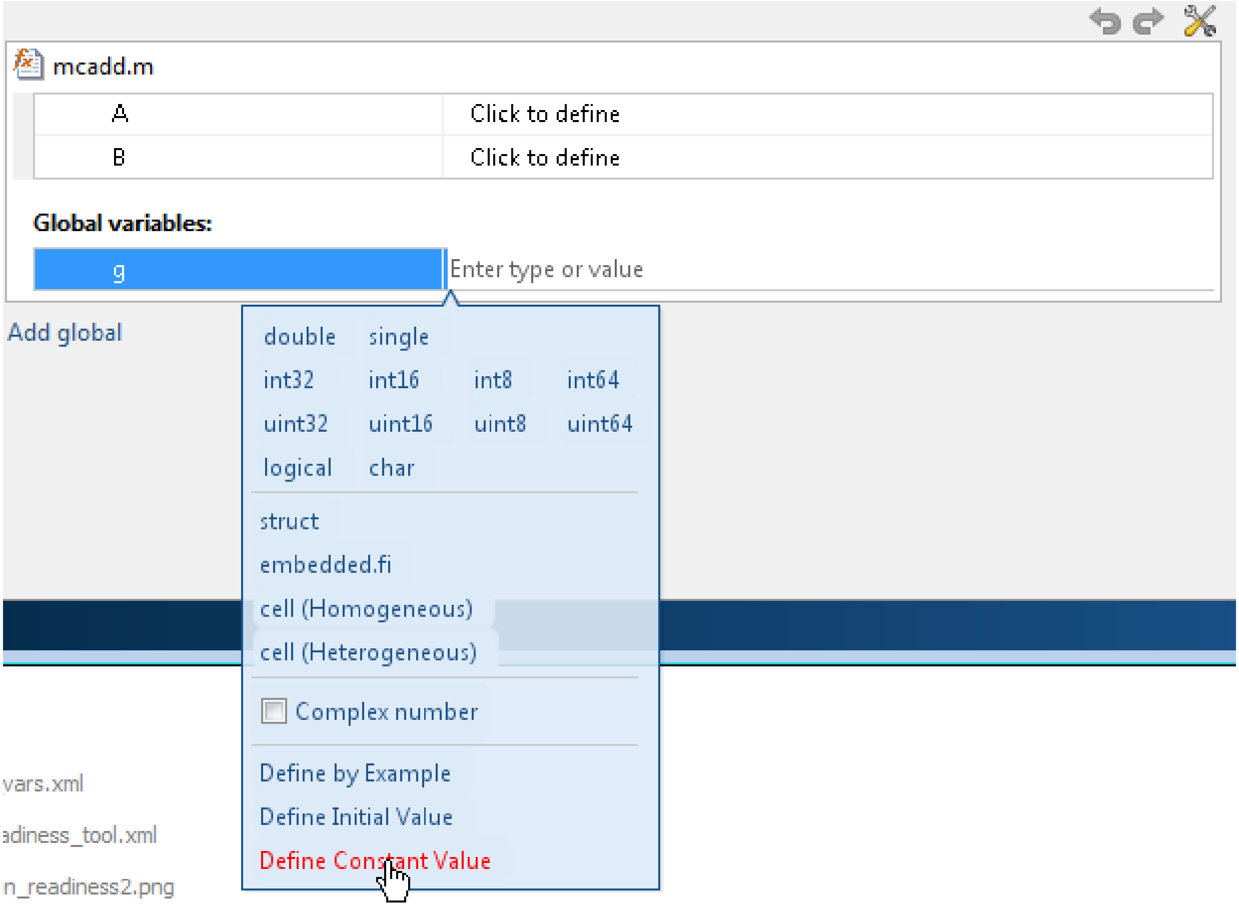
Define Constant Global Data Using the MATLAB Coder App

- On the **Define Input Types** page, automatically define input types or click **Let me enter input or global types directly**.

  The app displays a table of entry-point inputs.

1. To add a global variable, click **Add global**.

   By default, the app names the first global variable in a project `g`, and subsequent global variables `g1`, `g2`, and so on.
2. Under **Global Variables**, enter a name for the global variable.
3. Click the field to the right of the global variable name.
4. Select `Define Constant Value`.



5 In the field to the right of the global variable, enter a MATLAB expression.

Define Constant Global Data at the Command Line

To specify that a global variable is constant using the `codegen` command, use the `-globals` option with the `coder.Constant` class.

1 Define a configuration object for the code generation output type that you want. For example, define a configuration object for MEX code generation:

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

2. Use `coder.Constant` to specify that a global variable has a constant value. For example, the following code specifies that the global variable `g` has initial value `4` and that global variable `gc` has the constant value `42`.

   ```
   global_values = {'g', 4, 'gc', coder.Constant(42)};
   ```

3. Generate the code using the `-globals` option. For example, generate code for `myfunction` specifying that the global variables are defined in the cell array `global_values`.

   ```
   codegen -config cfg -globals global_values myfunction
   ```

Consistency Between MATLAB and Constant Global Data

By default, the generated MEX function verifies that the values of constant global data in the MATLAB workspace are consistent with the compile-time values in the generated MEX. It tests for consistency at function entry and after calls to extrinsic functions. If the MEX function detects an inconsistency, it ends with an error. To control when the MEX function tests for consistency, use the global synchronization mode and the `coder.extrinsic` synchronization options.

The following table shows how the global data synchronization mode and the `coder.extrinsic` synchronization option setting determine when a MEX function verifies consistency between the compile-time constant global data values and MATLAB.

Global Data Synchronization Mode (Project)	GlobalDataSyncMethod (MEX Configuration Object)	Verify Consistency of Constant Global Values at MEX Function Entry	coder.extrinsic synchronization option	Verify Consistency of Constant Global Values After Extrinsic Function Call
`At MEX-function entry, exit and extrinsic calls` (default)	`'SyncAlways'`	yes	`'sync:on'` (default)	yes
			`'sync:off'`	no
`At MEX-function entry and exit`	`'SyncAtEntryAndExits'`	yes	`'sync:on'`	yes
			`'sync:off'` (default)	no
`Disabled`	`'NoSync'`	no	N/A	N/A

Constant Global Data in a Code Generation Report

The code generation report provides the following information about a constant global variable:

- Type of `Global` on the **Variables** tab.
- Highlighted variable name in the **Function** pane.

See "MATLAB Code Variables in a Report" on page 21-17.

Limitations of Using Global Data

You cannot use global data with the `coder.cstructname` function.

Specify Global Cell Arrays at the Command Line

To specify global cell array inputs, use the `-globals` option of the `codegen` command with this syntax:

```
codegen myfunction -globals {global_var, {type_object, initial_value}}
```

For example:

- To specify that the global variable `g` is a 1x3 cell array whose elements have class double and whose initial value is `{1 2 3}`, use:

  ```
  codegen myfunction -globals {'g', {coder.typeof({1 1 1}), {1 2 3}}}
  ```

  Alternatively, use:

  ```
  t = coder.typeof({1 1 1});
  codegen myfunction -globals {'g', {t, {1 2 3}}}
  ```

  The global variable `g` is a 1x3 homogeneous cell array whose elements are 1x1 double.

  To make `g` heterogeneous, use:

  ```
  t = makeHeterogeneous(coder.typeof({1 1 1}));
  codegen myfunction -globals {'g', {t, {1 2 3}}}
  ```
- To specify that `g` is a cell array whose first element has type char, whose second element has type double, and whose initial value is `{'a', 1}`, use:

  ```
  codegen myfunction -globals {'g', {coder.typeof({'a', 1}), {'a', 1}}}
  ```

  The global variable `g` is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.
- To specify that `g` is a cell array whose first element has type double, whose second element is a 1x2 double array, and whose initial value is `{1 [2 3]}`, use:

  ```
  codegen myfunction -globals {'g', {coder.typeof({1 [2 3]}), {1 [2 3]}}}
  ```

  Alternatively, use:

  ```
  t = coder.typeof({1 [2 3]});
  codegen myfunction -globals {'g', {t, {1 [2 3]}}}
  ```

> The global variable g is a 1x2 heterogeneous cell array whose first element is 1x1 double and whose second element is 1x2 double.

Global variables that are cell arrays cannot have variable size.

See Also

codegen | coder.typeof

Related Examples

- "Generate Code for Global Data" on page 20-95

Generate Code for Enumerations

The basic workflow for generating code for enumerated types in MATLAB code is:

1. Define an enumerated data type that derives from one of these base types: `int8`, `uint8`, `int16`, `uint16`, or `int32`.
2. Save the enumerated data type in a file on the MATLAB path.
3. Write a MATLAB function that uses the enumerated type.
4. Specify enumerated type inputs.
5. Generate code.

More About

- "Code Generation for Enumerations" on page 9-2
- "Generate Code for an LED Control Function That Uses Enumerated Types" on page 20-188
- "Customize Enumerated Types in Generated Code" on page 9-6
- "Specify an Enumerated Type Input Parameter by Example" on page 17-13
- "Specify an Enumerated Type Input Parameter by Type" on page 17-18

Generate Code for Variable-Size Data

In this section...

Variable-size data is data whose size might change at run time. You can use MATLAB Coder to generate C/C++ code from MATLAB code that uses variable-size data. MATLAB supports bounded and unbounded variable-size data for code generation. *Bounded variable-size data* has fixed upper bounds. This data can be allocated statically on the stack or dynamically on the heap. *Unbounded variable-size data* does not have fixed upper bounds. This data must be allocated on the heap. By default, for MEX and C/C++ code generation, support for variable-size data is enabled and dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to a configurable threshold.

Disable Support for Variable-Size Data

By default, for MEX and C/C++ code generation, support for variable-size data is enabled. You modify variable sizing settings from the project settings dialog box, the command line, or using dialog boxes.

Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Memory** tab, select or clear **Enable variable-sizing**.

At the Command Line

1. Create a configuration object for code generation. For example, for a library:

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

2 Set the `EnableVariableSizing` option:

```
cfg.EnableVariableSizing = false;
```

3 Using the `-config` option, pass the configuration object to `codegen` :

```
codegen -config cfg foo
```

Control Dynamic Memory Allocation

By default, dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to a configurable threshold. If you disable support for variable-size data (see “Disable Support for Variable-Size Data” on page 20-108), you also disable dynamic memory allocation. You can modify dynamic memory allocation settings from the project settings dialog box or the command line.

Using the MATLAB Coder App

1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.

2 Click **More Settings**.

3 On the **Memory** tab, set **Dynamic memory allocation** to one of the following options:

Setting	Action
`Never`	Dynamic memory allocation is disabled. Variable-size data is allocated statically on the stack.
`For all variable-sized arrays`	Dynamic memory allocation is enabled for variable-size arrays. Variable-size data is allocated dynamically on the heap.
`For arrays with max size at or above threshold`	Dynamic memory allocation is enabled for variable-size arrays whose size is greater than or equal to the **Dynamic memory allocation threshold**. Variable-size arrays whose size is less

Setting	Action
	than this threshold are allocated on the stack.

4 Optionally, if you set **Dynamic memory allocation** to `For arrays with maximum size at or above threshold`, configure **Dynamic memory allocation threshold** to fine-tune memory allocation.

At the Command Line

1 Create a configuration object for code generation. For example, for a MEX function:

```
mexcfg = coder.config('mex');
```

2 Set the `DynamicMemoryAllocation` option:

Setting	Action
`mexcfg.DynamicMemoryAllocation='Off';`	Dynamic memory allocation is disabled. Variable-size data is allocated statically on the stack.
`mexcfg.DynamicMemoryAllocation='AllVariableSizeArrays';`	Dynamic memory allocation is enabled for variable-size arrays. Variable-size data is allocated dynamically on the heap.
`mexcfg.DynamicMemoryAllocation='Threshold';`	Dynamic memory allocation is enabled for variable-size arrays whose size (in bytes) is greater than or equal to the value specified using the `Dynamic memory allocation threshold` parameter. Variable-size arrays whose size is less than this threshold are allocated on the stack.

3 Optionally, if you set `Dynamic memory allocation` to `‘Threshold’`, configure `Dynamic memory allocation threshold` to fine tune memory allocation.

4 Using the `-config` option, pass the configuration object to `codegen`:

```
codegen -config mexcfg foo
```

Generating Code for MATLAB Functions with Variable-Size Data

Here is a basic workflow that first generates MEX code for verifying the generated code and then generates standalone code after you are satisfied with the result of the prototype.

To work through these steps with a simple example, see "Generate Code for a MATLAB Function That Expands a Vector in a Loop" on page 20-112

1. In the MATLAB Editor, add the compilation directive `%#codegen` at the top of your function.

   This directive:

   - Indicates that you intend to generate code for the MATLAB algorithm
   - Turns on checking in the MATLAB Code Analyzer to detect potential errors during code generation

2. Address issues detected by the Code Analyzer.

   In some cases, the MATLAB Code Analyzer warns you when your code assigns data a fixed size but later grows the data, such as by assignment or concatenation in a loop. If that data is supposed to vary in size at run time, you can ignore these warnings.

3. Generate a MEX function using `codegen` to verify the generated code. Use the following command-line options:

   - `-args {coder.typeof...}` if you have variable-size inputs
   - `-report` to generate a code generation report

   For example:

   ```
   codegen -report foo -args {coder.typeof(0,[2 4],1)}
   ```
   This command uses `coder.typeof` to specify one variable-size input for function `foo`. The first argument, `0`, indicates the input data type (`double`) and complexity (`real`). The second argument, `[2 4]`, indicates the size, a matrix with two dimensions. The third argument, `1`, indicates that the input is variable sized. The upper bound is 2 for the first dimension and 4 for the second dimension.

Note: During compilation, `codegen` detects variables and structure fields that change size after you define them, and reports these occurrences as errors. In addition, `codegen` performs a run-time check to generate errors when data exceeds upper bounds.

4 Fix size mismatch errors:

Cause	How To Fix	For More Information
You try to change the size of data after its size has been locked.	Declare the data to be variable sized.	See “Diagnosing and Fixing Size Mismatch Errors” on page 6-20.

5 Fix upper bounds errors

Cause	How To Fix	For More Information
MATLAB cannot determine or compute the upper bound	Specify an upper bound.	See“Specify Upper Bounds for Variable-Size Arrays” on page 6-7 and “Diagnosing and Fixing Size Mismatch Errors” on page 6-20.
MATLAB attempts to compute an upper bound for unbounded variable-size data.	If the data is unbounded, enable dynamic memory allocation.	See “Control Dynamic Memory Allocation” on page 20-109.

6 Generate C/C++ code using the `codegen` function.

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

About the MATLAB Function myuniquetol

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

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

Step 1: Add Compilation Directive for Code Generation

Add the %#codegen compilation directive at the top of the function:

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

Step 2: Address Issues Detected by the Code Analyzer

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

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

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

Step 3: Generate MEX Code

It is a best practice to generate MEX code before you generate C/C++ code. Generating MEX code can identify code generation issues that are harder to detect at run time.

1 Generate a MEX function for `myuniquetol`:

```
codegen -report myuniquetol -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 `myuniquetol`:

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

  For more information, see "Specify Variable-Size Inputs at the Command Line" on page 20-54.
- The second argument, `coder.typeof(0)`, defines input `tol` as a real double scalar.

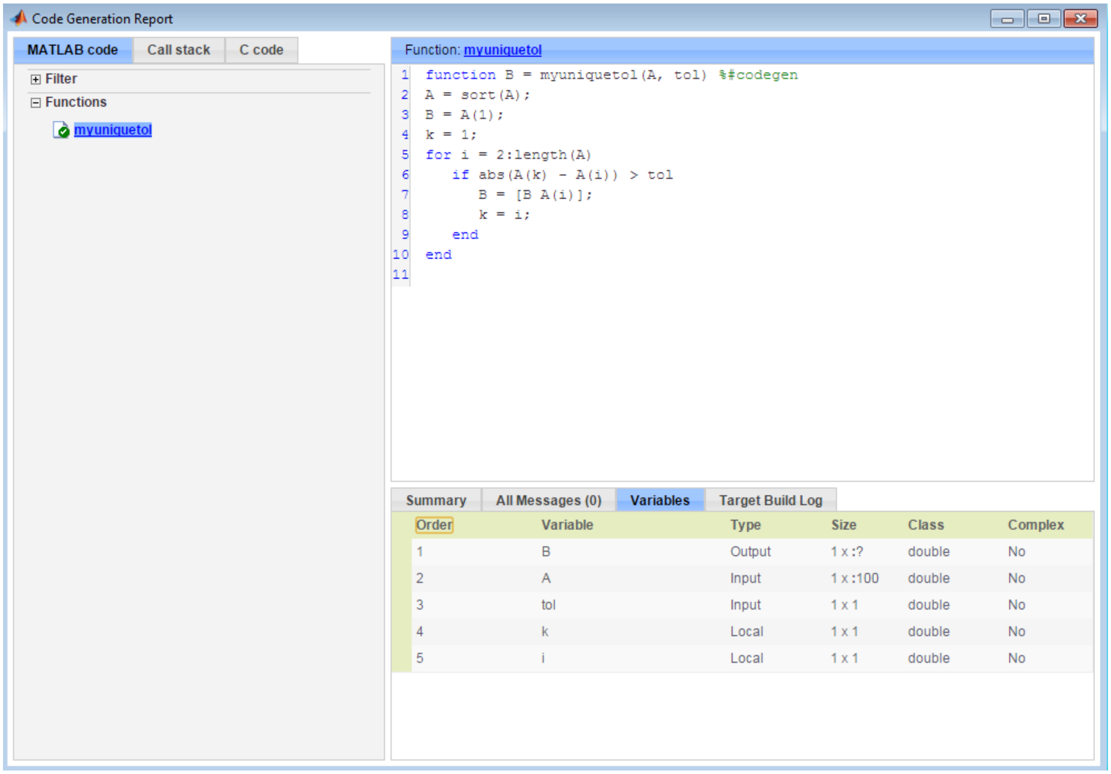
The `-report` option instructs `codegen` to generate a code generation report, regardless of whether errors or warnings occur.

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

Code generation is successful. `codegen` does not detect issues. In the current folder, `codegen` generates a MEX function for `myuniquetol` and provides a link to the code generation report.

2 Click the **View report** link.

3 In the code generation report, select the **Variables** tab.



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

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

The size of A is 1x:100 because you specified that A is variable size with an upper bound of 100. The size of variable B is 1x:?, indicating that it is variable size with no upper bounds.

Step 4: Generate C Code

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

1 Create a configuration option for C library generation:

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

2 Issue this command:

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

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

3 Click the **View report** link.

4 In the code generation report, click the **C code** tab.

5 On the **C code** tab, click the link to myuniquetol.h.

The function declaration is:

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

codegen computes the size of A and, because its maximum size is less than the default dynamic memory allocation threshold of 64k bytes, allocates this memory statically. The generated code contains:

- double A_data[]: the definition of A.
- int A_size[2]: the actual size of the input.

The code generator determines that B is variable size with unknown upper bounds. It represents B as emxArray_real_T. MATLAB provides utility functions for creating and interacting with emxArrays in your generated code. For more information, see "C Code Interface for Arrays" on page 6-15.

Step 5: Specify an Upper Bound for the Output Vector

You specified that the input A is variable size with an upper bound of 100. Therefore, you know that the output B cannot be larger than 100 elements.

- Use coder.varsize to indicate that B is variable size with an upper bound of 100.

```
function B = myuniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 100], [0 1]);
B = A(1);
k = 1;
for i = 2:length(A)
   if abs(A(k) - A(i)) > tol
```

```
         B = [B A(i)];
         k = i;
      end
   end
```

- Generate code.

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

The function declaration is:

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

The code generator statically allocates the memory for B. It stores the size of B in int B_size[2].

Step 6: Change the Dynamic Memory Allocation Threshold

In this step, you reduce the dynamic memory allocation threshold and generate code for an input that exceeds this threshold. This step specifies that the second dimension of A has an upper bound of 10000.

1 Change the upper bound of B to match the upper bound of A.

```
function B = myuniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 10000], [0 1]);
B = A(1);
k = 1;
for i = 2:length(A)
   if abs(A(k) - A(i)) > tol
      B = [B A(i)];
      k = i;
   end
end
```

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

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

3 View the generated code in the report. Because the maximum size of A and B now exceed the dynamic memory allocation threshold, codegen allocates

A and B dynamically on the heap. In the generated code, A and B have type emxArray_real_T.

```
extern void myuniquetol(const emxArray_real_T *A, ...
  double tol, emxArray_real_T *B);
```

Using Dynamic Memory Allocation for an "Atoms" Simulation

This example shows how to generate code for a MATLAB algorithm that runs a simulation of bouncing "atoms" and returns the result after a number of iterations. There are no upper bounds on the number of atoms that the algorithm accepts, so this example takes advantage of dynamic memory allocation.

Prerequisites

There are no prerequisites for this example.

Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will contain only the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_atoms');
```

About the 'run_atoms' Function

The run_atoms.m function runs a simulation of bouncing atoms (also applying gravity and energy loss).

```
help run_atoms
```

```
  atoms = run_atoms(atoms,n)
  atoms = run_atoms(atoms,n,iter)
  Where 'atoms' the initial and final state of atoms (can be empty)
        'n' is the number of atoms to simulate.
        'iter' is the number of iterations for the simulation
           (if omitted it is defaulted to 3000 iterations.)
```

Set Up Code Generation Options

Create a code generation configuration object

```
cfg = coder.config;
% Enable dynamic memory allocation for variable size matrices.
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

Set Up Example Inputs

Create a template structure 'Atom' to provide the compiler with the necessary information about input parameter types. An atom is a structure with four fields (x,y,vx,vy) specifying position and velocity in Cartesian coordinates.

```
atom = struct('x', 0, 'y', 0, 'vx', 0, 'vy', 0);
```

Generate a MEX Function for Testing

Use the command 'codegen' with the following arguments:

'-args {coder.typeof(atom, [1 Inf]),0,0}' indicates that the first argument is a row vector of atoms where the number of columns is potentially infinite. The second and third arguments are scalar double values.

'-config cfg' enables dynamic memory allocation, defined by workspace variable cfg

```
codegen run_atoms -args {coder.typeof(atom, [1 Inf]),0,0} -config cfg -o run_atoms_mex
```

Run the MEX Function

The MEX function simulates 10000 atoms in approximately 1000 iteration steps given an empty list of atoms. The return value is the state of all the atoms after simulation is complete.

```
atoms = repmat(atom,1,0);
atoms = run_atoms_mex(atoms,10000,1000)
```

```
Iteration: 50
Iteration: 100
Iteration: 150
Iteration: 200
Iteration: 250
Iteration: 300
Iteration: 350
```

```
Iteration: 400
Iteration: 450
Iteration: 500
Iteration: 550
Iteration: 600
Iteration: 650
Iteration: 700
Iteration: 750
Iteration: 800
Iteration: 850
Iteration: 900
Iteration: 950
Iteration: 1000
Completed iterations: 1000

atoms = 

  1×10000 struct array with fields:

    x
    y
    vx
    vy
```

Run the MEX Function Again

Continue the simulation with another 500 iteration steps

```
atoms = run_atoms_mex(atoms,10000,500)
```

```
Iteration: 50
Iteration: 100
Iteration: 150
Iteration: 200
Iteration: 250
Iteration: 300
Iteration: 350
Iteration: 400
Iteration: 450
Iteration: 500
Completed iterations: 500

atoms = 
  1×10000 struct array with fields:

    x
    y
    vx
    vy
```

Generate a Standalone C Code Library

To generate a C library, create a standard configuration object for libraries:

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

Enable dynamic memory allocation

```
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

In MATLAB the default data type is double. However, integers are usually used in C code, so pass int32 integer example values to represent the number of atoms and iterations.

```
codegen run_atoms -args {coder.typeof(atom, [1 Inf]),int32(0),int32(0)} -config cfg
```

Inspect Generated Code

When creating a library the code is generated in the folder codegen/lib/run_atoms/ The code in this folder is self contained. To interface with the compiled C code you need only the generated header files and the library file.

```
dir codegen/lib/run_atoms
```

```
.                          rt_nonfinite.h             run_atoms_emxutil.obj
..                         rt_nonfinite.obj           run_atoms_initialize.c
buildInfo.mat              rtw_proj.tmw               run_atoms_initialize.h
codeInfo.mat               rtwtypes.h                 run_atoms_initialize.obj
examples                   run_atoms.c                run_atoms_ref.rsp
interface                  run_atoms.h                run_atoms_rtw.bat
rtGetInf.c                 run_atoms.lib              run_atoms_rtw.mk
rtGetInf.h                 run_atoms.obj              run_atoms_terminate.c
rtGetInf.obj               run_atoms_emxAPI.c         run_atoms_terminate.h
rtGetNaN.c                 run_atoms_emxAPI.h         run_atoms_terminate.obj
rtGetNaN.h                 run_atoms_emxAPI.obj       run_atoms_types.h
rtGetNaN.obj              run_atoms_emxutil.c       target
rt_nonfinite.c            run_atoms_emxutil.h
```

Write a C Main Function

Typically, the main function is platform-dependent code that performs rendering or some other processing. In this example, a pure ANSI-C function produces a file 'run_atoms_state.m' which (when run) contains the final state of the atom simulation.

```
type run_atoms_main.c
```

```
/* Include standard C libraries */
#include <stdio.h>

/* The interface to the main function we compiled. */
#include "codegen/exe/run_atoms/run_atoms.h"

/* The interface to EMX data structures. */
#include "codegen/exe/run_atoms/run_atoms_emxAPI.h"

int main(int argc, char **argv)
{
    FILE *fid;
    int i;
    emxArray_Atom *atoms;

    /* Main arguments unused */
    (void) argc;
    (void) argv;

    /* Initially create an empty row vector of atoms (1 row, 0 columns) */
    atoms = emxCreate_Atom(1, 0);

    /* Call the function to simulate 10000 atoms in 1000 iteration steps */
    run_atoms(atoms, 10000, 1000);

    /* Call the function again to do another 500 iteration steps */
    run_atoms(atoms, 10000, 500);

    /* Print the result to a file */
    fid = fopen("atoms_state.txt", "w");
    for (i = 0; i < atoms->size[1]; i++) {
        fprintf(fid, "%f %f %f %f\n",
            atoms->data[i].x, atoms->data[i].y, atoms->data[i].vx, atoms->data[i].vy);
    }

    /* Close the file */
    fclose(fid);

    /* Free memory */
    emxDestroyArray_Atom(atoms);
    return(0);
}
```

Create a Configuration Object for Executables

```
cfg = coder.config('exe');
cfg.DynamicMemoryAllocation = 'AllVariableSizeArrays';
```

Generate a Standalone Executable

You must pass the function (run_atoms.m) as well as custom C code (run_atoms_main.c) The 'codegen' command automatically generates C code from the MATLAB code, then calls the C compiler to bundle this generated code with the custom C code (run_atoms_main.c).

```
codegen run_atoms run_atoms_main.c -args {coder.typeof(atom, [1 Inf]),int32(0),int32(0
```

Run the Executable

After simulation is complete, this produces the file 'atoms_state.txt'. The TXT file is a 10000x4 matrix, where each row is the position and velocity of an atom (x, y, vx, vy) representing the current state of the whole system.

```
system(['.' filesep 'run_atoms']);
```

Fetch the State

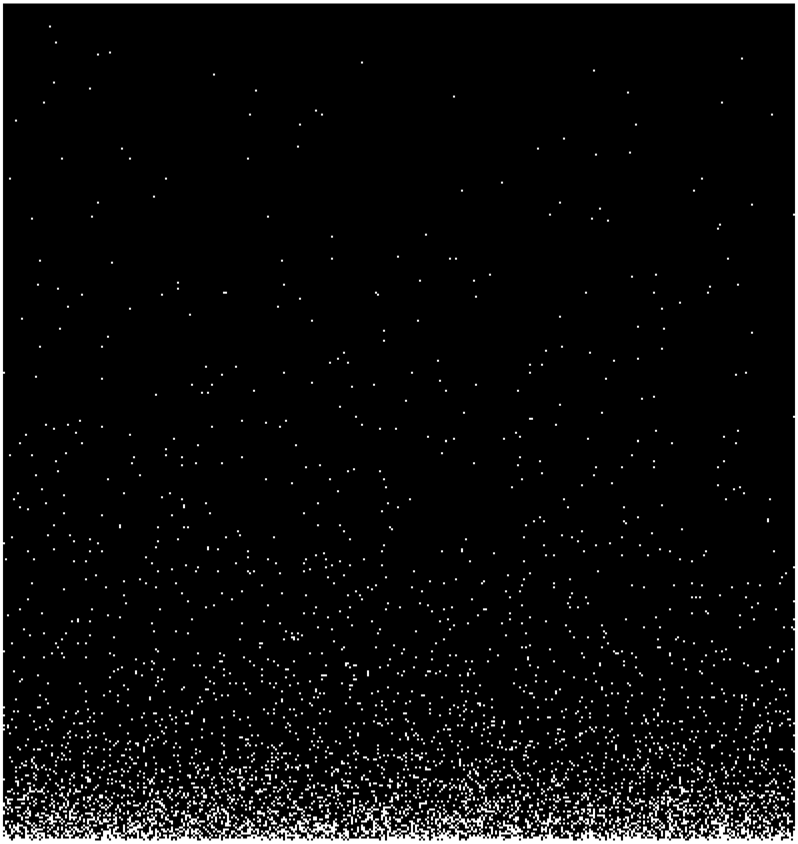
Running the executable produced 'atoms_state.txt'. Now, recreate the structure array from the saved matrix

```
load atoms_state.txt -ascii
clear atoms
for i = 1:size(atoms_state,1)
    atoms(1,i).x  = atoms_state(i,1);
    atoms(1,i).y  = atoms_state(i,2);
    atoms(1,i).vx = atoms_state(i,3);
    atoms(1,i).vy = atoms_state(i,4);
end
```

Render the State

Call 'run_atoms_mex' with zero iterations to render only

```
run_atoms_mex(atoms, 10000, 0);
```



Clean Up

Remove files and return to original folder

Run Command: Cleanup

```
if ispc
    delete run_atoms.exe
else
    delete run_atoms
end
delete atoms_state.txt
cleanup
```

How MATLAB Coder Partitions Generated Code

In this section...

Partitioning Generated Files

By default, during code generation, MATLAB Coder partitions the code to match your MATLAB file structure. This one-to-one mapping lets you easily correlate your files generated in C/C++ with the compiled MATLAB code. MATLAB Coder cannot produce the same one-to-one correspondence for MATLAB functions that are inlined in generated code (see "File Partitioning and Inlining" on page 20-134).

Alternatively, you can select to generate all C/C++ functions into a single file. For more information, see "How to Select the File Partitioning Method" on page 20-126. This option facilitates integrating your code with existing embedded software.

How to Select the File Partitioning Method

Using the MATLAB Coder App

1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2 Click **More Settings**.
3 On the **Code Appearance** tab, set the **Generated file partitioning method** to `Generate one file for each MATLAB file` or `Generate all functions into a single file`.

At the Command Line

Use the `codegen` configuration object `FilePartitionMethod` option. For example, to compile the function `foo` that has no inputs and generate one C/C++ file for each MATLAB function:

1. Create a MEX configuration object and set the `FilePartitionMethod` option:

   ```
   mexcfg = coder.config('mex');
   mexcfg.FilePartitionMethod = 'MapMFileToCFile';
   ```

2. Using the `-config` option, pass the configuration object to `codegen`:

   ```
   codegen -config mexcfg -O disable:inline foo
   % Disable inlining to generate one C/C++ file for each MATLAB function
   ```

Partitioning Generated Files with One C/C++ File Per MATLAB File

By default, for MATLAB functions that are not inlined, MATLAB Coder generates one C/C++ file for each MATLAB file. In this case, MATLAB Coder partitions generated C/C++ code so that it corresponds to your MATLAB files.

How MATLAB Coder Partitions Entry-Point MATLAB Functions

For each entry-point (top-level) MATLAB function, MATLAB Coder generates one C/C++ source, header, and object file with the same name as the MATLAB file.

For example, suppose you define a simple function `foo` that calls the function `identity`. The source file `foo.m` contains the following code:

```
function y = foo(u,v) %#codegen
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

Here is the code for `identity.m` :

```
function y = identity(u) %#codegen
y = u;
```

In the MATLAB Coder app, to generate a C static library for `foo.m`:

1. Define the inputs `u` and `v`. For more information, see “Specify Properties of Entry-Point Function Inputs Using the App” on page 17-3.
2. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
3. Set the **Build type** to `Static Library`

4 Click **More Settings**.

5 On the **All Settings** tab, under **Function Inlining**, set the **Inline threshold** parameter to `0`

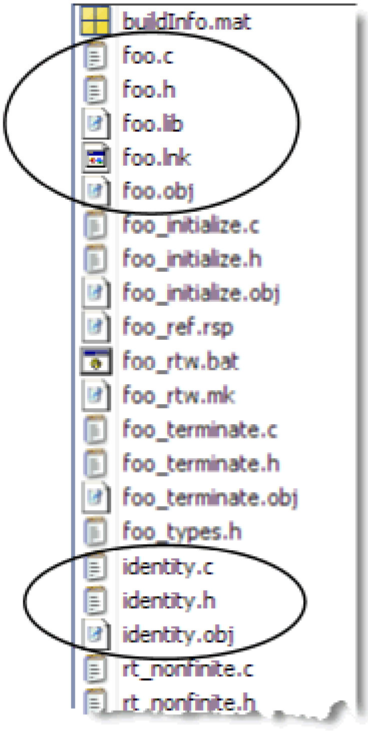
6 Click **Close**

7 To generate the library, click **Generate**.

To generate a C static library for `foo.m`, at the command line, enter:

```
codegen  -config:lib -O disable:inline foo -args {0, 0}
% Use the -args option to specify that u and v are both
% real, scalar doubles
```

MATLAB Coder generates source, header, and object files for `foo` and `identity` in your output folder.



buildInfo.mat
foo.c
foo.h
foo.lib
foo.lnk
foo.obj
foo_initialize.c
foo_initialize.h
foo_initialize.obj
foo_ref.rsp
foo_rtw.bat
foo_rtw.mk
foo_terminate.c
foo_terminate.h
foo_terminate.obj
foo_types.h
identity.c
identity.h
identity.obj
rt_nonfinite.c
rt_nonfinite.h

How MATLAB Coder Partitions Local Functions

For each local function, MATLAB Coder generates code in the same C/C++ file as the calling function. For example, suppose you define a function `foo` that calls a local function `identity`:

```
function y = foo(u,v) %#codegen
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);

function y = identity(u)
y = u;
```

To generate a C++ library, before generating code, select a C++ compiler and set C++ as your target language. For example, at the command line:

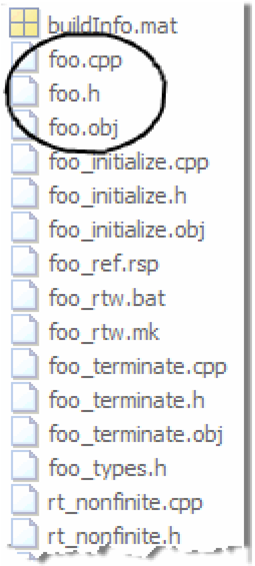
1. Select C++ as your target language:

   ```
   cfg = coder.config('lib')
   cfg.TargetLang='C++'
   ```

2. Generate the C++ library:

   ```
   codegen -config cfg   foo -args {0, 0}
   % Use the -args option to specify that u and v are both
   % real, scalar doubles
   ```

   In the primary function `foo`, MATLAB Coder inlines the code for the `identity` local function.



buildInfo.mat
foo.cpp
foo.h
foo.obj
foo_initialize.cpp
foo_initialize.h
foo_initialize.obj
foo_ref.rsp
foo_rtw.bat
foo_rtw.mk
foo_terminate.cpp
foo_terminate.h
foo_terminate.obj
foo_types.h
rt_nonfinite.cpp
rt_nonfinite.h

Note: If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Here is an excerpt of the generated code in foo.cpp:

```
...
/* Function Definitions */
double foo(double u, double v)
{
  return (double)(float)u + v;
}
...
```

How MATLAB Coder Partitions Overloaded Functions

An overloaded function is a function that has multiple implementations to accommodate different classes of input. For each implementation (that is not inlined), MATLAB Coder generates a separate C/C++ file with a unique numeric suffix.

For example, suppose you define a simple function multiply_defined:

```
%#codegen
function y = multiply_defined(u)

y = u+1;
```

You then add two more implementations of multiply_defined, one to handle inputs of type single (in an @single subfolder) and another for inputs of type double (in an @double subfolder).

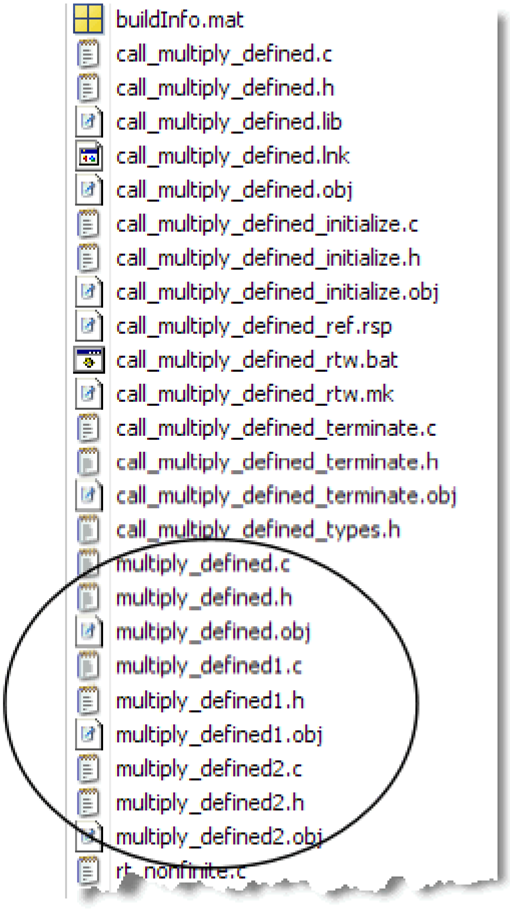
To call each implementation, define the function call_multiply_defined:

```
%#codegen
function [y1,y2,y3] = call_multiply_defined

y1 = multiply_defined(int32(2));
y2 = multiply_defined(2);
y3 = multiply_defined(single(2));
```

Next, generate C code for the overloaded function multiply_defined. For example, at the MATLAB command line, enter:

```
codegen -O disable:inline -config:lib call_multiply_defined
```

MATLAB Coder generates C source, header, and object files for each implementation of multiply_defined, as highlighted. Use numeric suffixes to create unique file names.



buildInfo.mat
call_multiply_defined.c
call_multiply_defined.h
call_multiply_defined.lib
call_multiply_defined.lnk
call_multiply_defined.obj
call_multiply_defined_initialize.c
call_multiply_defined_initialize.h
call_multiply_defined_initialize.obj
call_multiply_defined_ref.rsp
call_multiply_defined_rtw.bat
call_multiply_defined_rtw.mk
call_multiply_defined_terminate.c
call_multiply_defined_terminate.h
call_multiply_defined_terminate.obj
call_multiply_defined_types.h
multiply_defined.c
multiply_defined.h
multiply_defined.obj
multiply_defined1.c
multiply_defined1.h
multiply_defined1.obj
multiply_defined2.c
multiply_defined2.h
multiply_defined2.obj
rt_nonfinite.c

Generated Files and Locations

The types and locations of generated files depend on the target that you specify. For all targets, if errors or warnings occur during build or if you explicitly request a report, MATLAB Coder generates reports.

Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

Generated Files for MEX Targets

By default, MATLAB Coder generates the following files for MEX function (`mex`) targets.

Type of Files	Location
Platform-specific MEX files	Current folder
MEX, and C/C++ source, header, and object files	`codegen/mex/`*`function_name`*
HTML reports	`codegen/mex/`*`function_name`*`/html`

Generated Files for C/C++ Static Library Targets

By default, MATLAB Coder generates the following files for C/C++ static library targets.

Type of Files	Location
C/C++ source, library, header, and object files	`codegen/lib/`*`function_name`*
HTML reports	`codegen/lib/`*`function_name`*`/html`

Generated Files for C/C++ Dynamic Library Targets

By default, MATLAB Coder generates the following files for C/C++ dynamic library targets.

Type of Files	Location
C/C++ source, library, header, and object files	`codegen/dll/`*`function_name`*
HTML reports	`codegen/dll/`*`function_name`*`/html`

Generated Files for C/C++ Executable Targets

By default, MATLAB Coder generates the following files for C/C++ executable targets.

Type of Files	Location
C/C++ source, header, and object files	`codegen/exe/`*`function_name`*

Type of Files	Location
HTML reports	`codegen/exe/`*`function_name`*`/html`

Changing Names and Locations of Generated Files

Using the MATLAB Coder App

To change	Action
The output file name	1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼. 2 In the **Output file name** field, enter the file name.
The output file location	1 To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼. 2 Click **More Settings**. 3 On the **Paths** tab, set **Build folder** to `Specified folder`. 4 For the **Build folder name** field, either browse to the output file location or enter the full path. The output file location must not contain: • Spaces (Spaces can lead to code generation failures in certain operating system configurations). • Tabs • \, $, #, *, ? • Non-7-bit ASCII characters, such as Japanese characters.

At the Command Line

You can change the name and location of generated files by using the `codegen` options `-o` and `-d`.

File Partitioning and Inlining

How MATLAB Coder partitions generated C/C++ code depends on whether you choose to generate one C/C++ file for each MATLAB file and whether you inline your MATLAB functions.

If you	MATLAB Coder
Generate all C/C++ functions into a single file and disable inlining	Generates a single C/C++ file without inlining functions.
Generate all C/C++ functions into a single file and enable inlining	Generates a single C/C++ file. Inlines functions whose sizes fall within the inlining threshold.
Generate one C/C++ file for each MATLAB file and disable inlining	Partitions generated C/C++ code to match MATLAB file structure. See “Partitioning Generated Files with One C/C ++ File Per MATLAB File” on page 20-127.
Generate one C/C++ file for each MATLAB file and enable inlining	Places inlined functions in the same C/C++ file as the function into which they are inlined. Even when you enable inlining, MATLAB Coder inlines only those functions whose sizes fall within the inlining threshold. For MATLAB functions that are not inlined, MATLAB Coder partitions the generated C/C++ code, as described.

Tradeoffs Between File Partitioning and Inlining

Weighing file partitioning against inlining represents a trade-off between readability, efficiency, and ease of integrating your MATLAB code with existing embedded software.

If You Generate	Generated C/C++ Code	Advantages	Disadvantages
All C/C++ functions into a single file	Does not match MATLAB file structure	Easier to integrate with existing embedded software	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and enable inlining	Does not exactly match MATLAB file structure	Program executes faster	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and disable inlining	Matches MATLAB file structure	Easy to map C/C++ code to original MATLAB file	Program runs less efficiently

How Disabling Inlining Affects File Partitioning

Inlining is enabled by default. Therefore, to generate one C/C++ file for each top-level MATLAB function, you must:

- Select to generate one C/C++ file for each top-level MATLAB function. For more information, see “How to Select the File Partitioning Method” on page 20-126.
- Explicitly disable inlining, either globally or for individual MATLAB functions.

How to Disable Inlining Globally Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **All Settings** tab, under **Function Inlining** set the **Inline threshold** to `0`.

How to Disable Inlining Globally at the Command Line

To disable inlining of functions, use the `-O disable:inline` option with `codegen`. For example, to disable inlining and generate a MEX function for a function `foo` that has no inputs:

```
codegen  -O disable:inline foo
```

For more information, see the description of `codegen`.

How to Disable Inlining for Individual Functions

To disable inlining for an individual MATLAB function, add the directive `coder.inline('never');` on a separate line in the source MATLAB file, after the function signature.

```
function y = foo(u,v) %#codegen
coder.inline('never');
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

`codegen` does not inline entry-point functions.

The `coder.inline` directive applies only to the function in which it appears. In this example, inlining is disabled for function `foo`, but not for `identity`, a top-level function defined in a separate MATLAB file and called by `foo`. To disable inlining for `identity`, add this directive after its function signature in the source file `identity.m`. For more information, see `coder.inline`.

For a more efficient way to disable inlining for both functions, see “How to Disable Inlining Globally at the Command Line” on page 20-136.

Correlating C/C++ Code with Inlined Functions

To correlate the C/C++ code that you generate with the original inlined functions, add comments in the MATLAB code to identify the function. These comments will appear in the C/C++ code and help you map the generated code back to the original MATLAB functions.

Modifying the Inlining Threshold

To change inlining behavior, adjust the inlining threshold parameter.

Modifying the Inlining Threshold Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **All Settings** tab, under **Function Inlining**, set the value of the **Inline threshold** parameter.

Modifying the Inlining Threshold at the Command Line

Set the value of the `InlineThreshold` parameter of the configuration object. See coder.MexCodeConfig, coder.CodeConfig, coder.EmbeddedCodeConfig.

Requirements for Signed Integer Representation

You must compile the code that is generated by the MATLAB Coder software on a target that uses a two's complement representation for signed integer values. The generated code does not verify that the target uses a two's complement representation for signed integer values.

Customize the Post-Code-Generation Build Process

For certain applications, you might want to control aspects of the build process that occur after code generation but before compilation. For example, you might want to specify compiler or linker options. You can customize build processing that occurs after code generation using MATLAB Coder for MEX functions, C/C++ libraries and C/C++ executables.

You can customize your build using:

- The `coder.updateBuildInfo` function in your MATLAB code
- A post-code-generation command

In this section...

Customize Build Using coder.updateBuildInfo

To customize the post-code-generation build from your MATLAB code:

1. In your MATLAB code, call `coder.updateBuildInfo` to update the build information object. You specify a build information object method and the input arguments for the method. See `coder.updateBuildInfo` and "Build Information Methods" on page 20-141.
2. Generate code from your MATLAB code using the `codegen` command or from the project interface.

Customize Build Using Post-Code-Generation Command

To customize your build using the post-code-generation command:

1. “Write Post-Code-Generation Command” on page 20-175. Typically, you use this command to get the project name and build information or to add data to the build information object.
2. “Use Post-Code-Generation Command to Customize Build” on page 20-176.

Build Information Object

At the start of a build, the MATLAB Coder build process logs the following project, build option, and dependency information to a build information object, `RTW.BuildInfo`:

- Compiler options
- Preprocessor identifier definitions
- Linker options
- Source files and paths
- Include files and paths
- Precompiled external libraries

Use the “Build Information Methods” on page 20-141 to access this information in the build information object. “Write Post-Code-Generation Command” on page 20-175 explains how to use the functions to control a post-code-generation build.

When code generation is complete, MATLAB Coder creates a `buildInfo.mat` file in the build folder.

Build Information Methods

Use these methods to access or write data to the build information object. The syntax is:

```
buildInfo.method_name(input_arg1, ..., input_argn)
```

addCompileFlags

- Purpose: Add compiler options to build information.
- Syntax: `addCompileFlags(`*`buildinfo`*`, `*`options`*`, `*`groups`*`)`

  `groups` is optional.
- Arguments:

  `buildinfo`

Build information stored in `RTW.BuildInfo`.

`options`

A character array or cell array of character arrays that specifies the compiler options to be added to the build information. The function adds each option to the end of a compiler option vector. If you specify multiple options within a single character array, for example `'-Zi -Wall'`, the function adds the options to the vector as a single element. For example, if you add `'-Zi -Wall'` and then `'-O3'`, the vector consists of two elements, as shown below.

```
'-Zi -Wall'    '-O3'
```

`groups` (optional)

A character array or cell array of character arrays that groups specified compiler options. You can use groups to

- Document the use of specific compiler options
- Retrieve or apply collections of compiler options

You can apply

- A single group name to one or more compiler options
- Multiple group names to collections of compiler options (available for nonmakefile build environments only)

To...	**Specify *`groups`* as a...**
Apply one group name to compiler options	Character array.
Apply different group names to compiler options	Cell array of character arrays such that the number of group names matches the number of elements you specify for *`options`*.

- Description:

The `addCompileFlags` function adds specified compiler options to the project's build information. MATLAB Coder stores the compiler options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *`buildinfo`* and *`options`* arguments, you can use an optional *`groups`* argument to group your options.

addDefines

- Purpose: Add preprocessor macro definitions to build information.
- Syntax: `addDefines(`*`buildinfo`*`, `*`macrodefs`*`, `*`groups`*`)`

  `groups` is optional.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `macrodefs`

  A character array or cell array of character arrays that specifies the preprocessor macro definitions to be added to the object. The function adds each definition to the end of a compiler option vector. If you specify multiple definitions within a single character array, for example `'-DRT -DDEBUG'`, the function adds the options to the vector as a single element. For example, if you add `'-DPROTO -DDEBUG'` and then `'-DPRODUCTION'`, the vector consists of two elements, as shown below.

  ```
  '-DPROTO -DDEBUG'    '-DPRODUCTION'
  ```

  `groups` (optional)

  A character array or cell array of character arrays that groups specified definitions. You can use groups to

  - Document the use of specific macro definitions
  - Retrieve or apply groups of macro definitions

  You can apply

  - A single group name to one or more macro definitions
  - Multiple group names to collections of macro definitions (available for nonmakefile build environments only)

To...	Specify *`groups`* as a...
Apply one group name to macro definitions	Character array.

To...	Specify *groups* as a...
Apply different group names to macro definitions	Cell array of character arrays such that the number of group names matches the number elements you specify for *macrodefs*.

- Description:

  The addDefines function adds specified preprocessor macro definitions to the project's build information. The MATLAB Coder software stores the definitions in a vector. The function adds definitions to the end of the vector based on the order in which you specify them.

  In addition to the required *buildinfo* and *macrodefs* arguments, you can use an optional *groups* argument to group your options.

addIncludeFiles

- Purpose: Add include files to build information.
- Syntax: addIncludeFiles(*buildinfo*, *filenames*, *paths*, *groups*)

  paths and *groups* are optional.
- Arguments:

  buildinfo

  Build information stored in RTW.BuildInfo.

  filenames

  A character array or cell array of character arrays that specifies names of include files to be added to the build information.

  The filename can include wildcard characters, provided that the dot delimiter (.) is present. Examples are '*.*', '*.h', and '*.h*'.

  The function adds the filenames to the end of a vector in the order that you specify them.

  The function removes duplicate include file entries that

  - You specify as input
  - Already exist in the include file vector
  - Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path and corresponding filename.

`paths` (optional)

A character array or cell array of character arrays that specifies paths to the include files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

`groups` (optional)

A character array or cell array of character arrays that groups specified include files. You can use groups to

- Document the use of specific include files
- Retrieve or apply groups of include files

You can apply

- A single group name to an include file
- A single group name to multiple include files
- Multiple group names to collections of multiple include files

To...	Specify *`groups`* as a...
Apply one group name to include files	Character array.
Apply different group names to include files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for *`filenames`*.

- Description:

  The `addIncludeFiles` function adds specified include files to the project's build information. The MATLAB Coder software stores the include files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

  In addition to the required *`buildinfo`* and *`filenames`* arguments, you can specify optional *`paths`* and *`groups`* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to the include files it adds to the build information
Cell array of character arrays	Pairs each character array with a specified include file. Thus, the length of the cell array must match the length of the cell array you specify for *`filenames`*.

If you choose to specify *`groups`*, but omit *`paths`*, specify a null character vector (`''`) for *`paths`*.

addIncludePaths

- Purpose: Add include paths to build information.
- Syntax: `addIncludePaths(`*`buildinfo`*`, `*`paths`*`, `*`groups`*`)`

  `groups` is optional.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `paths`

  A character array or cell array of character arrays that specifies include file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

  The function removes duplicate include file entries that

  - You specify as input
  - Already exist in the include path vector
  - Have a path that matches the path of a matching filename

  A duplicate entry consists of an exact match of a path and corresponding filename.

  `groups` (optional)

  A character array or cell array of character arrays that groups specified include paths. You can use groups to

- Document the use of specific include paths
- Retrieve or apply groups of include paths

You can apply

- A single group name to an include path
- A single group name to multiple include paths
- Multiple group names to collections of multiple include paths

To	Specify `groups` as a
Apply one group name to include paths	Character array.
Apply different group names to include paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for *`paths`*.

- Description:

  The `addIncludePaths` function adds specified include paths to the project's build information. The MATLAB Coder software stores the include paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

  In addition to the required *`buildinfo`* and *`paths`* arguments, you can specify an optional *`groups`* argument. You can specify *`groups`* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a	The Function
Character array	Applies the character array to the include paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified include path. Thus, the length of the cell array must match the length of the cell array you specify for *`paths`*.

addLinkFlags

- Purpose: Add link options to build information.

- Syntax: `addLinkFlags(`*`buildinfo`*`, `*`options`*`, `*`groups`*`)`

  `groups` is optional.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `options`

  A character array or cell array of character arrays that specifies the linker options to be added to the build information. The function adds each option to the end of a linker option vector. If you specify multiple options within a single character array, for example `'-MD -Gy'`, the function adds the options to the vector as a single element. For example, if you add `'-MD -Gy'` and then `'-T'`, the vector consists of two elements, as shown below.

  ```
  '-MD -Gy'    '-T'
  ```

  `groups` (optional)

  A character array or cell array of character arrays that groups specified linker options. You can use groups to

  - Document the use of specific linker options
  - Retrieve or apply groups of linker options

  You can apply

  - A single group name to one or more linker options
  - Multiple group names to collections of linker options (available for nonmakefile build environments only)

To	Specify *`groups`* as a.
Apply one group name to linker options	Character array.
Apply different group names to linker options	Cell array of character arrays such that the number of group names matches the number of elements you specify for *`options`*.

- Description:

The `addLinkFlags` function adds specified linker options to the project's build information. The MATLAB Coder software stores the linker options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *`buildinfo`* and *`options`* arguments, you can use an optional *`groups`* argument to group your options.

addLinkObjects

- Purpose: Add link objects to build information.
- Syntax: `addLinkObjects(`*`buildinfo`*`, `*`linkobjs`*`, `*`paths`*`, `*`priority`*`, `*`precompiled`*`, `*`linkonly`*`, `*`groups`*`)`

  The arguments except *`buildinfo`* , *`linkobjs`*, and *`paths`* are optional. If you specify an optional argument, you must specify the optional arguments preceding it.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `linkobjs`

  A character array or cell array of character arrays that specifies the filenames of linkable objects to be added to the build information. The function adds the filenames that you specify in the function call to a vector that stores the object filenames in priority order. If you specify multiple objects that have the same priority (see *`priority`* below), the function adds them to the vector based on the order in which you specify the object filenames in the cell array.

  The function removes duplicate link objects that

  - You specify as input
  - Already exist in the linkable object filename vector
  - Have a path that matches the path of a matching linkable object filename

  A duplicate entry consists of an exact match of a path and corresponding linkable object filename.

  `paths`

A character array or cell array of character arrays that specifies paths to the linkable objects. If you specify a character array, the path applies to all linkable objects.

`priority` (optional)

A numeric value or vector of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority. The default priority is 1000.

`precompiled` (optional)

The logical value `true` or `false` or a vector of logical values that indicates whether each specified link object is precompiled.

Specify `true` if the link object has been prebuilt for faster compiling and linking and exists in a specified location.

If precompiled is `false` (the default), the MATLAB Coder build process creates the link object in the build folder.

This argument is ignored if *`linkonly`* equals `true`.

`linkonly` (optional)

The logical value `true` or `false` or a vector of logical values that indicates whether each specified link object is to be used only for linking.

Specify `true` if the MATLAB Coder build process should not build, nor generate rules in the makefile for building, the specified link object, but should include it when linking the final executable. For example, you can use this to incorporate link objects for which source files are not available. If *`linkonly`* is true, the value of *`precompiled`* is ignored.

If *`linkonly`* is `false` (the default), rules for building the link objects are added to the makefile. In this case, the value of *`precompiled`* determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (`true`) or `START_EXPAND_LIBRARIES` (`false`). The software performs the expansion of the `START_PRECOMP_LIBRARIES` or `START_EXPAND_LIBRARIES` macro only if your code generation target uses the template makefile approach for building code.

`groups` (optional)

A character array or cell array of character arrays that groups specified link objects. You can use groups to

- Document the use of specific link objects
- Retrieve or apply groups of link objects

You can apply

- A single group name to a linkable object
- A single group name to multiple linkable objects
- Multiple group name to collections of multiple linkable objects

To.	Specify ***groups*** as a
Apply one group name to link objects	Character array.
Apply different group names to link objects	Cell array of character arrays such that the number of group names matches the number elements you specify for *linkobjs*.

The default value of *groups* is `{''}`.

- Description:

  The `addLinkObjects` function adds specified link objects to the project's build information. The MATLAB Coder software stores the link objects in a vector in relative priority order. If multiple objects have the same priority or you do not specify priorities, the function adds the objects to the vector based on the order in which you specify them.

  In addition to the required *buildinfo*, *linkobjs*, and *paths* arguments, you can specify the optional arguments *priority*, *precompiled*, *linkonly*, and *groups*. You can specify *paths* and *groups* as a character array or a cell array of character arrays.

If You Specify ***paths*** or ***groups*** as a...	The Function...
Character array	Applies the character array to the objects it adds to the build information.

If You Specify *paths* or *groups* as a...	The Function...
Cell array of character arrays	Pairs each character array with a specified object. Thus, the length of the cell array must match the length of the cell array you specify for *linkobjs*.

Similarly, you can specify *priority*, *precompiled*, and *linkonly* as a value or vector of values.

If You Specify *priority*, *precompiled*, or *linkonly* as a	The Function
Value	Applies the value to the objects it adds to the build information.
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for *linkobjs*.

If you choose to specify an optional argument, you must specify the optional arguments preceding it. For example, to specify that objects are precompiled using the *precompiled* argument, you must specify the *priority* argument that precedes *precompiled*. You could pass the default priority value 1000, as shown below.

```
addLinkObjects(myBuildInfo, 'test1', '/proj/lib/lib1', 1000, true);
```

addNonBuildFiles

- Purpose: Add nonbuild-related files to build information.
- Syntax: `addNonBuildFiles(`*`buildinfo`*`, `*`filenames`*`, `*`paths`*`, `*`groups`*`)`

  paths and *groups* are optional.
- Arguments:

  buildinfo

  Build information stored in `RTW.BuildInfo`.

  filenames

  A character array or cell array of character arrays that specifies names of nonbuild-related files to be added to the build information.

The filename can include wildcard characters, provided that the dot delimiter (`.`) is present. Examples are `'*.*'`, `'*.DLL'`, and `'*.D*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate nonbuild file entries that

- Already exist in the nonbuild file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path and corresponding filename.

`paths` (optional)

A character array or cell array of character arrays that specifies paths to the nonbuild files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

`groups` (optional)

A character array or cell array of character arrays that groups specified nonbuild files. You can use groups to

- Document the use of specific nonbuild files
- Retrieve or apply groups of nonbuild files

You can apply

- A single group name to a nonbuild file
- A single group name to multiple nonbuild files
- Multiple group names to collections of multiple nonbuild files

To	Specify *`groups`* as a.
Apply one group name to nonbuild files	Character array.
Apply different group names to nonbuild files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for *`filenames`*.

- Description:

  The `addNonBuildFiles` function adds specified nonbuild-related files, such as DLL files required for a final executable, or a README file, to the project's build information. The MATLAB Coder software stores the nonbuild files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

  In addition to the required *`buildinfo`* and *`filenames`* arguments, you can specify optional *`paths`* and *`groups`* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a	The Function
Character array	Applies the character array to the nonbuild files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified nonbuild file. Thus, the length of the cell array must match the length of the cell array you specify for *`filenames`*.

  If you choose to specify *`groups`*, but omit *`paths`*, specify a null character vector (`''`) for *`paths`*.

addSourceFiles

- Purpose: Add source files to build information.
- Syntax: `addSourceFiles(`*`buildinfo`*`, `*`filenames`*`, `*`paths`*`, `*`groups`*`)`

  `paths` and *`groups`* are optional.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `filenames`

  A character array or cell array of character arrays that specifies names of the source files to be added to the build information.

  The filename can include wildcard characters, provided that the dot delimiter (`.`) is present. Examples are `'*.*'`, `'*.c'`, and `'*.c*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path and corresponding filename.

`paths` (optional)

A character array or cell array of character arrays that specifies paths to the source files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

`groups` (optional)

A character array or cell array of character arrays that groups specified source files. You can use groups to

- Document the use of specific source files
- Retrieve or apply groups of source files

You can apply

- A single group name to a source file
- A single group name to multiple source files
- Multiple group names to collections of multiple source files

To	Specify *`group`* as a
Apply one group name to source files	Character array.
Apply different group names to source files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for *`filenames`*.

- Description:

The addSourceFiles function adds specified source files to the project's build information. The MATLAB Coder software stores the source files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a.	The Function
Character array	Applies the character array to the source files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source file. Thus, the length of the cell array must match the length of the cell array you specify for *filenames*.

If you choose to specify *groups*, but omit *paths*, specify a null character vector ('') for *paths*.

addSourcePaths

- Purpose: Add source paths to build information.
- Syntax: addSourcePaths(*buildinfo*, *paths*, *groups*)

  groups is optional.
- Arguments:

  buildinfo

  Build information stored in RTW.BuildInfo.

  paths

  A character array or cell array of character arrays that specifies source file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

  The function removes duplicate source file entries that

  - You specify as input

- Already exist in the source path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path and corresponding filename.

Note: The MATLAB Coder software does not check whether a specified path is valid.

`groups` (optional)

A character array or cell array of character arrays that groups specified source paths. You can use groups to

- Document the use of specific source paths
- Retrieve or apply groups of source paths

You can apply

- A single group name to a source path
- A single group name to multiple source paths
- Multiple group names to collections of multiple source paths

To	Specify *groups* as a
Apply one group name to source paths	Character array.
Apply different group names to source paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for *paths*.

- Description:

  The `addSourcePaths` function adds specified source paths to the project's build information. The MATLAB Coder software stores the source paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

  In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument . You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a	The Function
Character array	Applies the character array to the source paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source path. Thus, the length of the character array or cell array must match the length of the cell array you specify for *paths*.

Note: The MATLAB Coder software does not check whether a specified path is valid.

addTMFTokens

- Purpose: Add template makefile (TMF) tokens that provide build-time information for makefile generation.

- Syntax: `addTMFTokens(`*`buildinfo`*`, `*`tokennames`*`, `*`tokenvalues`*`, `*`groups`*`)`

  `groups` is optional.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `tokennames`

  A character array or cell array of character arrays that specifies names of TMF tokens (for example, `'|>CUSTOM_OUTNAME<|'`) to be added to the build information. The function adds the token names to the end of a vector in the order that you specify them.

  If you specify a token name that already exists in the vector, the first instance takes precedence and its value used for replacement.

  `tokenvalues`

  A character array or cell array of character arrays that specifies TMF token values corresponding to the previously-specified TMF token names. The function adds the token values to the end of a vector in the order that you specify them.

  `groups` (optional)

  A character array or cell array of character arrays that groups specified TMF tokens. You can use groups to

  - Document the use of specific TMF tokens
  - Retrieve or apply groups of TMF tokens

  You can apply

  - A single group name to a TMF token
  - A single group name to multiple TMF tokens
  - Multiple group names to collections of multiple TMF tokens

To	Specify *`groups`* as a
Apply one group name to TMF tokens	Character array.
Apply different group names to TMF tokens	Cell array of character arrays such that the number of group names that you specify

To	Specify *groups* as a
	matches the number of elements you specify for *tokennames*.

- Description:

  Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled appropriately in the template makefile (TMF) for the target selected for your project. For example, if your post code generation command calls `addTMFTokens` to add a TMF token named `|>CUSTOM_OUTNAME<|` that specifies an output file name for the build, the TMF must act on the value of `|>CUSTOM_OUTNAME<|` to achieve the desired result.

  The `addTMFTokens` function adds specified TMF token names and values to the project's build information. The MATLAB Coder software stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

  In addition to the required *buildinfo*, *tokennames*, and *tokenvalues* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a	The Function
Character array	Applies the character array to the TMF tokens it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified TMF token. Thus, the length of the cell array must match the length of the cell array you specify for *tokennames*.

findIncludeFiles

- Purpose: Find and add include (header) files to build information.
- Syntax: `findIncludeFiles(`*`buildinfo`*`, `*`extPatterns`*`)`

  extPatterns is optional.
- Arguments:

  buildinfo

Build information stored in `RTW.BuildInfo`.

`extPatterns` (optional)

A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern

- Must start with `*.`
- Can include a combination of alphanumeric and underscore (_) characters

The default pattern is `*.h`.

Examples of valid patterns include

```
*.h
*.hpp
*.x*
```

- Description:

  The `findIncludeFiles` function

  - Searches for include files, based on specified file name extension patterns, in the source and include paths recorded in a project's build information object
  - Adds the files found, along with their full paths, to the build information object
  - Deletes duplicate entries

getCompileFlags

- Purpose: Get compiler options from build information.
- Syntax: *`options`* `= getCompileFlags(`*`buildinfo`*`,` *`includeGroups`*`,` *`excludeGroups`*`)`

  `includeGroups` and *`excludeGroups`* are optional.
- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `includeGroups` (optional)

  A character array or cell array of character arrays that specifies groups of compiler flags you want the function to return.

  `excludeGroups` (optional)

A character array or cell array of character arrays that specifies groups of compiler flags you do not want the function to return.

- Output arguments:

  Compiler options stored in the project's build information.
- Description:

  The `getCompileFlags` function returns compiler options stored in the project's build information. Using optional *`includeGroups`* and *`excludeGroups`* arguments, you can selectively include or exclude groups of options the function returns.

  If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

getDefines

- Purpose: Get preprocessor macro definitions from build information.
- Syntax: `[`*`macrodefs`*`, `*`identifiers`*`, `*`values`*`] = getDefines(`*`buildinfo`*`, `*`includeGroups`*`, `*`excludeGroups`*`)`

  `includeGroups` and *`excludeGroups`* are optional.
- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `includeGroups` (optional)

  A character array or cell array of character arrays that specifies groups of macro definitions you want the function to return.

  `excludeGroups` (optional)

  A character array or cell array of character arrays that specifies groups of macro definitions you do not want the function to return.
- Output arguments:

  Preprocessor macro definitions stored in the project's build information. The function returns the macro definitions in three vectors.

Vector	Description
`macrodefs`	Complete macro definitions with `-D` prefix

Vector	Description
identifiers	Names of the macros
values	Values assigned to the macros (anything specified to the right of the first equals sign) ; the default is an empty character vector ('')

- Description:

  The `getDefines` function returns preprocessor macro definitions stored in the project's build information. When the function returns a definition, it automatically

  - Prepends a `-D` to the definition if the `-D` was not specified when the definition was added to the build information
  - Changes a lowercase `-d` to `-D`

  Using optional *`includeGroups`* and *`excludeGroups`* arguments, you can selectively include or exclude groups of definitions the function is to return.

  If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

getFullFileList

- Purpose: Get All files from project's build information.
- Syntax: `[`*`fPathNames`*`, `*`names`*`] = getFullFileList(`*`buildinfo`*`, `*`fcase`*`)`

  `fcase` is optional.
- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `fcase` (optional)

  The character vector `'source'`, `'include'`, or `'nonbuild'`. If the argument is omitted, the function returns all the files from the build information object.

If You Specify	The Function
`'source'`	Returns source files from the build information object.

If You Specify	The Function
`'include'`	Returns include files from the build information object.
`'nonbuild'`	Returns nonbuild files from the build information object.

- Output arguments:

  Fully-qualified file paths and file names for files stored in the project's build information.

Note: Usually it is unnecessary to resolve the path of every file in the project build information, because the makefile for the project build will resolve file locations based on source paths and rules. Therefore, `getFullFileList` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

- Description:

  The `getFullFileList` function returns the fully-qualified paths and names of all files, or files of a selected type (source, include, or nonbuild), stored in the project's build information.

getIncludeFiles

- Purpose: Get include files from build information.
- Syntax: `files = getIncludeFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`

  `includeGroups` and *`excludeGroups`* are optional.
- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `concatenatePaths`

  The logical value `true` or `false`.

If You Specify	The Function
`true`	Concatenates and returns each filename with its corresponding path.
`false`	Returns only filenames.

`replaceMatlabroot`

The logical value `true` or `false`.

If You Specify	The Function
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

`includeGroups` (optional)

A character array or cell array of character arrays that specifies groups of include files you want the function to return.

`excludeGroups` (optional)

A character array or cell array of character arrays that specifies groups of include files you do not want the function to return.

- Output arguments:

  Names of include files stored in the project's build information.
- Description:

  The `getIncludeFiles` function returns the names of include files stored in the project's build information. Use the *`concatenatePaths`* and *`replaceMatlabroot`* arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional *`includeGroups`* and *`excludeGroups`* arguments, you can selectively include or exclude groups of include files the function returns.

  If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

getIncludePaths

- Purpose: Get include paths from build information.

- Syntax: *`files`*`=getIncludePaths(`*`buildinfo`*`, `*`replaceMatlabroot`*`, `*`includeGroups`*`, `*`excludeGroups`*`)`

  `includeGroups` and *`excludeGroups`* are optional.
- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `replaceMatlabroot`

  The logical value `true` or `false`.

If You Specify	The Function
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

  `includeGroups` (optional)

  A character array or cell array of character arrays that specifies groups of include paths you want the function to return.

  `excludeGroups` (optional)

  A character array or cell array of character arrays that specifies groups of include paths you do not want the function to return.
- Output arguments:

  Paths of include files stored in the build information object.
- Description:

  The `getIncludePaths` function returns the names of include file paths stored in the project's build information. Use the *`replaceMatlabroot`* argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional *`includeGroups`* and *`excludeGroups`* arguments, you can selectively include or exclude groups of include file paths the function returns.

  If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

getLinkFlags

- Purpose: Get link options from build information.
- Syntax: *`options`*`=getLinkFlags(`*`buildinfo`*`, `*`includeGroups`*`, `*`excludeGroups`*`)`

  `includeGroups` and *`excludeGroups`* are optional.
- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `includeGroups` (optional)

  A character array or cell array that specifies groups of linker flags you want the function to return.

  `excludeGroups` (optional)

  A character array or cell array that specifies groups of linker flags you do not want the function to return. To exclude groups and not include specific groups, specify an empty cell array (`''`) for *`includeGroups`*.
- Output arguments:

  Linker options stored in the project's build information.
- Description:

  The `getLinkFlags` function returns linker options stored in the project's build information. Using optional *`includeGroups`* and *`excludeGroups`* arguments, you can selectively include or exclude groups of options the function returns.

  If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

getNonBuildFiles

- Purpose: Get nonbuild-related files from build information.
- Syntax: *`files`*`=getNonBuildFiles(`*`buildinfo`*`, `*`concatenatePaths`*`, `*`replaceMatlabroot`*`, `*`includeGroups`*`, `*`excludeGroups`*`)`

  `includeGroups` and *`excludeGroups`* are optional.
- Input arguments:

buildinfo

Build information stored in RTW.BuildInfo.

concatenatePaths

The logical value true or false.

If You Specify	The Function
true	Concatenates and returns each filename with its corresponding path.
false	Returns only filenames.

replaceMatlabroot

The logical value true or false.

If You Specify	The Function
true	Replaces the token $(MATLAB_ROOT) with the absolute path for your MATLAB installation folder.
false	Does not replace the token $(MATLAB_ROOT).

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you do not want the function to return.

- Output arguments:

  Names of nonbuild files stored in the project's build information.

- Description:

  The getNonBuildFiles function returns the names of nonbuild-related files, such as DLL files required for a final executable, or a README file, stored in the project's build information. Use the *concatenatePaths* and *replaceMatlabroot* arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of nonbuild files the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null character vector (`''`) for *includeGroups*.

getSourceFiles

- Purpose: Get source files from project's build information.
- Syntax: *srcfiles*=`getSourceFiles(`*buildinfo*, *concatenatePaths*, *replaceMatlabroot*, *includeGroups*, *excludeGroups*`)`

  includeGroups and *excludeGroups* are optional.
- Input arguments:

  buildinfo

  Build information stored in `RTW.BuildInfo`.

  concatenatePaths

  The logical value `true` or `false`.

If You Specify	The Function
`true`	Concatenates and returns each filename with its corresponding path.
`false`	Returns only filenames.

Note: Usually it is unnecessary to resolve the path of every file in the project build information, because the makefile for the project build will resolve file locations based on source paths and rules. Therefore, specifying true for `concatenatePaths` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

replaceMatlabroot

The logical value `true` or `false`.

If You Specify	The Function
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

`includeGroups` (optional)

A character array or cell array of character arrays that specifies groups of source files you want the function to return.

`excludeGroups` (optional)

A character array or cell array of character arrays that specifies groups of source files you do not want the function to return.

- Output arguments:

  Names of source files stored in the project's build information.

- Description:

  The `getSourceFiles` function returns the names of source files stored in the project's build information. Use the *`concatenatePaths`* and *`replaceMatlabroot`* arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional *`includeGroups`* and *`excludeGroups`* arguments, you can selectively include or exclude groups of source files the function returns.

  If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

getSourcePaths

- Purpose: Get source paths from build information.
- Syntax: *`files`*`=getSourcePaths(`*`buildinfo`*`, `*`replaceMatlabroot`*`, `*`includeGroups`*`, `*`excludeGroups`*`)`

  `includeGroups` and *`excludeGroups`* are optional.

- Input arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

replaceMatlabroot

The logical value `true` or `false`.

If You Specify	The Function
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of source paths you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of source paths you do not want the function to return.

- Output arguments:

  Paths of source files stored in the project's build information.
- Description:

  The `getSourcePaths` function returns the names of source file paths stored in the project build information. Use the *replaceMatlabroot* argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of source file paths that the function returns.

  If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null character vector (`''`) for *includeGroups*.

packNGo

- Purpose: Package generated code in zip file for relocation.
- Syntax: `packNGo(`*`buildinfo`*`, `*`propVals`*`...)`

  propVals is optional.
- Arguments:

  buildinfo

Build information loaded from the build folder.

propVals (optional)

A cell array of property-value pairs that specify packaging details.

To	Specify Property	With Value
Package generated code files in a zip file as a single, flat folder.	'packType'	'flat' (default)
Package generated code files hierarchically in a primary zip file. The value of the 'nestedZipFiles' property determines whether the primary zip file contains secondary zip files or folders.	'packType'	'hierarchical'
Create a primary zip file that contains three secondary zip files: • mlrFiles.zip — files in your *matlabroot* folder tree • sDirFiles.zip — files in and under your build folder • otherFiles.zip — required files not in the *matlabroot* or start folder trees Paths for files in the secondary zip files are relative to the root folder of the primary zip file.	'nestedZipFiles'	true (default)
Create a primary zip file that contains folders, for example, your build folder and *matlabroot*.	'nestedZipFiles'	false
Specify a file name for the primary zip file.	'fileName'	'*name*' Default:'untitled.zip' If you omit the .zip file extension, the function adds it.

To	Specify Property	With Value
Include only the minimal header files required to build the code in the zip file.	`'minimalHeaders'`	`true` (default)
Include header files found on the include path in the zip file.	`'minimalHeaders'`	`false`
Include the `html` folder for your code generation report.	`'includeReport'`	`true` (default is `false`)
Direct `packNGo` not to error out on parse errors.	`'ignoreParseError'`	`true` (default is `false`)
Direct `packNGo` not to error out if files are missing.	`'ignoreFileMissing'`	`true` (default is `false`)

- Description:

  The `packNGo` function packages the following code files in a compressed zip file so you can relocate, unpack, and rebuild them in another development environment.

  - Source files (for example, `.c` and `.cpp` files)
  - Header files (for example, `.h` and `.hpp` files)
  - Nonbuild-related files (for example, `.dll` files required for a final executable file and `.txt` informational files)
  - MAT-file that contains the build information object (`.mat` file)

  Use this function to relocate files so that they can be recompiled for a specific target environment, or rebuilt in a development environment in which MATLAB is not installed.

  By default, the `packNGo` function packages the files as a flat folder structure in a zip file, `foo.zip`. The zip file is located in the current working folder.

  You can customize the output by specifying property name and value pairs as previously described.

  After relocating the zip file, use a standard zip utility to unpack the compressed file.
- Limitations:

  The following limitations apply to use of the `packNGo` function:

  - The function operates on source files only, such as `*.c`, `*.cpp`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
  - The function does not package example main source and header files that you generate using the default configuration settings. To package the example main files, configure code generation to generate and compile the example main function, generate your code, and then package the build files.
  - Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if they are not used.
  - `packNGo` does not package the code generated for MEX targets.
- See Also:
  - "Package Code for Other Development Environments" on page 24-46

updateFilePathsAndExtensions

- Purpose: Update files in project build information with missing paths and file extensions.
- Syntax: `updateFilePathsAndExtensions(`*`buildinfo`*`, `*`extensions`*`)`

  `extensions` is optional.
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `extensions` (optional)

  A cell array of character arrays that specifies the extensions (file types) of files for which to search and include in the update processing. By default, the function searches for files with a `.c` extension. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For example, if you specify `{'.c' '.cpp'}`, and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.
- Description:

  Using paths that already exist in a project's build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information need to be updated with a path or file extension. This function can be particularly useful for

- Maintaining build information for a toolchain that requires the use of file extensions
- Updating multiple customized instances of build information for a given project

updateFileSeparator

- Purpose: Change file separator used in project's build information.
- Syntax: `updateFileSeparator(`*`buildinfo`*`, `*`separator`*`)`
- Arguments:

  `buildinfo`

  Build information stored in `RTW.BuildInfo`.

  `separator`

  A character array that specifies the file separator \ (Windows) or / (UNIX®) to be applied to file path specifications.
- Description:

  The `updateFileSeparator` function changes instances of the current file separator (/ or \) in a project's build information to the specified file separator.

  The default value for the file separator matches the value returned by the MATLAB command `filesep`. For makefile based builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile. If the `GenerateMakefile` parameter is set, the MATLAB Coder software overrides the default separator and updates the build information after evaluating the `PostCodeGenCommand` configuration parameter.

Write Post-Code-Generation Command

A post-code-generation command is a MATLAB file that typically calls functions that get data from or add data to the build information object. For example, you can access the project name in the variable `projectName` and the `RTW.BuildInfo` object in the variable `buildInfo`. You can write the command as a script or a function.

If You Write the Command as a	Then the
Script	Script can gain access to the project (top-level function) name and the build information directly.
Function	Function can receive the project name and the build information as arguments.

If your post-code-generation command calls user-defined functions, make sure that the functions are on the MATLAB path. If the build process cannot find a function that you use in your command, the process fails.

You can call combinations of build information functions to customize the post-code-generation build. See "Write and Use Post-Code-Generation Command at the Command Line" on page 20-177

Write Post-Code-Generation Command as a Script

Set `PostCodeGenCommand` to the script name.

At the command line, enter:

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

Write Post-Code-Generation Command as a Function

Set `PostCodeGenCommand` to the function signature. When you define the command as a function, you can specify an arbitrary number of input arguments. If you want to access the project name, include `projectName` as an argument. If you want to modify or access build information, add `buildInfo` as an argument.

At the command line, enter:

```
cfg = coder.config('lib');
cfg.PostCodeGenCommand = 'FunctionName(projectName, buildInfo)';
```

Use Post-Code-Generation Command to Customize Build

After you have written a post-code-generation command, you must include this command in the build processing. You can include the command from the project settings dialog box or the command line.

Use Post-Code-Generation Command in the MATLAB Coder App.

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Custom Code** tab, set the **Post-code-generation command** parameter.

   How you use the `PostCodeGenCommand` option depends on whether you write the command as a script or a function. See “Use Post-Code-Generation Command at the Command Line” on page 20-177 and “Use Post-Code-Generation Command in the MATLAB Coder App.” on page 20-177.

Use Post-Code-Generation Command at the Command Line

Set the `PostCodeGenCommand` option for the code generation configuration object (coder.MexCodeConfig, coder.CodeConfig or coder.EmbeddedCodeConfig).

How you use the `PostCodeGenCommand` option depends on whether you write the command as a script or a function. See “Use Post-Code-Generation Command at the Command Line” on page 20-177 and “Use Post-Code-Generation Command in the MATLAB Coder App.” on page 20-177.

Write and Use Post-Code-Generation Command at the Command Line

The following example shows how to write and use a post-code-generation command as a function. The `setbuildargs` function takes the build information object as a parameter, sets up link options, and adds them to the build information object.

1. Create a post-code-generation command as a function, `setbuildargs`, which takes the `buildInfo` object as a parameter:

   ```
   function setbuildargs(buildInfo)
   % The example being compiled requires pthread support.
   % The -lpthread flag requests that the pthread library be included
   % in the build
       linkFlags = {'-lpthread'};
       buildInfo.addLinkFlags(linkFlags);
   ```

2. Create a code generation configuration object. Set the `PostCodeGenCommand` option to `'setbuildargs(buildInfo)'` so that this command is included in the build processing:

```
cfg = coder.config('mex');
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';
```

3 Using the `-config` option, generate a MEX function passing the configuration object to `codegen`. For example, for the function `foo` that has no input parameters:

```
codegen -config cfg foo
```

Run-time Stack Overflow

If your C compiler reports a run-time stack overflow, set the value of the maximum stack usage parameter to be less than the available stack size. In a project, in the project settings dialog box **Memory** tab, set the **Stack usage max** parameter. For command-line configuration objects (coder.MexCodeConfig, coder.CodeConfig, coder.EmbeddedCodeConfig), set the `StackUsageMax` parameter.

Pass Structure Arguments by Reference or by Value in Generated Code

This example shows how to control whether structure arguments to generated entry-point functions are passed by reference or by value.

Passing by reference uses a pointer to access the structure arguments. If the function writes to an element of the input structure, it overwrites the input value. Passing by value makes a copy of the input or output structure argument. To reduce memory usage and execution time, use pass by reference.

If a structure argument is both an input and output, the generated entry-point function passes the argument by reference. Generated MEX functions pass structure arguments by reference. For MEX function output, you cannot specify that you want to pass structure arguments by value.

Specify Pass by Reference or by Value Using the MATLAB Coder App

To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow.

Set the **Build type** to one of the following:

- Source Code
- Static Library
- Dynamic Library
- Executable

Click **More Settings**.

On the **All Settings** tab, set the **Pass structures by reference to entry-point functions** option to:

- Yes, for pass by reference (default)
- No, for pass by value

Specify Pass by Reference or by Value Using the Command-Line Interface

Create a code configuration object for a static library, a dynamic library, or an executable program. For example, create a code configuration object for a static library.

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

Set the PassStructByReference property to:

- true, for pass by reference (default)
- false, for pass by value

For example:

```
cfg.PassStructByReference = true;
```

Pass Input Structure Argument by Reference

Write the MATLAB function my_struct_in that has an input structure argument.

```
function y = my_struct_in(s)
%#codegen

y = s.f;
```

Define a structure variable mystruct in the MATLAB® workspace.

```
mystruct = struct('f', 1:4);
```

Create a code generation configuration object for a C static library.

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

Specify that you want to pass structure arguments by reference.

```
cfg.PassStructByReference = true;
```

Generate code. Specify that the input argument has the type of the variable mystruct.

```
codegen -config cfg -args {mystruct} my_struct_in
```

View the generated C code.

```
type codegen/lib/my_struct_in/my_struct_in.c
```

```
/*
 * my_struct_in.c
 *
 * Code generation for function 'my_struct_in'
 *
 */

/* Include files */
#include "rt_nonfinite.h"
#include "my_struct_in.h"

/* Function Definitions */
void my_struct_in(const struct0_T *s, double y[4])
{
  int i0;
  for (i0 = 0; i0 < 4; i0++) {
    y[i0] = s->f[i0];
  }
}

/* End of code generation (my_struct_in.c) */
```

The generated function signature for `my_struct_in` is

```
void my_struct_in(const struct0_T *s, double y[4])
```

`my_struct_in` passes the input structure `s` by reference.

Pass Input Structure Argument by Value

Specify that you want to pass structure arguments by value.

```
cfg.PassStructByReference = false;
```

Generate code. Specify that the input argument has the type of the variable `mystruct`.

```
codegen -config cfg -args {mystruct} my_struct_in
```

View the generated C code.

```
type codegen/lib/my_struct_in/my_struct_in.c
```

```
/*
 * my_struct_in.c
 *
 * Code generation for function 'my_struct_in'
 *
 */

/* Include files */
#include "rt_nonfinite.h"
#include "my_struct_in.h"

/* Function Definitions */
void my_struct_in(const struct0_T s, double y[4])
{
  int i0;
  for (i0 = 0; i0 < 4; i0++) {
    y[i0] = s.f[i0];
  }
}

/* End of code generation (my_struct_in.c) */
```

The generated function signature for `my_struct_in` is

```
void my_struct_in(const struct0_T s, double y[4]
```

`my_struct_in` passes the input structure `s` by value.

Pass Output Structure Argument by Reference

Write the MATLAB function `my_struct_out` that has an output structure argument.

```
function s = my_struct_out(x)
%#codegen

s.f = x;
```

Define a variable `a` in the MATLAB® workspace.

```
a = 1:4;
```

Create a code generation configuration object for a C static library.

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

Specify that you want to pass structure arguments by reference.

```
cfg.PassStructByReference = true;
```

Generate code. Specify that the input argument has the type of the variable `a`.

```
codegen -config cfg -args {a} my_struct_out
```

View the generated C code.

```
type codegen/lib/my_struct_out/my_struct_out.c
```

```
/*
 * my_struct_out.c
 *
 * Code generation for function 'my_struct_out'
 *
 */

/* Include files */
#include "rt_nonfinite.h"
#include "my_struct_out.h"

/* Function Definitions */
void my_struct_out(const double x[4], struct0_T *s)
{
  int i0;
  for (i0 = 0; i0 < 4; i0++) {
    s->f[i0] = x[i0];
  }
}

/* End of code generation (my_struct_out.c) */
```

The generated function signature for `my_struct_out` is

```
void my_struct_out(const double x[4], struct0_T *s)
```

`my_struct_out` passes the output structure `s` by reference.

Pass Output Structure Argument by Value

Specify that you want to pass structure arguments by value.

```
cfg.PassStructByReference = false;
```

Generate code. Specify that the input argument has the type of the variable `a`.

```
codegen -config cfg -args {a} my_struct_out
```

View the generated C code.

```
type codegen/lib/my_struct_out/my_struct_out.c
```

```
/*
 * my_struct_out.c
 *
 * Code generation for function 'my_struct_out'
 *
 */

/* Include files */
#include "rt_nonfinite.h"
#include "my_struct_out.h"

/* Function Definitions */
struct0_T my_struct_out(const double x[4])
{
  struct0_T s;
  int i0;
  for (i0 = 0; i0 < 4; i0++) {
    s.f[i0] = x[i0];
  }

  return s;
}

/* End of code generation (my_struct_out.c) */
```

The generated function signature for `my_struct_out` is

```
struct0_T my_struct_out(const double x[4])
```

`my_struct_out` returns an output structure.

Pass Input and Output Structure Argument by Reference

When an argument is both an input and an output, the generated C function passes the argument by reference even when `PassStructByReference` is false.

Write the MATLAB function `my_struct_inout` that has a structure argument that is both an input argument and an output argument.

```
function [y,s] = my_struct_inout(x,s)
%#codegen

y = x + sum(s.f);
```

Define the variable `a` and structure variable `mystruct` in the MATLAB® workspace.

```
a = 1:4;
mystruct = struct('f',a);
```

Create a code generation configuration object for a C static library.

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

Specify that you want to pass structure arguments by value.

```
cfg.PassStructByReference = false;
```

Generate code. Specify that the first input has the type of `a` and the second input has the type of `mystruct`.

```
codegen -config cfg -args {a, mystruct} my_struct_inout
```

View the generated C code.

```
type codegen/lib/my_struct_inout/my_struct_inout.c
```

```
/*
 * my_struct_inout.c
 *
 * Code generation for function 'my_struct_inout'
 *
 */

/* Include files */
#include "rt_nonfinite.h"
#include "my_struct_inout.h"

/* Function Definitions */
void my_struct_inout(const double x[4], const struct0_T *s, double y[4])
{
  double b_y;
  int k;
  b_y = s->f[0];
  for (k = 0; k < 3; k++) {
    b_y += s->f[k + 1];
  }

  for (k = 0; k < 4; k++) {
    y[k] = x[k] + b_y;
  }
}

/* End of code generation (my_struct_inout.c) */
```

The generated function signature for `my_struct_inout` is

```
void my_struct_inout(const double x[4], const struct0_T *s, double y[4])
```

`my_struct_inout` passes the structure `s` by reference even though `PassStructByReference` is `false`.

More About

- "Structure Definition for Code Generation" on page 7-2

Generate Code for an LED Control Function That Uses Enumerated Types

This example shows how to generate code for a function that uses enumerated types. In this example, the enumerated types inherit from base type int32. The base type can be int8, uint8, int16, uint16, or int32.

Define the enumerated type sysMode. Store it in sysMode.m on the MATLAB path.

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

Define the enumerated type LEDcolor. Store it in LEDcolor.m on the MATLAB path.

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

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

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

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

Generate a MEX function for `displayState`. Specify that `displayState` takes one input argument that has an enumerated data type `sysMode`.

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

Test the MEX function.

```
displayState_mex(sysMode.OFF)
```

```
ans = 

    RED
```

Generate a static library for the function `displayState`. Specify that `displayState` takes one input argument that has an enumerated data type `sysMode`.

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

codegen generates a C static library with the default name, `displayState`. It generates supporting files in the default folder, `codegen/lib/displayState`.

View the header file `displayState_types.h`.

```
type codegen/lib/displayState/displayState_types.h
```

```
/*
 * File: displayState_types.h
 *
 * MATLAB Coder version            : 3.3
 * C/C++ source code generated on  : 24-Feb-2017 11:27:00
 */

#ifndef DISPLAYSTATE_TYPES_H
#define DISPLAYSTATE_TYPES_H

/* Include Files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef enum_LEDcolor
#define enum_LEDcolor
```

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

#endif                                 /*enum_LEDcolor*/

#ifndef typedef_LEDcolor
#define typedef_LEDcolor

typedef enum LEDcolor LEDcolor;

#endif                                 /*typedef_LEDcolor*/

#ifndef enum_sysMode
#define enum_sysMode

enum sysMode
{
  OFF,
  ON
};

#endif                                 /*enum_sysMode*/

#ifndef typedef_sysMode
#define typedef_sysMode

typedef enum sysMode sysMode;

#endif                                 /*typedef_sysMode*/
#endif

/*
 * File trailer for displayState_types.h
 *
 * [EOF]
 */
```

The enumerated type `LEDcolor` is represented as a C enumerated type because the base type in the class definition for `LEDcolor` is `int32`. When the base type is `int8`, `uint8`, `int16`, or `uint16`, the code generator produces a `typedef` for the enumerated type. It produces `#define` statements for the enumerated type values. For example:

```
typedef short LEDcolor;
#define GREEN ((LEDcolor)1)
#define RED ((LEDcolor)2)
```

More About

- “Code Generation for Enumerations” on page 9-2
- “Customize Enumerated Types in Generated Code” on page 9-6

Verify Generated C/C++ Code

Generate Traceable Code

In this section...

You can configure MATLAB Coder to generate code that includes the MATLAB source code as comments. Include this information in the generated code to:

- Correlate the generated code with your source code.
- Understand how the generated code implements your algorithm.
- Evaluate the quality of the generated code.

In these generated comments, a traceability tag immediately precedes each line of source code. This traceability tag provides details about the location of the source code. See "Format of Traceability Tags" on page 21-3.

If you have Embedded Coder, you can also generate C/C++ code that includes the MATLAB function help text in the function banner. See "Tracing Between Generated C Code and MATLAB Code" (Embedded Coder).

Include MATLAB Source Code as Comments by Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Code Appearance** tab, under **Comments**, select the **MATLAB source code as comments** check box.
4. Generate code.

Include MATLAB Source Code as Comments by Using the Command-Line Interface

In the code generation configuration object, set the `MATLABSourceComments` parameter to `true`. For example, this code generates a static library for `foo.m` and includes the source code as comments in the generated code:

```
cfg = coder.config('lib');
cfg.MATLABSourceComments = true;
codegen -config cfg foo
```

Format of Traceability Tags

In the generated code, traceability tags appear immediately before the MATLAB source code in the comment. The format of the tag is:
`<filename>:<line number>`.

For example, the comment indicates that the code `x = r * cos(theta);` appears at line 4 in the source file `straightline.m`.

```
/* 'straightline:4' x = r * cos(theta); */
```

Note: With an Embedded Coder license, the traceability tags in the code generation report are hyperlinks to the MATLAB source code. For more information, see "Tracing Between Generated C Code and MATLAB Code" (Embedded Coder).

Location of Comments in Generated Code

The generated comments containing the source code and traceability tag appear in the generated code as follows.

Straight-Line Source Code

In straight-line source code without `if`, `while`, `for` or `switch` statements, the comment containing the source code precedes the generated code that implements the source code statement. This comment appears after user comments that precede the generated code.

For example, in the following code, the user comment, `/* Convert polar to Cartesian */`, appears before the generated comment containing the first line of source code, together with its traceability tag,

`/* 'straightline:4' x = r * cos(theta); */`.

MATLAB Code

```
function [x, y] = straightline(r,theta)
%#codegen
% Convert polar to Cartesian
x = r * cos(theta);
y = r * sin(theta);
```

Commented C Code

```
void straightline(double r, double theta, double *x, double *y)
{
  /*  Convert polar to Cartesian */
  /* 'straightline:4' x = r * cos(theta); */
  *x = r * cos(theta);

  /* 'straightline:5' y = r * sin(theta); */
  *y = r * sin(theta);
}
```

If Statements

The comment for the `if` statement immediately precedes the code that implements the statement. This comment appears after user comments that precede the generated code. The comments for the `elseif` and `else` clauses appear immediately after the code that implements the clause, and before the code generated for statements in the clause.

MATLAB Code

```
function y = ifstmt(u,v)
%#codegen
if u > v
    y = v + 10;
elseif u == v
    y = u * 2;
else
    y = v - 10;
end
```

Commented C Code

```
double ifstmt(double u, double v)
{
  double y;
  /* 'ifstmt:3' if u > v */
  if (u > v) {
    /* 'ifstmt:4' y = v + 10; */
    y = v + 10.0;
  } else if (u == v) {
    /* 'ifstmt:5' elseif u == v */
    /* 'ifstmt:6' y = u * 2; */
    y = u * 2.0;
  } else {
    /* 'ifstmt:7' else */
    /* 'ifstmt:8' y = v - 10; */
    y = v - 10.0;
  }

  return y;
}
```

For Statements

The comment for the `for` statement header immediately precedes the generated code that implements the header. This comment appears after user comments that precede the generated code.

MATLAB Code

```
function y = forstmt(u)
%#codegen
y = 0;
for i = 1:u
    y = y + 1;
end
```

Commented C Code

```
double forstmt(double u)
{
  double y;
  int i;

  /* 'forstmt:3' y = 0; */
  y = 0.0;

  /* 'forstmt:4' for i = 1:u */
  for (i = 0; i < (int)u; i++) {
    /* 'forstmt:5' y = y + 1; */
    y++;
  }

  return y;
}
```

While Statements

The comment for the `while` statement header immediately precedes the generated code that implements the statement header. This comment appears after user comments that precede the generated code.

MATLAB Code

```
function y = subfcn(y)
coder.inline('never');
while y < 100
    y = y + 1;
end
```

Commented C Code

```
void subfcn(double *y)
{
  /* 'subfcn:2' coder.inline('never'); */
  /* 'subfcn:3' while y < 100 */
  while (*y < 100.0) {
    /* 'subfcn:4' y = y + 1; */
    (*y)++;
  }
}
```

Switch Statements

The comment for the `switch` statement header immediately precedes the generated code that implements the statement header. This comment appears after user comments that precede the generated code. The comments for the `case` and `otherwise` clauses appear immediately after the generated code that implements the clause, and before the code generated for statements in the clause.

MATLAB Code

```
function y = switchstmt(u)
%#codegen
```

```
y = 0;
switch u
    case 1
        y = y + 1;
    case 3
        y = y + 2;
    otherwise
        y = y - 1;
end
```

Commented C Code

```
double switchstmt(double u)
{
  double y;

  /* 'switchstmt:3' y = 0; */
  /* 'switchstmt:4' switch u */
  switch ((int)u) {
   case 1:
    /* 'switchstmt:5' case 1 */
    /* 'switchstmt:6' y = y + 1; */
    y = 1.0;
    break;

   case 3:
    /* 'switchstmt:7' case 3 */
    /* 'switchstmt:8' y = y + 2; */
    y = 2.0;
    break;

   default:
    /* 'switchstmt:9' otherwise */
    /* 'switchstmt:10' y = y - 1; */
    y = -1.0;
    break;
  }

  return y;
}
```

Traceability Limitations

For MATLAB Coder, there are traceability limitations.

- You cannot include MATLAB source code as comments for:
  - MathWorks toolbox functions
  - P-code
- The appearance or location of comments can vary depending on the following conditions:
  - Even if the implementation code is eliminated, for example, due to constant folding, comments can still appear in the generated code.
  - If a complete function or code block is eliminated, comments can be eliminated from the generated code.
  - For certain optimizations, the comments can be separated from the generated code.
  - Even if you do not choose to include source code comments in the generated code, the generated code includes legally required comments from the MATLAB source code.

More About

- “Tracing Between Generated C Code and MATLAB Code” (Embedded Coder)
- “Code Generation Reports” on page 21-9

Code Generation Reports

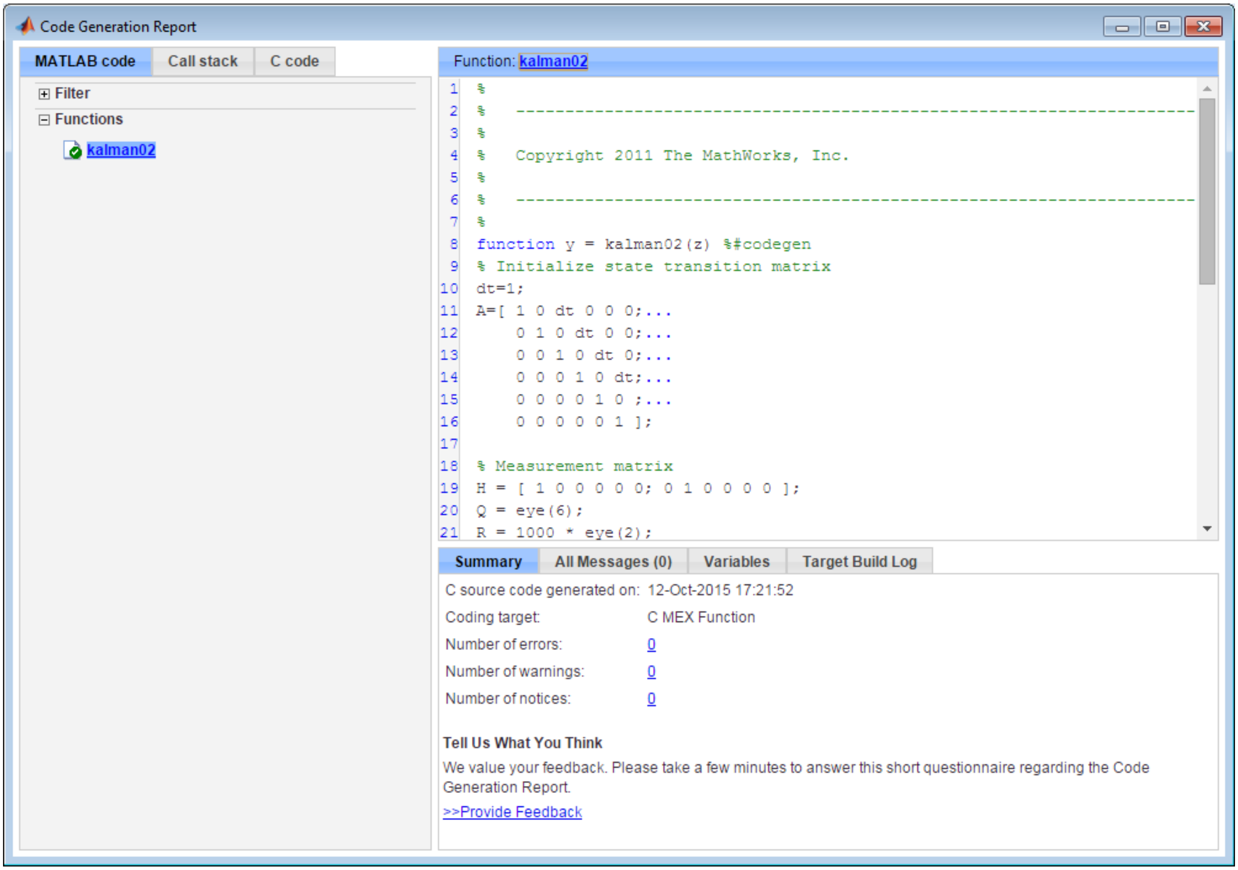
In this section...

Code Generation Report Overview

At code generation time, MATLAB Coder produces a report. The report helps you to verify that your MATLAB code is suitable for code generation and to debug code generation issues in your code.

The report provides links to your MATLAB code and C/C++ code files. It also provides compile-time type information for the variables and expressions in your MATLAB code. This information helps you to find sources of error messages and to understand type propagation rules.

Here is an example of a code generation report.



```
%
%   ----------------------------------------------------------------------
%
%   Copyright 2011 The MathWorks, Inc.
%
%   ----------------------------------------------------------------------
%
function y = kalman02(z) %#codegen
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];

% Measurement matrix
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
Q = eye(6);
R = 1000 * eye(2);
```

C source code generated on: 12-Oct-2015 17:21:52

Coding target: C MEX Function

Number of errors: 0

Number of warnings: 0

Number of notices: 0

Tell Us What You Think

We value your feedback. Please take a few minutes to answer this short questionnaire regarding the Code Generation Report.

>>Provide Feedback

The report provides:

- MATLAB code information, including a list of functions and classes.
- Call stack information, providing information on the nesting of function calls.
- Links to generated C/C++ code files.
- Summary of build results, including type of target and number of warnings or errors.
- List of error and warning messages.
- List of variables in your MATLAB code.
- Target build log of compilation and linking activities

Generating and Opening Reports

- Using the MATLAB Coder app:

  After code generation, on the **Generate Code** page, the MATLAB Coder app generates a code generation report and provides a link to the report. If the code generation report is enabled, or build errors occur, the **Target Build Log** tab also provides a link to the report. To view the report, click the report link.
- Using the `codegen` command:

  If the code generation report is enabled, or build errors occur, MATLAB Coder generates a code generation report and provides a link to the report. To open the report, click the report link. If you specify the `LaunchReport` option when you generate code, MATLAB Coder opens the report.

Names and Locations of Reports

MATLAB Coder produces code generation reports in the following locations. The top-level HTML file at each location is `index.html`.

- For MEX functions:

  `output_folder``/mex/`*`primary_function_name`*`/html`
- For C/C++ executables:

  `output_folder``/exe/`*`primary_function_name`*`/html`
- For C/C++ libraries:

  `output_folder``/lib/`*`primary_function_name`*`/html`

Note: The default output folder is `codegen`, but you can specify a different folder. For more information, see “Configure Build Settings” on page 20-26.

MATLAB Code in a Report

To view your MATLAB code, click the **MATLAB code** tab. The code generation report displays the code for the function or class highlighted in the list.

The **MATLAB code** tab provides:

- A list of the MATLAB functions and classes. Depending on the build results, the report displays icons next to each function or class name:

  - ❌ Errors in function or class.
  - ⚠ Warnings in function or class.
  - ✅ Completed build, no errors or warnings.

- A filter control. You can use **Filter functions and methods** to sort functions and methods by:

  - Size
  - Complexity
  - Class

- An optional highlight control to highlight potential data type issues in the generated C/C++ code. This option requires an Embedded Coder license. See “Highlight Potential Data Type Issues in a Report” (Embedded Coder).

Local Functions

In the function list on the **MATLAB code** tab, the code generation report annotates the local function with the name of the parent function.

For example, if the MATLAB function `fcn1` contains the local function `local_fcn1`, and `fcn2` contains the local function `local_fcn2`, the report displays:

```
fcn1 > local_fcn1
fcn2 > local_fcn2
```

Specializations

If your MATLAB function calls the same function with different types of inputs, the code generation report numbers these **specializations** in the function list on the **MATLAB code** tab.

For example, if the function `fcn` calls the function `subfcn` with different types of inputs:

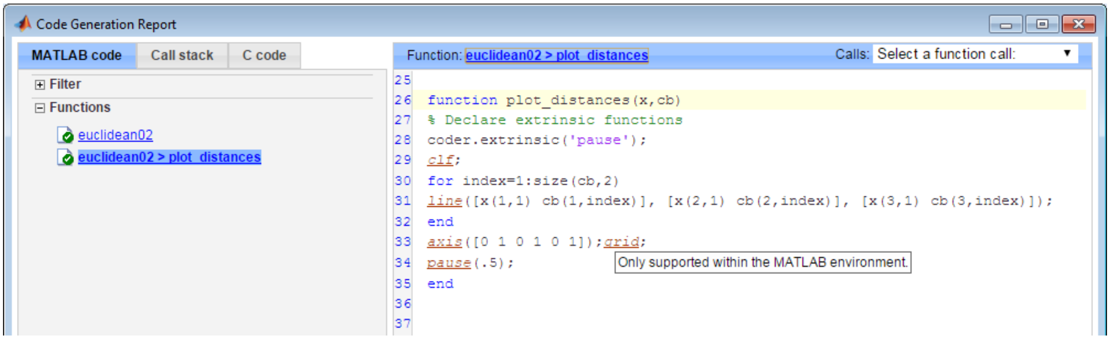
```
function y = fcn(u) %#codegen
% Specializations
y = y + subfcn(single(u));
y = y + subfcn(double(u));
```

The code generation report numbers the specializations in the function list:

```
fcn > subfcn > 1
fcn > subfcn > 2
```

Extrinsic Functions

The report highlights the extrinsic functions that are supported only within the MATLAB environment.


Code Generation Report
MATLAB code
Call stack
C code
Filter
Functions
euclidean02
euclidean02 > plot_distances
Function: euclidean02 > plot_distances
Calls: Select a function call:

```
function plot_distances(x,cb)
% Declare extrinsic functions
coder.extrinsic('pause');
clf;
for index=1:size(cb,2)
line([x(1,1) cb(1,index)], [x(2,1) cb(2,index)], [x(3,1) cb(3,index)]);
end
axis([0 1 0 1 0 1]);grid;
pause(.5);
end
```

Only supported within the MATLAB environment.

Call Stack Information in a Report

The code generation report provides call stack information:

- On the **Call stack** tab.
- In the list of **Calls** at the top right of the report.

  This list shows the calls from and to the function or method. If more than one function calls a function, this list provides details of each call-site. Otherwise, the list is disabled.

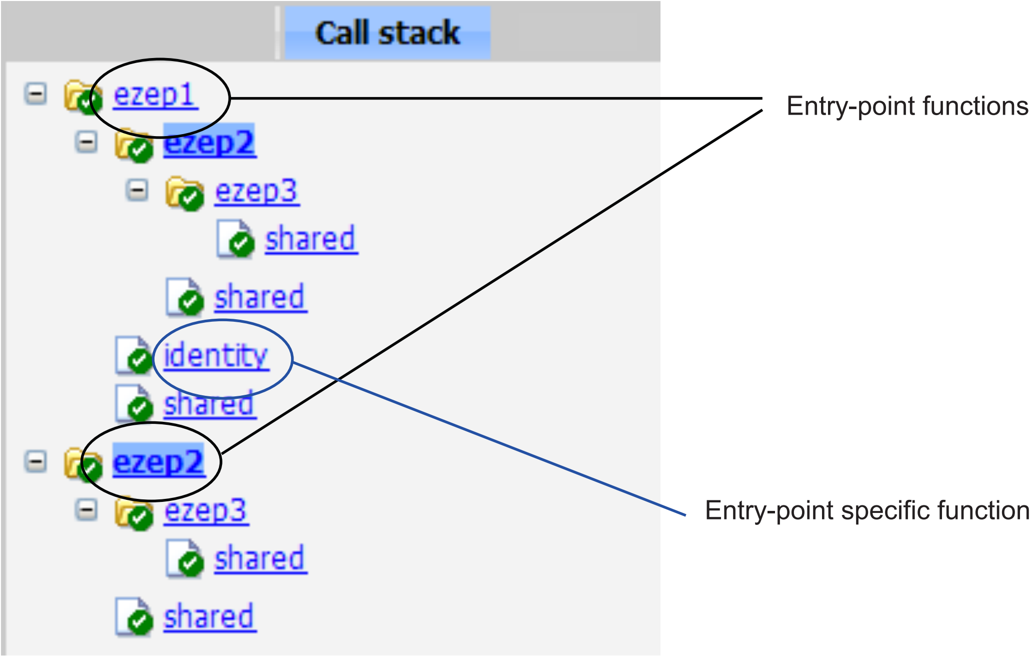
Call Stack Information on the Call stack Tab

To view call stack information, click the **Call stack** tab.

The call stack lists the functions and methods in the order that the top-level function calls them. It also lists the local functions that each function calls.

The call stack displays a separate tree for each entry point function. You can easily distinguish between shared and entry-point specific functions. If you click a shared function, the report highlights instances of this function. If you click an entry-point specific function, the report highlights only that instance.

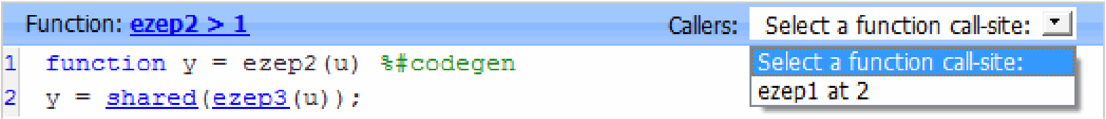
For example, in the following call stack, `ezep1` and `ezep2` are entry-point functions. `identity` is an entry-point specific function, called only by `ezep1`. Functions `ezep3` and `shared` are shared functions.



The Calls List

If more than one function calls a function, or if the functions calls other functions, the **Calls** list provides details of each call site. To navigate between call sites, select a call site from the **Calls** list. If the function is not called more than once, this list is disabled.

If you select the entry-point function `ezep2` in the call stack, the **Calls** list displays the other call site in `ezep1`.



```
function y = ezep2(u) %#codegen
y = shared(ezep3(u));
```

Generated C/C++ Code in a Report

To view a list of the generated C or C++ files, click the **C code** tab. The code generation report displays a list of the generated files in the **Target Source Files** pane. To view the code for a file, click the file.

If you generate a MEX function, a list of support files that the code generator uses appears in the **Interface Source Files** pane of the **C code** tab. By default, this list is collapsed.

When you generate an example main function, by default, the code generation report does not include the generated example main files. If you configure code generation to generate and compile an example main function, and then you generate code, example main files appear in the code generation report. A list of source and header files for the example main function appears in the **Example Source Files** pane of the **C code** tab.

Tracing Generated Code to MATLAB Source Code

You can configure `codegen` to generate C code that includes the MATLAB source code as comments. In these generated comments, `codegen` precedes each line of source code with a traceability tag that provides details about the location of the source code. See "Generate Traceable Code" on page 21-2.

For code generated using the Embedded Coder software, these traceability tags are hyperlinks. To go the relevant line in the source code in the MATLAB editor, click the tag.

Navigating to MATLAB Source Files for Generated C/C++ Code

To open the MATLAB source code file associated with a generated C/C++ function, click the link at the top of the code pane.

Navigating to Type Definitions

To see the definition for a generated C/C++ data type, click the data type in the generated C/C++ code pane.

Custom Code in a Report

The report displays custom code with color syntax highlighting. To learn what these colors mean and how to customize color settings, see "Check Syntax as You Type" (MATLAB).

Build Summary Information in a Report

To view a summary of the build results, including type of target and number of errors or warnings, click the **Summary** tab.

Errors and Warnings in a Report

MATLAB Coder reports errors and warnings. If errors occur during the build, MATLAB Coder does not generate code. The report lists the messages in the order that MATLAB Coder detects them. It is a best practice to address the first message in the list, because often subsequent errors and warnings are related to the first message. If the build produces warnings, but no errors, MATLAB Coder does generate code.

The code generation report:

- Lists errors and warnings on the **All Messages** tab. The report lists these messages in chronological order.
- Highlights errors and warnings on the **MATLAB code** pane.
- Records compilation and linking errors and warnings on the **Target Build Log** tab. If the code generator detects compilation warnings, you see a message on the **All Messages** tab. The code generator detects compilation warnings only for MEX output or if you use a supported compiler for other types of output. For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

Errors and Warnings in the All Messages Tab

If errors or warnings occur during the build, click the **All Messages** tab to view a complete list of these messages. The code generation report marks messages as either:

☒	Error
⚠	Warning

To locate the incorrect line of code for an error or warning, click the message in the list. The code generation report highlights errors in the list. It highlights MATLAB code in red and warnings in orange. To go to the error in the source file, click the line number next to the incorrect line of code in the MATLAB code pane.

Note: You can fix errors only in the source file.

Error and Warning Information in Your MATLAB Code

If errors or warnings occur during the build, the code generation report underlines them in your MATLAB code. The report underlines errors in red and warnings in orange. To learn more about a particular error or warning, place your cursor over the underlined text.

Compilation and Linking Errors and Warnings

The code generation report highlights compilation and linking errors and warnings in red on the **Target Build Log** tab. For errors, the code generation report opens to the **Target Build Log** tab so that you can view the build log. For warnings, the report opens to the **All Messages** tab. A message instructs you to view the warnings on the **Target Build Log** tab.

MATLAB Code Variables in a Report

The report provides compile-time type information for the variables and expressions in your MATLAB code, including name, type, size, complexity, and class. It also provides type information for fixed-point data types, including word length and fraction length. You can use this type information to find the sources of error messages and to understand type propagation rules.

You can view information about the variables in your MATLAB code:

- On the **Variables** tab, view the list.
- In your MATLAB code, place your cursor over the variable name.

  In the MATLAB code, an orange variable name indicates a compile-time constant argument to an entry-point or a specialized function. The information for a constant argument includes the value. The information about constant arguments helps you to understand generated function signatures. It also helps you to see when code generation created function specializations for different constant argument values.

Variables on the Variables Tab

To view a list of the variables in your MATLAB function, click the **Variables** tab. The report displays a complete list of variables in the order that they appear in the function that you selected on the **MATLAB code** tab. Clicking a variable in the list highlights instances of that variable, and scrolls the MATLAB code pane so that you can view the first instance.

As applicable, the report provides the following information about each variable:

- Order
- Name
- Type
- Size
- Complexity
- Class
- DataTypeMode (DT mode) — for fixed-point data types only. For more information, see “Data Type and Scaling Properties” (Fixed-Point Designer).
- Signed — sign information for built-in data types, signedness information for fixed-point data types.
- Word length (WL) — for fixed-point data types only.
- Fraction length (FL) — for fixed-point data types only.

Note: For more information on viewing fixed-point data types, see “Use Fixed-Point Code Generation Reports” (Fixed-Point Designer).

The report displays a column only if at least one variable in the code has information in that column. For example, if the code does not contain fixed-point data types, the report does not display the **DT mode**, **WL**, or **FL** columns.

Sorting Variables on the Variables Tab

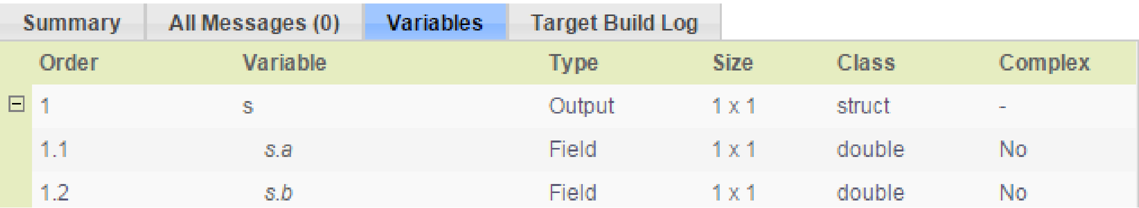
By default, the report lists the variables in the order that they appear in the selected function.

To sort the variables, click the column headings on the **Variables** tab. To sort the variables by multiple columns, hold down the **Shift** key when you click the column headings.

To restore the list to the original order, click the **Order** column heading.

Structures on the Variables Tab

To display structure field properties, expand the structure on the **Variables** tab.



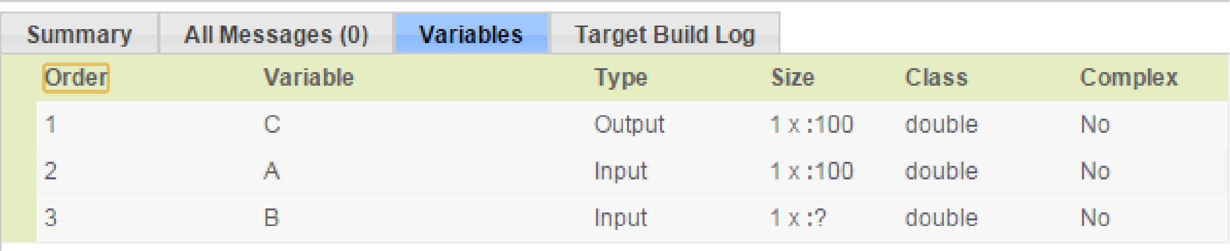
Order	Variable	Type	Size	Class	Complex
⊟ 1	s	Output	1 x 1	struct	-
1.1	*s.a*	Field	1 x 1	double	No
1.2	*s.b*	Field	1 x 1	double	No

If you sort the variables by type, size, complexity, or class, it is possible that a structure and its fields do not appear sequentially in the list. To restore the list to the original order, click the **Order** column heading.

Variable-Size Arrays in the Variables Tab

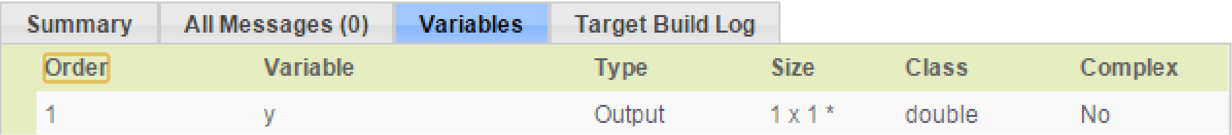
For variable-size arrays, the **Size** field includes information about the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon `:`. The size of an unbounded dimension is `:?`.

In the following report, variables A and B are variable-size. The second dimension of A has a maximum size of 100. The size of the second dimension of B is `:?`.



Order	Variable	Type	Size	Class	Complex
1	C	Output	1 x :100	double	No
2	A	Input	1 x :100	double	No
3	B	Input	1 x :?	double	No

If you declare a variable-size array, and then 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 sizes of its dimensions do not change during execution.



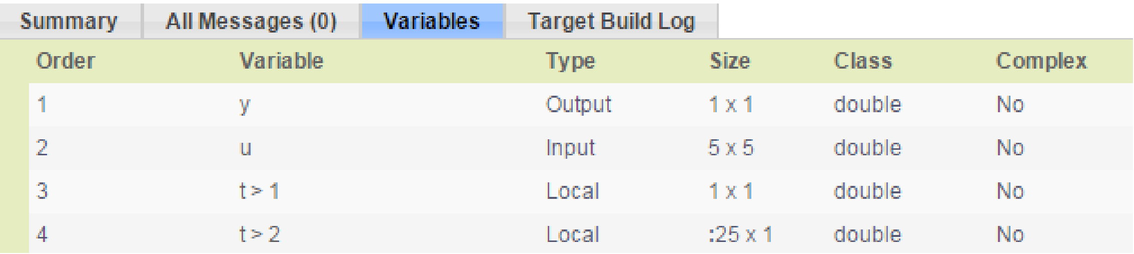
Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1 *	double	No

For information about how to use the size information for variable-size arrays, see "Code Generation for Variable-Size Arrays" on page 6-2.

Renamed Variables in the Variables Tab

If your MATLAB function reuses a variable with a different size, type, or complexity, the code generator attempts to create separate, uniquely named variables. For more information, see "Reuse the Same Variable with Different Properties" on page 4-10. The report numbers the renamed variables in the list on the **Variables** tab. When you place your cursor over a renamed variable, the report highlights only the instances of this variable that share data type, size, and complexity.

For example, suppose that your code uses the variable `t` to hold a scalar double, and reuses it outside the for-loop to hold a vector of doubles. In the list on the **Variables** tab, the report displays two variables, `t>1` and `t>2`,

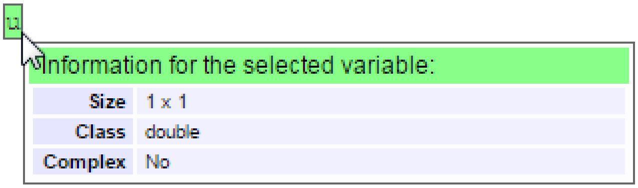


Order	Variable	Type	Size	Class	Complex
1	y	Output	1 x 1	double	No
2	u	Input	5 x 5	double	No
3	t > 1	Local	1 x 1	double	No
4	t > 2	Local	:25 x 1	double	No

Viewing Information About Variables and Expressions in Your MATLAB Function Code

To view information about a particular variable or expression in your MATLAB function code, on the MATLAB code pane, place your cursor over the variable name or expression. The report highlights variables and expressions in different colors:

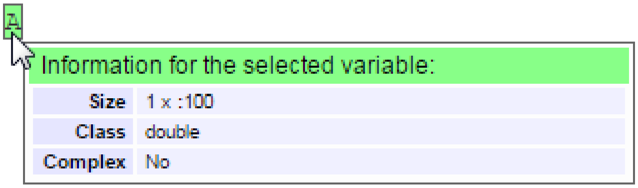
Green, when the variable has data type information at this location in the code



Information for the selected variable:

Size	1 x 1
Class	double
Complex	No

For variable-size arrays, the **Size** field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon **:**.



A

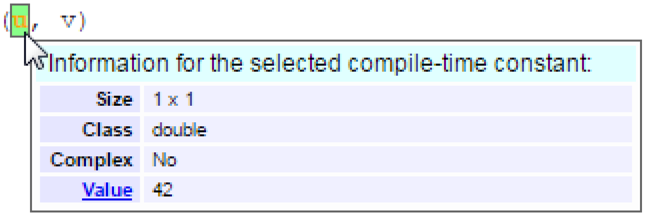
Information for the selected variable:

Size	1 x :100
Class	double
Complex	No

Green with orange text, when a constant argument has data type and value information

When the variable is a compile-time constant argument to an entry-point or a specialized function:

- The variable name is orange.
- The information for the variable includes the value.



(u, v)

Information for the selected compile-time constant:

Size	1 x 1
Class	double
Complex	No
Value	42

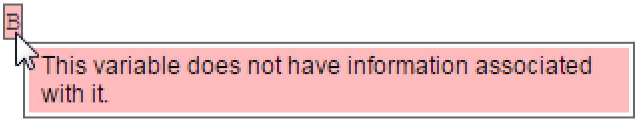
If you export the value as a variable to the base workspace, you can use the Workspace browser to view detailed information about the variable.

To export the value to the base workspace:

1. Click the **Value** link.
2. In the Export Constant Value dialog box, specify the **Variable name**.
3. Click **OK**.

   The variable and its value appear in the Workspace browser.

Pink, when the variable has no data type information

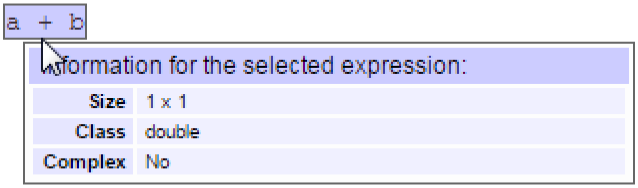


B

This variable does not have information associated with it.

Purple, information about expressions

You can also view information about expressions in your MATLAB code. On the MATLAB code pane, place your cursor over an expression. The report highlights expressions in purple and provides more detailed information.

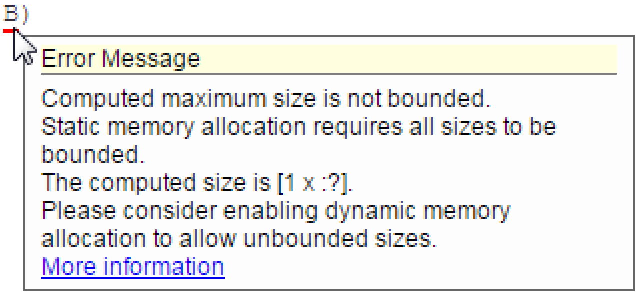


a + b

Information for the selected expression:

Size	1 x 1
Class	double
Complex	No

Red, when there is error information



B)

Error Message

Computed maximum size is not bounded.
Static memory allocation requires all sizes to be bounded.
The computed size is [1 x :?].
Please consider enabling dynamic memory allocation to allow unbounded sizes.
More information

Target Build Information in a Report

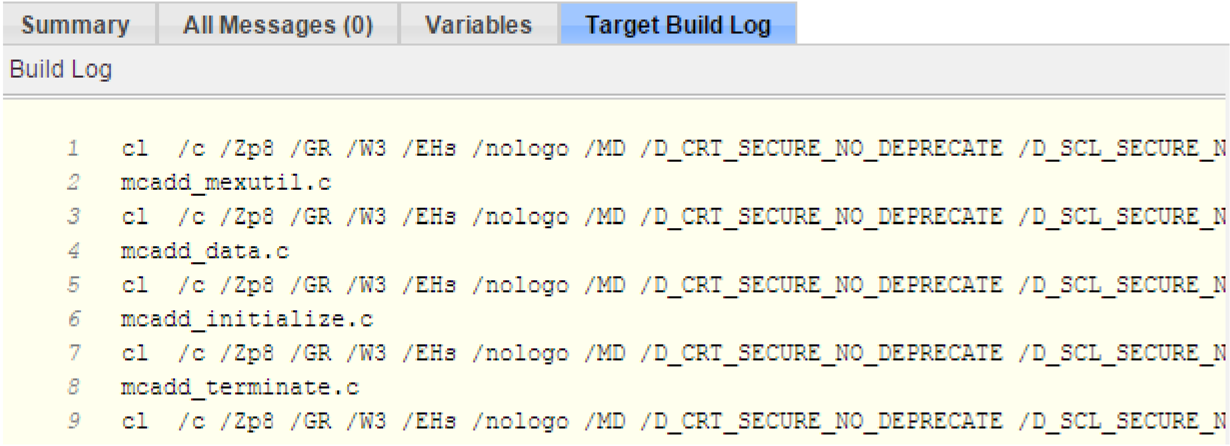
If the build completes, MATLAB Coder provides target build information on the **Target Build Log** tab, including:

- Build folder

  Clicking this link changes the MATLAB current folder to the build folder.
- Make wrapper

  The batch file name that MATLAB Coder used for this build.
- Build log

  If compilation or linking errors occur, the code generation report opens to the **Target Build Log** tab so that you can view the build log.



```
cl  /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_N
mcadd_mexutil.c
cl  /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_N
mcadd_data.c
cl  /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_N
mcadd_initialize.c
cl  /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_N
mcadd_terminate.c
cl  /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_N
```

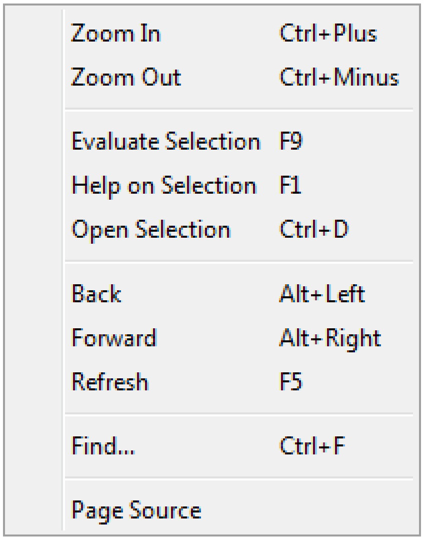
Keyboard Shortcuts for a Report

You can use keyboard shortcut settings to perform actions in the code generation report.

This table lists actions that you can associate with a keyboard shortcut. The keyboard shortcuts are defined in your MATLAB preferences. See “Define Keyboard Shortcuts” (MATLAB).

Action	Default Keyboard Shortcut for a Windows Platform
Zoom in	**Ctrl+Plus**
Zoom out	**Ctrl+Minus**
Evaluate selected MATLAB code	**F9**
Open help for selected MATLAB code	**F1**
Open selected MATLAB code	**Ctrl+D**
Step backward through files that you opened in the code pane	**Alt+Left**
Step forward through files that you opened in the code pane	**Alt+Right**
Refresh	**F5**
Search	**Ctrl+F**

Alternatively, you can select these actions from a context menu. To open the context menu, right-click anywhere in the report.



This table lists keyboard shortcuts that help you navigate between panes and tabs in the code generation report. To advance through data in the selected pane, use the **Tab** key. These keyboard shortcuts override the keyboard shortcut settings in your MATLAB preferences. See “Define Keyboard Shortcuts” (MATLAB).

To select	Use
MATLAB code tab	**Ctrl+M**
Call stack tab	**Ctrl+K**
C code Tab	**Ctrl+C**
Code Pane	**Ctrl+W**
Summary Tab	**Ctrl+S**
All Messages Tab	**Ctrl+A**
Variables Tab	**Ctrl+V**
Target Build Log Tab	**Ctrl+T**

Searching in a Report

Use the keyboard shortcut associated with `Find` in your MATLAB preferences. For example, on a Windows platform, the default keyboard shortcut for `Find` is **Ctrl+F**.

Report Limitations

The report displays information about the variables and expressions in your MATLAB code with the following limitations:

varargin and varargout

The report does not support `varargin` and `varargout` arrays.

Loop Unrolling

The report does not display full information for unrolled loops. It displays data types of one arbitrary iteration.

Dead Code

The report does not display information about dead code.

Structures

The report does not provide complete information about structures.

- The report does not provide information about all structure fields in the `struct()` constructor.
- If a structure has a nonscalar field, and an expression accesses an element of this field, the report does not provide information for the field.

Column Headings on the Variables Tab

If you scroll through the list of variables, the report does not display the column headings on the **Variables** tab.

Multiline Matrices

On the **MATLAB code** pane, the report does not support selection of multiline matrices. It supports only selection of individual lines at a time. For example, if you place your cursor over the following matrix, you cannot select the entire matrix.

```
out1 = [1 2 3;
       4 5 6];
```

The report does support selection of single line matrices.

```
out1 = [1 2 3; 4 5 6];
```

Enable Code Generation Reports

In this section...

Enable Code Generation Reports with the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Debugging** tab, select the **Always create a code generation report** check box.
4. If you want the app to open the report, select the **Automatically launch a report if one is generated** check box.

Enable Code Generation Reports at the Command Line

Use the `codegen` function `-report` option. To generate a standalone C/C++ static library and code generation report for a function `foo` that has no input parameters, at the MATLAB command line, enter:

```
codegen -config:lib -report foo
```

If you want the code generation or error report to automatically open, use the `-launchreport` option instead of the `-report` option.

Run-Time Error Detection and Reporting in Standalone C/C++ Code

You can generate standalone libraries and executables that detect and report run-time errors, such as out-of-bounds array indexing. If the generated code detects an error, it reports the error and terminates the program.

During development, before you generate C/C++ code, it is a best practice to test the generated code by running the MEX version of your algorithm. However, some errors occur only on the target hardware. To detect these errors, generate the standalone C/C ++ code with run-time error detection enabled. Run-time error detection can affect the performance of the generated code. If performance is a consideration for your application, do not generate production code with run-time error detection enabled.

By default, run-time error detection is disabled for standalone libraries and executables. To enable run-time error detection and reporting for standalone libraries and executables:

- At the command line, use the code configuration property `RuntimeChecks`.

  ```
  cfg = coder.config('lib'); % or 'dll' or 'exe'
  cfg.RuntimeChecks = true;
  codegen -config cfg myfunction
  ```
- In the MATLAB Coder app, in the project settings dialog box, on the **Debugging** pane, select the **Generate run-time error checks** check box.

Run-time error detection and reporting in standalone code has these requirements and limitations:

- The error reporting software uses `fprintf` to write error messages to `stderr`. It uses `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. The `abort` function abruptly terminates the program. If your system supports signals, you can catch the abort signal (`SIGABRT`) so that you can control the program termination.
- Error messages are in English only.
- Some error checks require double-precision support. Therefore, the hardware on which the generated code runs must support double-precision operations.
- If the program terminates, the error detection and reporting software does not display the run-time stack. To inspect the stack, attach a debugger. Additionally, the

error detection and reporting software does not release resources, such as allocated memory.

- If the program terminates, the error detection and reporting software does not release resources, such as allocated memory.
- In standalone code, the function `error` does not report an error and does not terminate execution.
- In standalone code, if called with more than 1 argument, the function `assert` does not report an error and does not terminate execution. If called with a single argument, for example, `assert(cond)`, if `cond` is not a constant `true` value, reports an error and terminates execution.

Related Examples

- "Generate Standalone Code That Detects and Reports Run-Time Errors" on page 21-30

More About

- "Why Test MEX Functions in MATLAB?" on page 19-2

Generate Standalone Code That Detects and Reports Run-Time Errors

This example shows how to generate C/C++ libraries or executables that detect and report run-time errors such as out-of-bounds array indexing. If the generated code detects an error, it reports a message and terminates the program. You can detect and fix errors that occur only on the target hardware.

Write the function `getelement` that indexes into one structure field using the value of the other structure field.

```
function y = getelement(S)
y = S.A(S.u);
end
```

Create a code configuration object for a standalone library or executable. For example, create a code configuration object for a static library. Enable the code generation report.

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

Enable generation of run-time error detection and reporting.

```
cfg.RuntimeChecks = true;
```

Define an example input that you can use to specify the properties of the input argument.

```
S.A = ones(2,2);
S.u = 0;
```

Generate code.

```
codegen -config cfg getelement -args {S}
```

To open the code generation report, click the **View report** link.

When the report opens, you see the generated C code. You can see the code that checks for an error and calls a function to report the error. For example, if the code detects an out-of-bounds array indexing error, it calls `rtDynamicBoundsError` to report the error and terminate the program.

```
/* Include Files */
#include "rt_nonfinite.h"
#include "getelement.h"
```

```
#include "getelement_rtwutil.h"
#include <stdio.h>
#include <stdlib.h>

/* Variable Definitions */
static rtBoundsCheckInfo emlrtBCI = { 1, 4, 2, 5, "S.A", "getelement",
  "C:\\coder\\runtime checks\\getelement.m", 0 };

static rtDoubleCheckInfo emlrtDCI = { 2, 5, "getelement",
  "C:\\coder\\runtime checks\\getelement.m", 1 };

/* Function Definitions */

/*
 * Arguments    : const struct0_T *S
 * Return Type  : double
 */
double getelement(const struct0_T *S)
{
  double d0;
  int i0;
  d0 = S->u;
  if (d0 != (int)floor(d0)) {
    rtIntegerError(d0, &emlrtDCI);
  }

  i0 = (int)d0;
  if (!((i0 >= 1) && (i0 <= 4))) {
    rtDynamicBoundsError(i0, 1, 4, &emlrtBCI);
  }

  return S->A[i0 - 1];
}
```

The error reporting software uses `fprintf` to write error messages to `stderr`. It uses `abort` to terminate the application. If `fprintf` and `abort` are not available, you must provide them. The `abort` function abruptly terminates the program. If your system supports signals, you can catch the abort signal (`SIGABRT`) so that you can control the program termination.

More About

- "Run-Time Error Detection and Reporting in Standalone C/C++ Code" on page 21-28

Testing Code Generated from MATLAB Code

MATLAB Coder helps you to test your generated code.

If you use the MATLAB Coder app to generate a MEX function, you can test the MEX function in the app. If you use `codegen` to generate a MEX function, test the MEX function by using `coder.runTest`. Alternatively, use the `codegen -test` option.

If you have Embedded Coder, you can verify the numerical behavior of generated C/C++ code by using software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution. You can also produce a profile of execution times.

More About

- “Verify MEX Functions in the MATLAB Coder App” on page 19-8
- “Verify MEX Functions at the Command Line” on page 19-9
- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” (Embedded Coder)
- “Execution Time Profiling for SIL and PIL” (Embedded Coder)
- “Unit Test Generated Code with MATLAB Coder” on page 21-33
- “Unit Test External C Code with MATLAB Coder” on page 21-41

Unit Test Generated Code with MATLAB Coder

This example shows how to test the output of generated code by using MATLAB® unit tests with MATLAB® Coder™.

To monitor for regressions in code functionality, you can write unit tests for your code. In MATLAB, you can create and run unit tests by using the MATLAB testing framework. To test MEX code and standalone code that you generate from MATLAB code, you can use the same unit tests that you use to test MATLAB code.

A MEX function includes instrumentation that helps you to detect issues before you generate production code. Running unit tests on a MEX function tests the instrumented code in MATLAB. Generated standalone code (static library or shared library) does not include the instrumentation and can include optimizations that are not present in the MEX code. To run unit tests on standalone code in a separate process outside of MATLAB, use software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution. To use SIL or PIL execution, you must have Embedded Coder®.

This example shows how to:

1. Create MATLAB unit tests that call your MATLAB function. This example uses class-based unit tests.
2. Generate a MEX function from your MATLAB function.
3. Run the unit tests on the MEX function.
4. Run the unit tests on standalone code by using SIL.

Examine the Files

To access the files that this example uses, click **Open Script**.

addOne.m

The example performs unit tests on the MEX function generated from the MATLAB function `addOne`. This function adds 1 to its input argument.

```
function y = addOne(x)
% Copyright 2014 - 2016 The MathWorks, Inc.

%#codegen
y = x + 1;
```

```
end
```

TestAddOne.m

The file `TestAddOne.m` contains a class-based unit test with two tests.

- `reallyAddsOne` verifies that when the input is 1, the answer is 2.
- `addsFraction` verifies that when the input is pi, the answer is pi + 1.

For more information about writing class based-unit tests, see “Author Class-Based Unit Tests in MATLAB” (MATLAB).

```
classdef TestAddOne < matlab.unittest.TestCase
    % Copyright 2014 - 2016 The MathWorks, Inc.

    methods ( Test )

        function reallyAddsOne( testCase )
            x = 1;
            y = addOne( x );
            testCase.verifyEqual( y, 2 );
        end

        function addsFraction( testCase )
            x = pi;
            y = addOne( x );
            testCase.verifyEqual( y, x+1 );
        end
    end
end
```

run_unit_tests.m

The file `run_unit_tests.m` calls `runtests` to run the tests in `TestAddOne.m`.

```
% Run unit tests
% Copyright 2014 - 2016 The MathWorks, Inc.

runtests('TestAddOne')
```

Run Unit Tests on a MEX Function with the MATLAB Coder App

To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

To prepare for code generation, advance through the app steps.

- On the **Select Source Files** page, specify that the entry-point function is `addOne`.
- On the **Define Input Types** page, specify that the input argument `x` is a double scalar.
- On the **Check for Run-Time Issues** step, enter code that calls `addOne` with representative input. For example, `addOne(2)`. Perform this step to make sure that you can generate code for your MATLAB function and that the generated code does not have run-time issues.

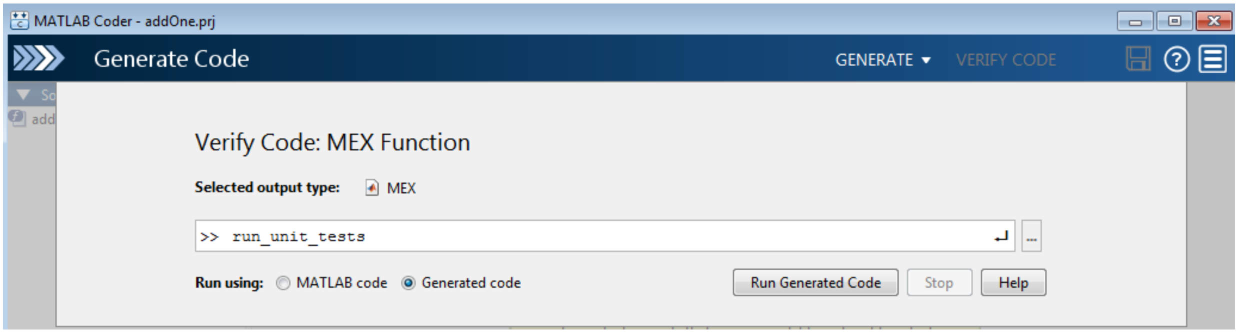
For more complicated MATLAB functions, you might want to provide a test file for the **Define Input Types** and **Check for Run-Time Issues** steps. This test file calls the MATLAB function with representative types. The app uses this file to determine the input types for you. The test file can be different from the test file that you use for unit testing.

To generate the MEX function, on the **Generate Code** page:

1. For **Build type**, specify `MEX`.
2. Click **Generate**.

Run the unit tests on the generated MEX.

1. Click **Verify Code**.
2. In the field for the test file, specify `run_unit_tests`.
3. Make sure that you set **Run using** to **Generated code**.
4. Click **Run Generated Code**.



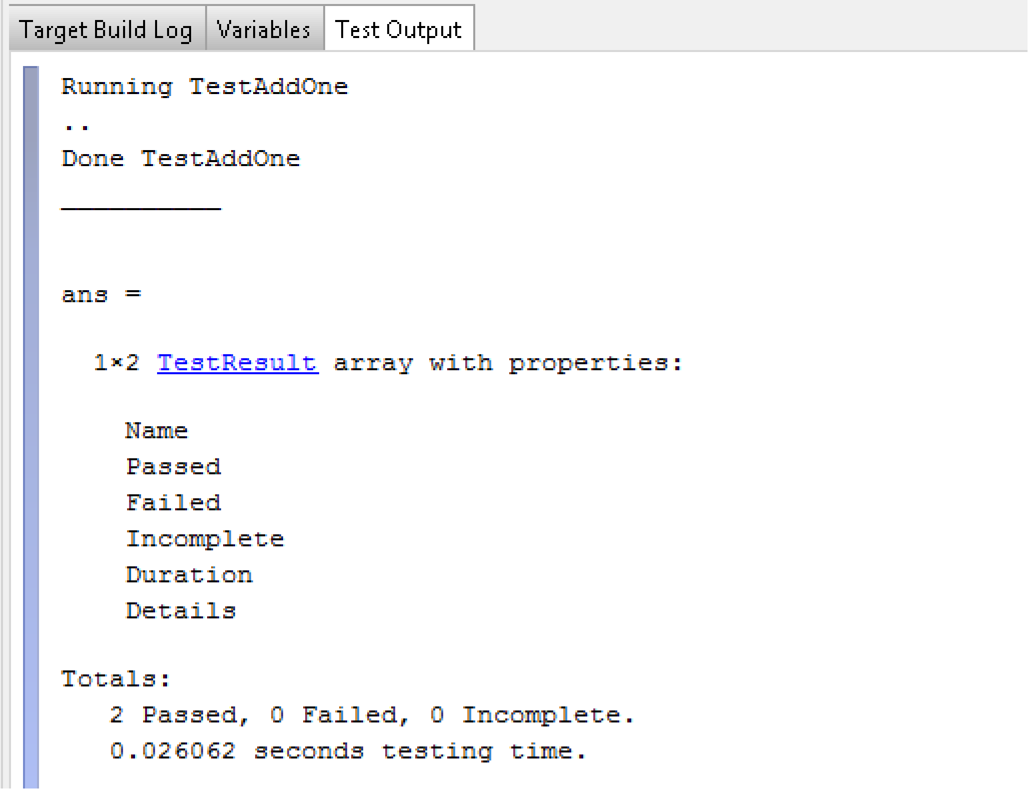
Verify Code: MEX Function

Selected output type: MEX

```
>> run_unit_tests
```

Run using: ○ MATLAB code ● Generated code

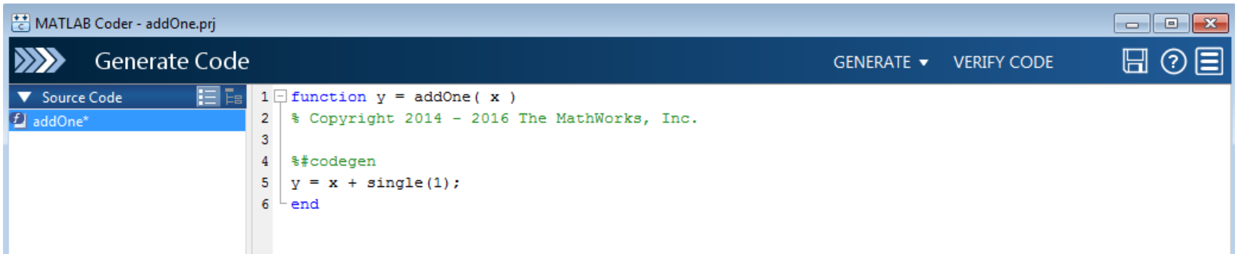
The app displays the test output on the **Test Output** tab. The unit tests pass.



```
Running TestAddOne
..
Done TestAddOne
__________


ans = 

  1×2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.026062 seconds testing time.
```

Run Unit Tests After Modifying MATLAB Code

Modify `addOne` so that the constant 1 is single-precision. To edit `addOne`, in the upper-left corner of the app, under **Source Code**, click `addOne`.



```
function y = addOne( x )
% Copyright 2014 - 2016 The MathWorks, Inc.

%#codegen
y = x + single(1);
end
```

To generate a MEX function for the modified function, click **Generate**.

To run the unit tests:

1. Click **Verify Code**.
2. Make sure that you set the test file to `run_unit_tests` and **Run using** to **Generated code**
3. Click **Run Generated Code**.

The unit tests fail.

- `reallyAddsOne` fails because the class of the output type is single, not double.
- `addsFraction` fails because the output class and value do not match the expected class and value. The output type is single, not double. The value of the single-precision output, 4.1415930, is not the same as the value of the double-precision output, 4.141592653589793.

Run Unit Tests With Software-in-the-Loop Execution in the App (Requires Embedded Coder)

If you have Embedded Coder, you can run the units tests on generated standalone code (static library or shared library) by using software-in-the-loop (SIL) execution.

Generate a library for `addOne`. For example, generate a static library.

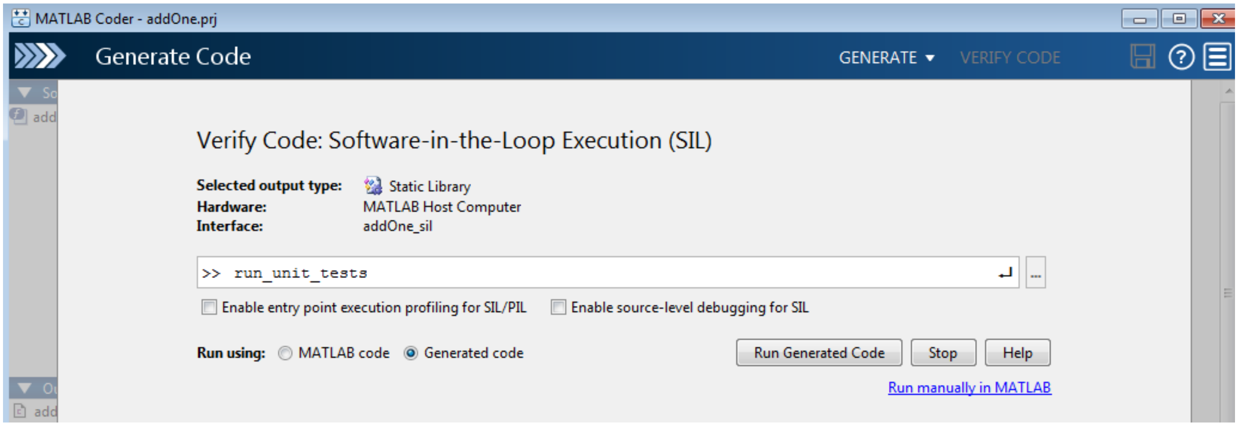
On the **Generate Code** page:

1. For **Build type**, specify `Static Library`.

2 Click **Generate**.

Run the unit tests on the generated code.

1 Click **Verify Code**.
2 In the field for the test file, specify `run_unit_tests`.
3 Make sure that you set **Run using** to **Generated code**.
4 Click **Run Generated Code**.



Verify Code: Software-in-the-Loop Execution (SIL)

Selected output type: Static Library
Hardware: MATLAB Host Computer
Interface: addOne_sil

```
>> run_unit_tests
```

□ Enable entry point execution profiling for SIL/PIL □ Enable source-level debugging for SIL

Run using: ○ MATLAB code ● Generated code

To terminate the SIL execution, click **Stop SIL Verification**.

Run Unit Tests on a MEX Function by Using the Command-Line Workflow

If you use the command-line workflow to generate code, you can run unit tests on a MEX function by using `coder.runTest` with a test file that runs the unit tests.

Generate a MEX function for the function that you want to test. For this example, specify that the input argument is a double scalar by providing a sample input value.

```
codegen addOne -args {2}
```

Run the units tests on the MEX function. Specify that the test file is `run_unit_tests` and that the function is `addOne`. When `coder.runTest` runs the test file, it replaces calls to `addOne` with calls to `addOne_mex`. The unit tests run on the MEX function instead of the original MATLAB function.

```
coder.runTest('run_unit_tests', 'addOne')
```

```
Running TestAddOne
..
Done TestAddOne
__________


ans = 

  1×2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.030621 seconds testing time.
```

Run Unit Tests With Software-in-the-Loop Execution at the Command Line (Requires Embedded Coder)

If you have Embedded Coder, you can run the units tests on generated standalone code (static library or shared library) by using software-in-the-loop (SIL) execution.

Create a `coder.EmbeddedCodeConfig` object for a static library.

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

Configure the object for SIL.

```
cfg.VerificationMode = 'SIL';
```

Generate code for the MATLAB function and the SIL interface.

```
codegen -config cfg -args {2} addOne
```

Run a test file that runs the unit tests with the SIL interface.

```
coder.runTest('run_unit_tests', ['addOne_sil.', mexext])
```

Terminate the SIL execution.

Click **clear addOne_sil**.

See Also

`coder.runTest` | `runtests`

More About

- “Author Class-Based Unit Tests in MATLAB” (MATLAB)
- “Software-in-the-Loop Execution with the MATLAB Coder App” (Embedded Coder)
- “Software-in-the-Loop Execution From Command Line” (Embedded Coder)
- “Unit Test External C Code with MATLAB Coder” on page 21-41

Unit Test External C Code with MATLAB Coder

This example shows how to test external C code by using MATLAB® unit tests with MATLAB® Coder™.

If you want to test C code, you can use MATLAB Coder to bring the code into MATLAB. You can then write unit tests by using the MATLAB testing framework. You can write richer, more flexible tests by taking advantage of the advanced numerical computing and visualization capabilities of MATLAB.

This example shows how to:

1. Bring your C code into MATLAB as a MEX function that you generate with MATLAB Coder.
2. Write a unit test by using the MATLAB testing framework.
3. Run the test on the MEX function.

If you have Embedded Coder®, you can run unit tests on generated standalone code (static library or shared library) by using the unit tests with software-in-the-loop (SIL) execution or processor-in-the-loop (PIL) execution.

Examine the Files

To access the files that this example uses, click **Open Script**.

kalmanfilter.c

`kalmanfilter.c` is the C function that the example tests. It estimates the position of a moving object based on its past positions.

kalmanfilter.h

`kalmanfilter.h` is the header file for `kalmanfilter.c`.

position.mat

`position.mat` contains the positions of the object.

callKalmanFilter.m

`callKalmanFilter` calls `kalmanfilter` by using `coder.ceval`.

```
function [a,b] = callKalmanFilter(position)
```

```
% Copyright 2014 - 2016 The MathWorks, Inc.

numPts = size(position,2);

a = zeros(2,numPts,'double');
b = zeros(2,numPts,'double');
y = zeros(2,1,'double');

% Main loop
for idx = 1: numPts
    z = position(:,idx);     % Get the input data

    % Call the initialize function
    coder.ceval('kalmanfilter_initialize');

    % Call the C function
    coder.ceval('kalmanfilter',z,coder.ref(y));

    % Call the terminate function
    coder.ceval('kalmanfilter_terminate');

    a(:,idx) = [z(1); z(2)];
    b(:,idx) = [y(1); y(2)];
end
end
```

TestKalmanFilter.m

`TestKalmanFilter` tests whether the error between the predicted position and actual position exceeds the specified tolerance. The unit tests are class-based unit tests. For more information, see "Author Class-Based Unit Tests in MATLAB" (MATLAB).

Although you want to test the MEX function, the unit tests in `TestKalmanFilter` call the original MATLAB function from which you generated the MEX function. When MATLAB Coder runs the tests, it replaces the calls to the MATLAB function with calls to the MEX function. You cannot run these tests directly in MATLAB because MATLAB does not recognize the `coder.ceval` calls in `callKalmanFilter`.

```
classdef TestKalmanFilter < matlab.unittest.TestCase
    % Copyright 2014 - 2016 The MathWorks, Inc.

    methods ( Test )
```

```
        function SSE_LessThanTolerance( testCase )
            load position.mat;
            [z,y] = callKalmanFilter( position );

            tolerance = 0.001; % tolerance of 0.0001 will break
            A = z-1000*y;
            error = sum(sum(A.^2));

            testCase.verifyLessThanOrEqual( error, tolerance);

            % For debugging
            plot_kalman_filter_trajectory(z,1000*y);
        end

        function SampleErrorLessThanTolerance( testCase )
            load position.mat;
            [z,y] = callKalmanFilter( position );

            tolerance = 0.01;   % tolerance of 0.001 will break
            A = z-1000*y;

            testCase.verifyEqual(1000*y, z, 'AbsTol', tolerance);
            % For debugging
            plot_kalman_filter_trajectory(z,1000*y);

            [value, location] = max(A(:));
            [R,C] = ind2sub(size(A),location);
            disp(['Max value ' num2str(value) ' is located at [' num2str(R) ',' num2str
        end
    end
end
```

run_unit_tests_kalman.m

run_unit_tests_kalman calls runtests to run the tests in TestKalmanFilter.m.

```
% Run unit tests
% Copyright 2014 - 2016 The MathWorks, Inc.

runtests('TestKalmanFilter')
```

plot_kalman_filter_trajectory.m

plot_kalman_filter_trajectory plots the trajectory of the estimated and actual positions of the object. Each unit test calls this function.

Generate MEX and Run Unit Tests in the MATLAB Coder App

To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

To prepare for code generation, advance through the app steps.

- On the **Select Source Files** page, specify that the entry-point function is callKalmanFilter.
- On the **Define Input Types** page, specify that the input argument x is a 2-by-310 array of doubles.

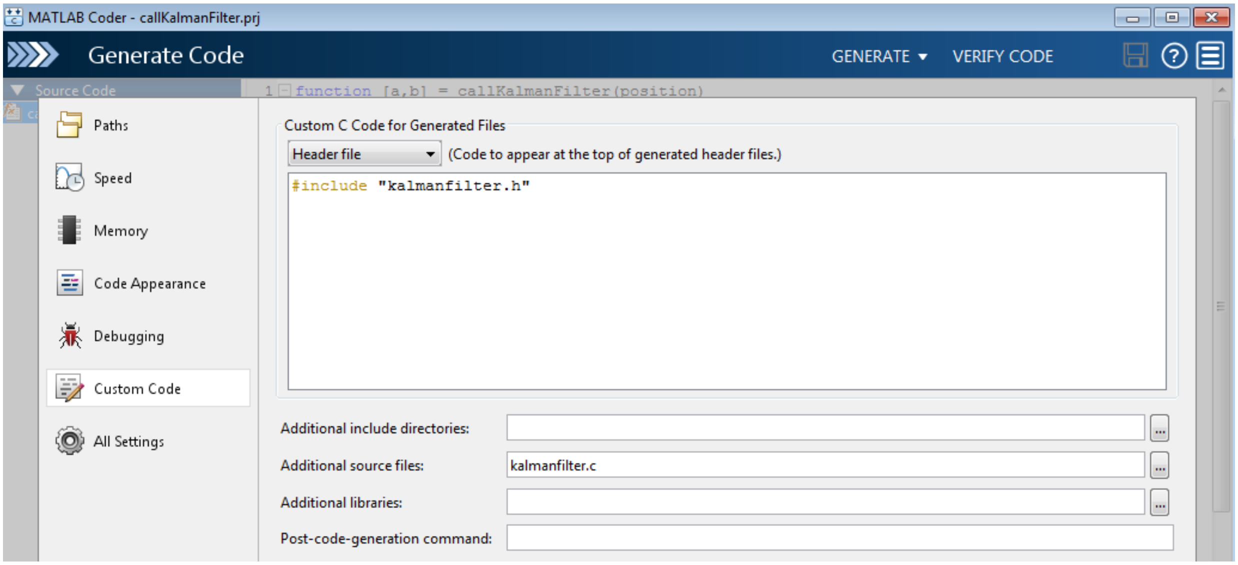
The unit tests load the variable position from position.mat and pass position to callKalmanFilter. Therefore, the input to callKalmanFilter must have the properties that position has. In the MATLAB workspace, if you load position.mat, you see that position is a 2-by-310 array of doubles.

- Skip the **Check for Run-Time Issues** step for this example.

Configure the app for MEX code generation. Specify the names of the C source and header files because callKalmanFilter integrates external C code.

1. For **Build type**, specify MEX.
2. Click **More Settings**.
3. On the **Custom Code** tab:

- Under **Custom C Code for Generated Files**, select **Header file**. In the custom code field, enter #include "kalmanfilter.h".
- In the **Additional source files** field, enter kalmanfilter.c.



Custom C Code for Generated Files

Header file (Code to appear at the top of generated header files.)

```
#include "kalmanfilter.h"
```

Additional include directories:

Additional source files: kalmanfilter.c

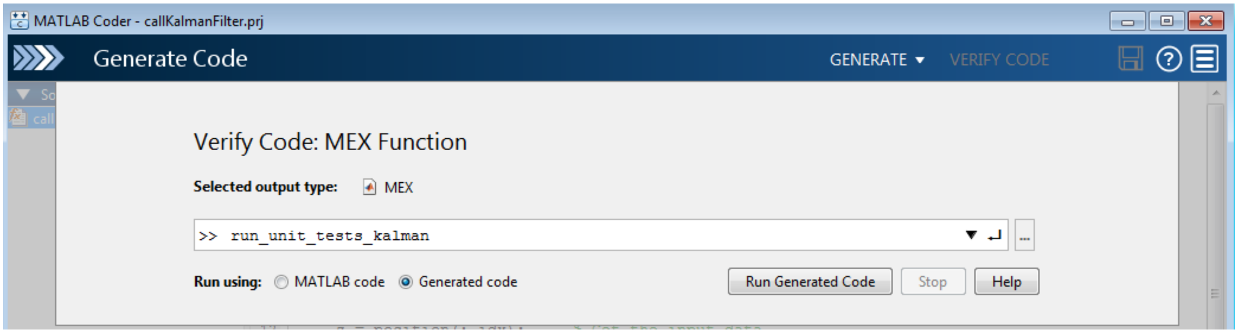
Additional libraries:

Post-code-generation command:

To generate the MEX function, click **Generate**.

Run the unit tests on the generated MEX.

1 Click **Verify Code**.
2 In the field for the test file, specify `run_unit_tests_kalman`.
3 Make sure that you set **Run using** to **Generated code**.
4 Click **Run Generated Code**.



Verify Code: MEX Function

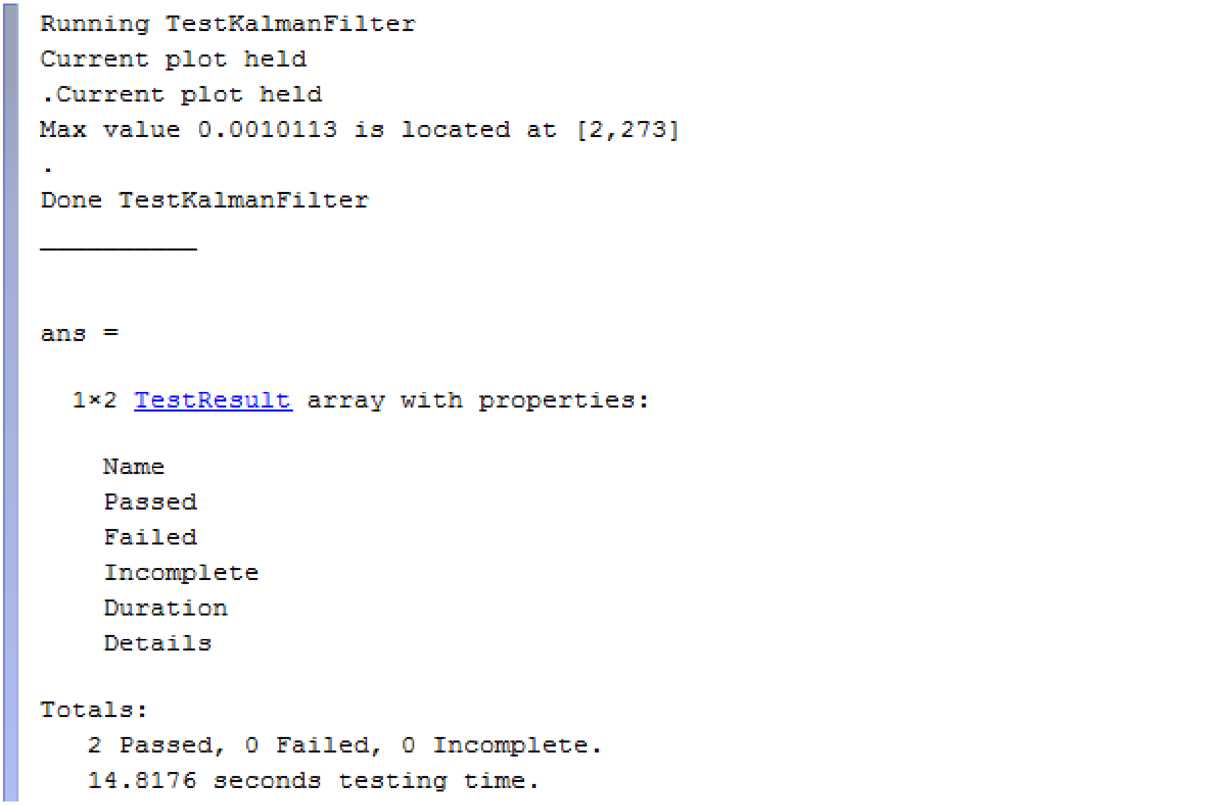
Selected output type: MEX

```
>> run_unit_tests_kalman
```

Run using: ○ MATLAB code ● Generated code

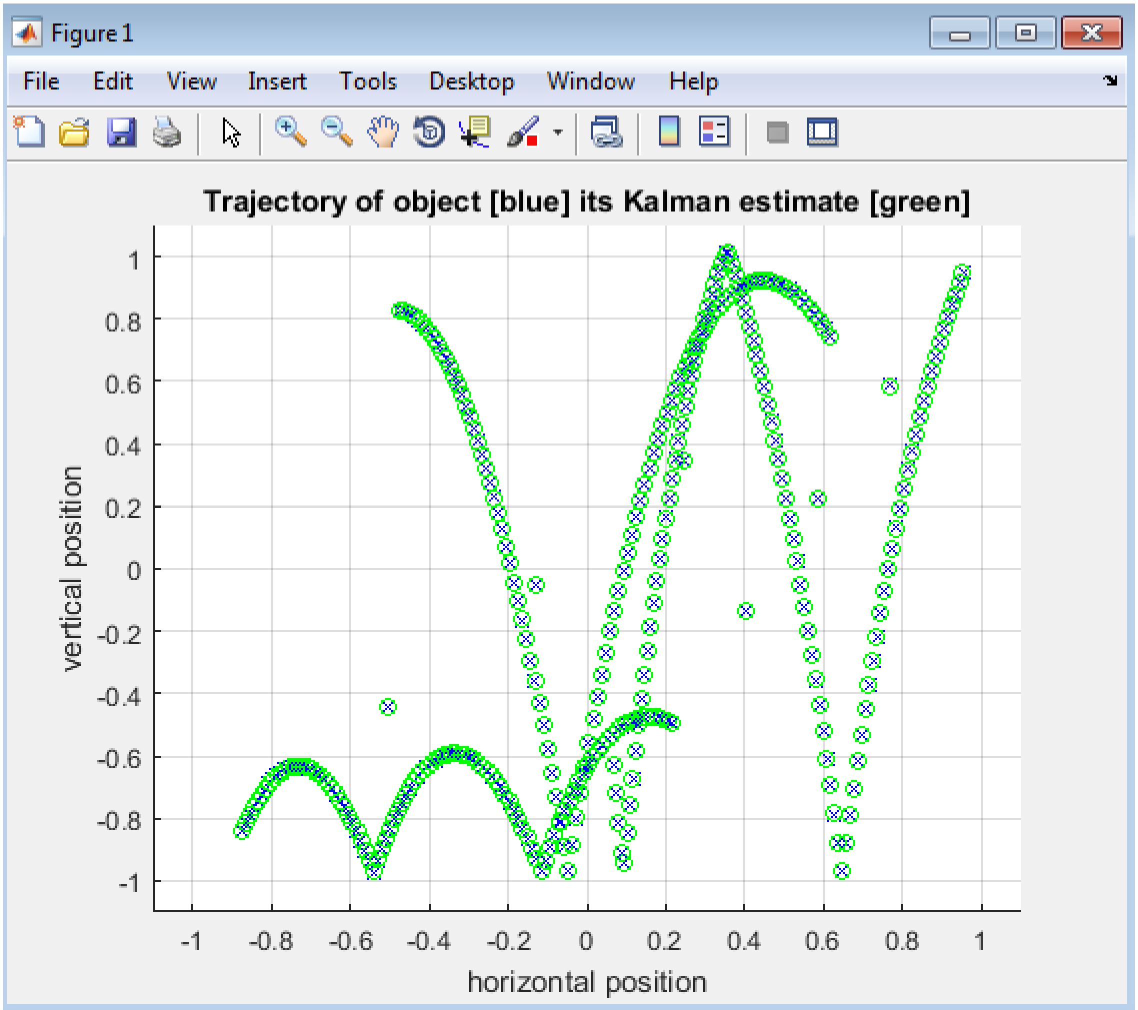
When the app runs the test file, it replaces calls to `callKalmanFilter` in the unit test with calls to `callKalmanFilter_mex`. The unit tests run on the MEX function instead of the original MATLAB function.

The app displays the test output on the **Test Output** tab. The unit tests pass.



```
Running TestKalmanFilter
Current plot held
.Current plot held
Max value 0.0010113 is located at [2,273]
.
Done TestKalmanFilter
__________



ans =

  1×2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   14.8176 seconds testing time.
```

From the plots, you can see that the trajectory of the estimated position converges with the trajectory of the actual position.



Run Unit Tests After Modifying C Code

When you modify the C code, to run the unit tests:

1. Regenerate the MEX function for the MATLAB function that calls the C code.
2. Repeat the verification step.

For example, modify kalmanfilter.c so that the value assigned to y[r2] is multiplied by 1.1.

```
y[r2] += (double)d_a[r2 + (i0 << 1)] * x_est[i0] * 1.1;
```

Edit kalmanfilter.c outside of the app because you can use the app to edit only MATLAB files listed in the **Source Code** pane of the app.

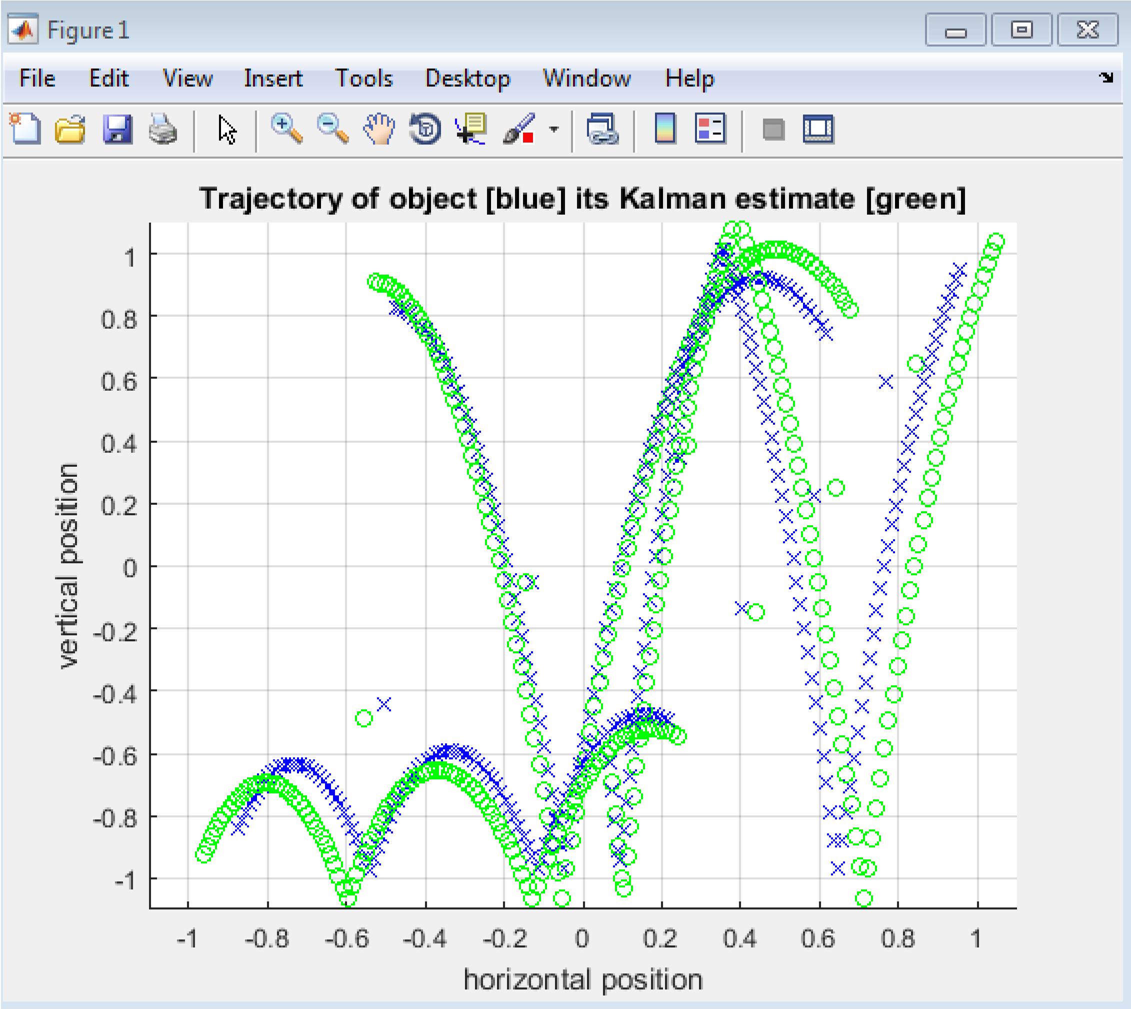
To generate the MEX function for the modified function, click **Generate**.

To run the unit tests:

1 Click **Verify Code**.
2 Make sure that you set the test file to run_unit_tests and **Run using** to **Generated code**
3 Click **Run Generated Code**.

The tests fail because the error exceeds the specified tolerance.

The plots show the error between the trajectory for the estimated position and the trajectory for the actual position.



Generate MEX and Run Unit Tests by Using the Command-Line Workflow

You can use the command-line workflow to run unit tests on external C code by using `coder.runTest`. Specify a test file that runs the unit tests on the MATLAB function that calls your C code.

Generate a MEX function for the MATLAB function that calls your C code. For this example, generate MEX for `callKalmanFilter`.

Create a configuration object for MEX code generation.

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

Specify the external source code and header file.

```
cfg.CustomSource = 'kalmanfilter.c';
cfg.CustomHeaderCode = '#include "kalmanfilter.h"';
```

To determine the type for the input to `callKalmanFilter`, load the position file.

```
load position.mat
```

To generate the MEX function, run `codegen`. Specify that the input to `callKalmanFilter` has the same type as `position`.

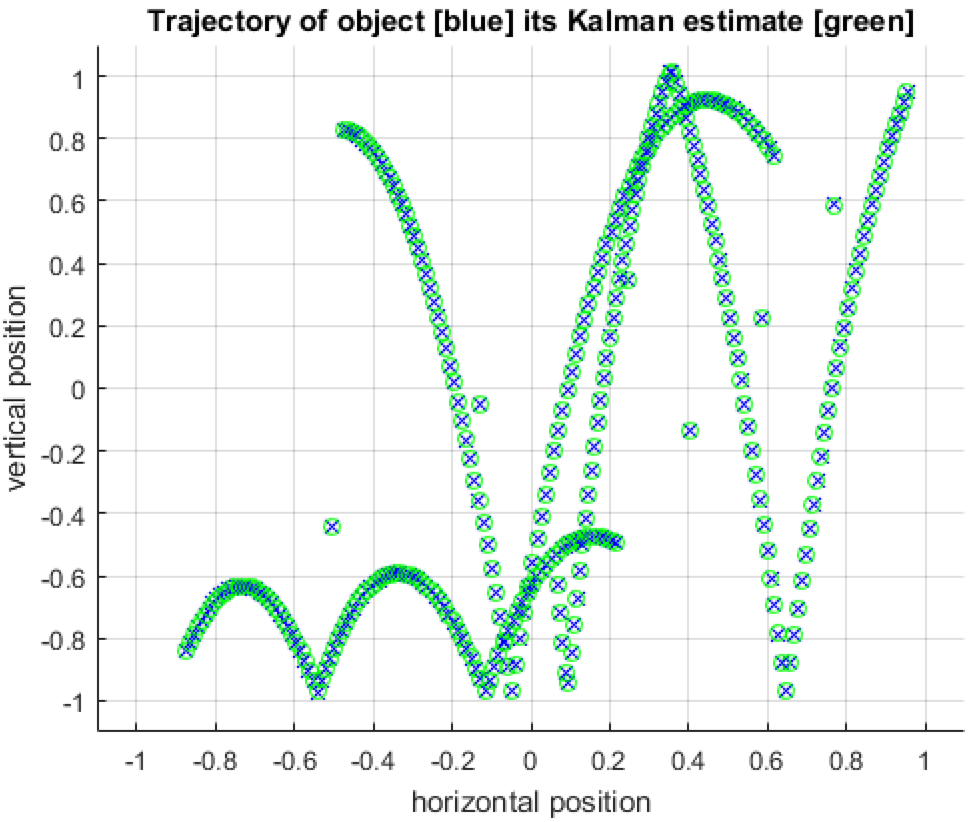
```
codegen -config cfg callKalmanFilter -args position
```

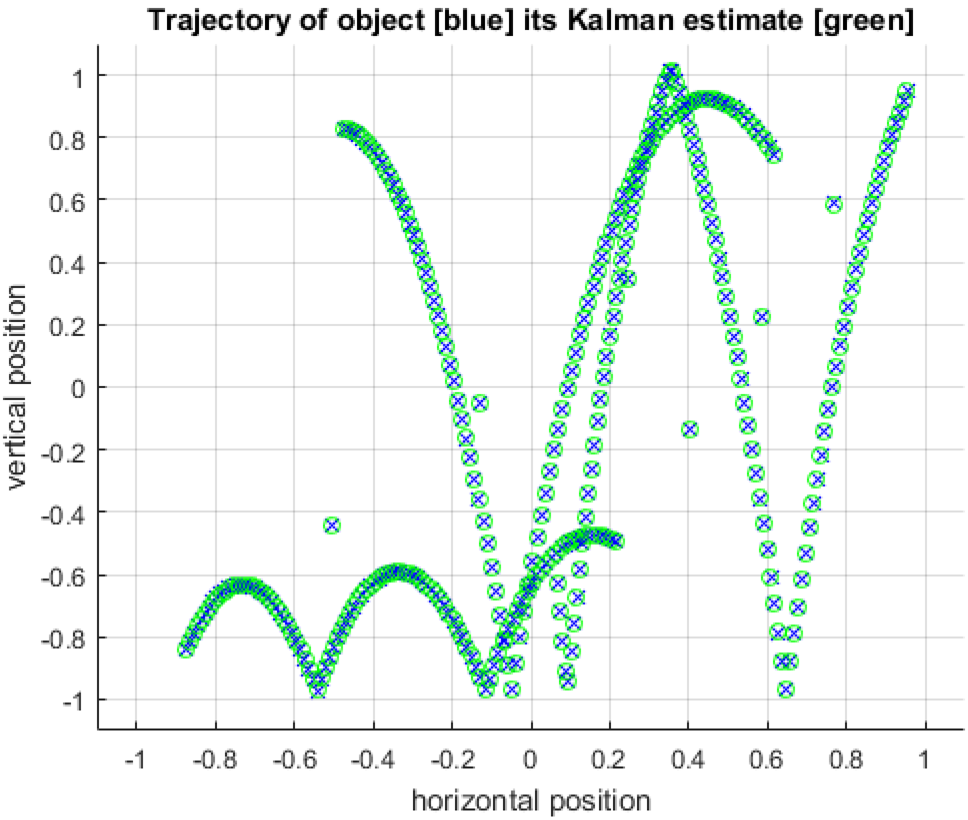
Run the units tests on the MEX function. Specify that the test file is `run_unit_tests_kalman` and that the function is `callKalmanfilter`. When `coder.runTest` runs the test file, it replaces calls to `callKalmanFilter` in the unit test with calls to `callKalmanFilter_mex`. The unit tests run on the MEX function instead of the original MATLAB function.

```
coder.runTest('run_unit_tests_kalman', 'callKalmanFilter')
```

```
Running TestKalmanFilter
Current plot held
.Current plot held
Max value 0.0010113 is located at [2,273]
.
Done TestKalmanFilter
__________


ans = 

  1×2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
```

```
        Details

    Totals:
       2 Passed, 0 Failed, 0 Incomplete.
       16.6895 seconds testing time.
```





See Also

`coder.runTest` | `runtests`

More About

- “Author Class-Based Unit Tests in MATLAB” (MATLAB)
- “Software-in-the-Loop Execution with the MATLAB Coder App” (Embedded Coder)
- “Software-in-the-Loop Execution From Command Line” (Embedded Coder)
- “Unit Test Generated Code with MATLAB Coder” on page 21-33

22

Code Replacement for MATLAB Code

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU[1] gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- AUTOSAR 4.0—Produces code that more closely aligns with the AUTOSAR standard. Requires an Embedded Coder license.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel® Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE for x86-64 (Windows)—Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
- Intel IPP for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.

[1] GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP library for the x86/ Pentium Windows platform.
- Intel IPP/SSE for x86/Pentium (Windows)—Generates calls to the Intel Performance IPP and SSE libraries for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the IPP library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

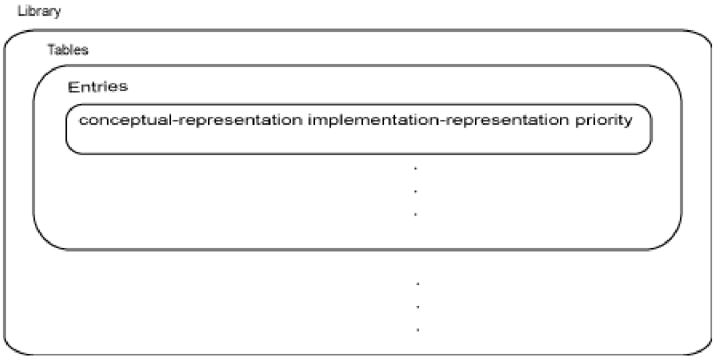

Library
Tables
Entries
conceptual-representation implementation-representation priority

Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.

Table Entry Component	Description
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions

Term	Definition
	and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key `RTW_OP_ADD` identifies the addition operator.
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Code replacement for matrices — Code replacement libraries do not support Dynamic and Symbolic sized matrices.

Related Examples

- “Choose a Code Replacement Library” on page 22-9
- “Replace Code Generated from MATLAB Code” on page 22-11

Choose a Code Replacement Library

In this section...

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

1. Explore available libraries. Identify one that best meets your application needs.
   - Consider the lists of application code replacement requirements and libraries that MathWorks provides in "What Is Code Replacement?" on page 22-2.
   - See "Explore Available Code Replacement Libraries" on page 22-9.
2. Explore the contents of the library. See "Explore Code Replacement Library Contents" on page 22-9.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library. For more information, see "What Is Code Replacement Customization?" (Embedded Coder) in the Embedded Coder documentation.

Explore Available Code Replacement Libraries

You can select the code replacement library to use for code generation in a project, on the **Custom Code** tab, by setting the **Code replacement library** parameter. Alternatively, in a code configuration object, set the `CodeReplacementLibrary` parameter.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

1. At the command prompt, type `crviewer`.

   ```
   >> crviewer
   ```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> crviewer('GNU C99 extensions')
```

2 In the left pane, select the name of a library. The viewer displays information about the library in the right pane.

3 In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.

4 In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

   If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.TflCOperationEntryGenerator` or `RTW.TflCOperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- “What Is Code Replacement?” on page 22-2
- “Replace Code Generated from MATLAB Code” on page 22-11

Replace Code Generated from MATLAB Code

This example shows how to replace generated code using a code replacement library. Code replacement is a technique for changing the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

1. Make sure that you have installed required software. Required software is:

   - MATLAB
   - MATLAB Coder
   - C compiler

   Some code replacement libraries available in your development environment require Embedded Coder.

   For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.
2. Identify an existing MATLAB function or create a new MATLAB function for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore available libraries.

Configure Code Generator To Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the MATLAB function. Do one of the following:

   - In a project, on the **Custom Code** tab, set the **Code replacement library** parameter.
   - In a code configuration object, set the `CodeReplacementLibrary` parameter.
2. Configure the code generator to produce only code. Before you build an executable, verify your code replacements. Do one of the following:

   - In a project, in the **Generate** dialog box, select the **Generate code only** check box.

- In a code configuration object, set the `GenCodeOnly` parameter.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information helps you verify code replacements. For more information, see “Verify Code Replacements” (Embedded Coder) in the Embedded Coder documentation.

Generate Replacement Code

Generate C/C++ code from the MATLAB code. If you configured the code generator to produce a report, generate a code generation report. For example, in the MATLAB Coder app, on the **Generate Code** page, click **Generate**. Or, at the command prompt, enter:

```
codegen -report myFunction -args {5} -config cfg
```

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. Code replacement can sometimes behave differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- “What Is Code Replacement?” on page 22-2
- “Choose a Code Replacement Library” on page 22-9
- “Configure Build Settings” on page 20-26

Custom Toolchain Registration

Custom Toolchain Registration

In this section...

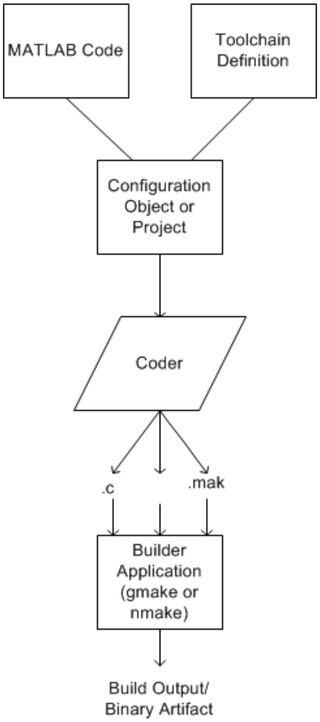
What Is a Custom Toolchain?

You can add support for software build tools to MATLAB Coder software. For example, you can add support for a third-party compiler/linker/archiver (toolchain) to your MATLAB Coder software. This customization can be useful when the added toolchain has support and optimizations for a specific type of processor or hardware. These added toolchains are called *custom toolchains*.

What Is a Factory Toolchain?

MATLAB Coder software includes factory-default support for a set of toolchains. These toolchains are called *factory toolchains* to distinguish them from custom toolchains. If you install factory toolchains on your host computer, MATLAB Coder can automatically detect and use them. Support for factory toolchains depends on the host operating system. Toolchains are identified by the compiler in the toolchain. A complete list of supported toolchains (compilers) is available at http://www.mathworks.com/support/compilers/.

What is a Toolchain Definition?


MATLAB Code
Toolchain Definition
Configuration Object or Project
Coder
.c
.mak
Builder Application (gmake or nmake)
Build Output/ Binary Artifact

A *toolchain definition* provides MATLAB Coder software with information about the software build tools, such as the compiler, linker, archiver. MATLAB Coder software uses this information, along with a configuration object or project, to build the generated code. This approach can be used when generating static libraries, dynamic libraries, and executables. MEX-file generation uses a different approach. To specify which compiler to use for MEX-function generation, see “Setting Up the C or C++ Compiler”.

MATLAB Coder software comes with a set of registered *factory toolchain* definitions. You can create and register *custom toolchain* definitions. You can customize and manage toolchain definitions. You can share custom toolchain definitions with others running MATLAB Coder software.

If you install toolchain software for one of the factory toolchains, MATLAB Coder can automatically detect and use the toolchain software. For more information about factory toolchains in MATLAB Coder software, see http://www.mathworks.com/support/compilers/.

Key Terms

It is helpful to understand the following concepts:

- *Toolchain* — Software that can create a binary executable and libraries from source code. A toolchain can include:
  - *Prebuild tools* that set up the environment
  - *Build tools*, such as an Assembler, C compiler, C++ Compiler, Linker, Archiver, that build a binary executable from source code
  - *Postbuild tools* that download and run the executable on the hardware, and clean up the environment
- *Custom toolchain* — A toolchain that you define and register for use by MATLAB Coder software
- *Factory toolchains* — Toolchains that are predefined and registered in MATLAB Coder software
- *Registered toolchains* — The sum of custom and factory toolchain definitions registered in MATLAB Coder software
- *ToolchainInfo object* — An instance of the `coder.make.ToolchainInfo` class that contains a toolchain definition. You save the `ToolchainInfo` object as a MAT file, register the file with MATLAB Coder. Then you can configure MATLAB Coder to load the `ToolchainInfo` object during code generation.
- *Toolchain definition file* — A MATLAB file that defines the properties of a toolchain. You use this file to create a `ToolchainInfo` object.

Note: This documentation also refers to the `ToolchainInfo` object as a `coder.make.ToolchainInfo` object.

Typical Workflow

The typical workflow for creating and using a custom toolchain definition is:

1. "Create and Edit Toolchain Definition File" on page 23-8

   a. Create a toolchain definition file that returns a `coder.make.ToolchainInfo` object.
   b. Update the file with information about the custom toolchain.
2. "Create and Validate ToolchainInfo Object" on page 23-16
   a. Use the toolchain definition file to create a `ToolchainInfo` object in the MATLAB workspace.
   b. Validate the `ToolchainInfo` object.
   c. Fix validation issues by updating the toolchain definition file, and creating/validating the updated `ToolchainInfo` object.
   d. Create a valid `ToolchainInfo` object and save it to a MAT-file.
3. "Register the Custom Toolchain" on page 23-17
   a. Create an rtwTargetInfo.m file and update it with information about the MAT-file.
   b. Register the custom toolchain in MATLAB Coder software using the rtwTargetInfo.m file.
4. "Use the Custom Toolchain" on page 23-19
   a. Configure MATLAB Coder software to use the custom toolchain.
   b. Build and run an executable using the custom toolchain.

This workflow requires an iterative approach, with multiple cycles to arrive at a finished version of the custom `ToolchainInfo` object. You will need access to detailed information about the custom toolchain.

For a tutorial example of this workflow, see .

For more information about the `ToolchainInfo` object, see "About coder.make.ToolchainInfo" on page 23-6.

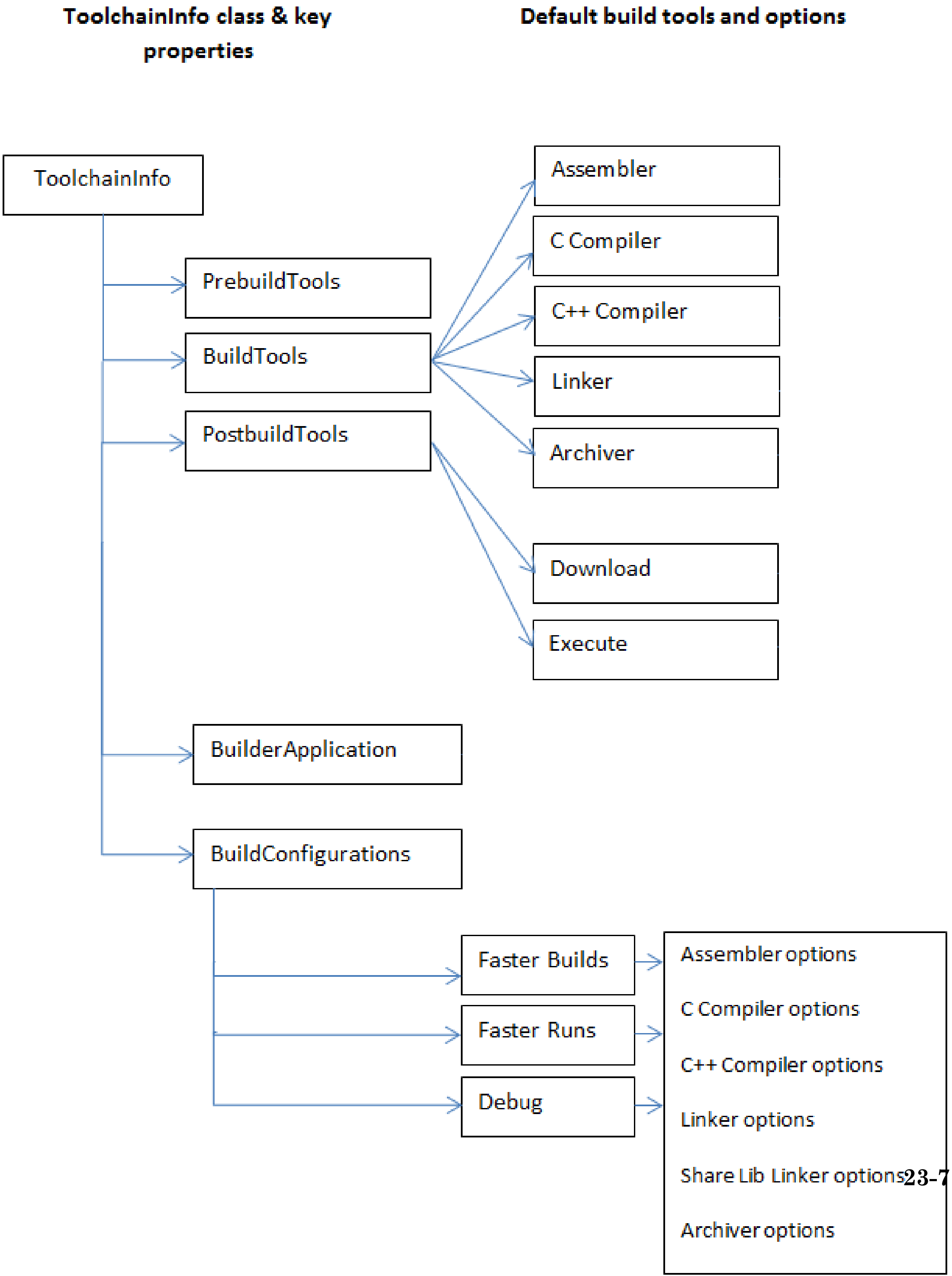
About coder.make.ToolchainInfo

The following properties in `coder.make.ToolchainInfo` represent your custom toolchain:

- `coder.make.ToolchainInfo.PrebuildTools` – Tools used before compiling the source files into object files.
- `coder.make.ToolchainInfo.BuildTools` – Tools used for compiling source files and linking/archiving them to form a binary.
- `coder.make.ToolchainInfo.PostbuildTools` – Tools used after the linker/ archiver is invoked.
- `coder.make.ToolchainInfo.BuilderApplication` – Tools used to call the `PrebuildTools`, `BuildTools`, and `PostbuildTools`. For example: `gmake`, `nmake`.

Each configuration in `coder.make.ToolchainInfo.BuildConfigurations` applies a set of options to the build tools specified by `coder.make.ToolchainInfo.BuildTools`. By default, these configurations alter the way the assembler, compiler, linker, and archiver operate to produce faster builds, faster runs, and debug.

If you instantiate `coder.make.ToolchainInfo` to support building sources that involve assembler, C, or C++ files, the `coder.make.ToolchainInfo` object contains the default set of build tools shown here.


ToolchainInfo class & key properties
Default build tools and options
ToolchainInfo
PrebuildTools
BuildTools
PostbuildTools
BuilderApplication
BuildConfigurations
Assembler
C Compiler
C++ Compiler
Linker
Archiver
Download
Execute
Faster Builds
Faster Runs
Debug
Assembler options
C Compiler options
C++ Compiler options
Linker options
Share Lib Linker options
Archiver options

Create and Edit Toolchain Definition File

This example shows how to create a toolchain definition file by copying and pasting an example file. You then update the relevant elements, and add or remove other elements as needed for your custom toolchain. This is the first step in the typical workflow for creating and using a custom toolchain definition. For more information about the workflow, see “Typical Workflow” on page 23-4.

1. Review the list of registered toolchains. In the MATLAB Command Window, enter:

   ```
   coder.make.getToolchains
   ```

   The resulting output includes the list of factory toolchains for your host computer environment, and previously-registered custom toolchains. For example, the following output shows the factory toolchains for a host computer running 64-bit Windows and no custom toolchains.

   ```
   ans =

       'Microsoft Visual C++ 2012 v11.0 | nmake (64-bit Windows)'
       'Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)'
       'Microsoft Visual C++ 2008 v9.0 | nmake (64-bit Windows)'
       'Microsoft Windows SDK v7.1 | nmake (64-bit Windows)'
   ```

2. Create the folder of example files from the example by entering the following command in the MATLAB Command Window:

   ```
   coderdemo_setup('coderdemo_intel_compiler');
   ```

3. Copy the example toolchain definition file to another location and rename it. For example:

   ```
   copyfile('intel_tc.m','../newtoolchn_tc.m')
   ```

4. Open the new toolchain definition file in the MATLAB Editor. For example:

   ```
   cd ../
   edit newtoolchn_tc.m
   ```

5. Edit the contents of the new toolchain definition file, providing information for the custom toolchain.

   For expanded commentary on an example toolchain definition file, see “Toolchain Definition File with Commentary” on page 23-10.

For reference information about the class attributes and methods you can use in the toolchain definition file, see coder.make.ToolchainInfo.

6 Save your changes to the toolchain definition file.

Next, create and validate a `coder.make.ToolchainInfo` object from the toolchain definition file, as described in “Create and Validate ToolchainInfo Object” on page 23-16

Toolchain Definition File with Commentary

In this section...

Steps Involved in Writing a Toolchain Definition File

This example shows how to create a toolchain definition file and explains each of the steps involved. The example is based on the definition file used in . For more information about the workflow, see “Typical Workflow” on page 23-4.

Write a Function That Creates a ToolchainInfo Object

```
function tc = intel_tc
% INTEL_TC Creates a Intel v12.1 ToolchainInfo object.
% This can be used as a template to add other toolchains on Windows.

%   Copyright 2012 The MathWorks,Inc.

tc = coder.make.ToolchainInfo('BuildArtifact','nmake makefile');
tc.Name             = 'Intel v12.1 | nmake makefile (64-bit Windows)';
tc.Platform         = 'win64';
tc.SupportedVersion = '12.1';

tc.addAttribute('TransformPathsWithSpaces');
tc.addAttribute('RequiresCommandFile');
tc.addAttribute('RequiresBatchFile');
```

The preceding code:

- Defines a function, `intel_tc`, that creates a `coder.make.ToolchainInfo` object and assigns it to a handle, `tc`.

- Overrides the BuildArtifact property to create a makefile for nmake instead of for gmake.
- Assigns values to the Name, Platform, and SupportedVersion properties for informational and display purposes.
- Adds three custom attributes to Attributes property that are required by this toolchain.
- 'TransformPathsWithSpaces' converts paths that contain spaces to short Windows paths.
- 'RequiresCommandFile' generates a linker command file that calls the linker. This avoids problems with calls that exceed the command line limit of 256 characters.
- 'RequiresBatchFile' creates a .bat file that calls the builder application.

Setup

```
% ------------------------------
% Setup
% ------------------------------
% Below we are using %ICPP_COMPILER12% as root folder where Intel Compiler is
% installed. You can either set an environment variable or give full path to the
% compilervars.bat file
tc.ShellSetup{1} = 'call %ICPP_COMPILER12%\bin\compilervars.bat intel64';
```

The preceding code:

- Assigns a system call to the ShellSetup property.
- The coder.make.ToolchainInfo.setup method runs these system calls before it runs tools specified by PrebuildTools property.
- Calls compilervars.bat, which is shipped with the Intel compilers, to get the set of environment variables for Intel compiler and linkers.

Macros

```
% ------------------------------
% Macros
% ------------------------------
tc.addMacro('MW_EXTERNLIB_DIR',['$(MATLAB_ROOT)\extern\lib\' tc.Platform '\microsoft']);
tc.addMacro('MW_LIB_DIR',['$(MATLAB_ROOT)\lib\' tc.Platform]);
tc.addMacro('CFLAGS_ADDITIONAL','-D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('CPPFLAGS_ADDITIONAL','-EHs -D_CRT_SECURE_NO_WARNINGS');
tc.addMacro('LIBS_TOOLCHAIN','$(conlibs)');
tc.addMacro('CVARSFLAG','');

tc.addIntrinsicMacros({'ldebug','conflags','cflags'});
```

The preceding code:

- Uses coder.make.ToolchainInfo.addMacro method to define macros and assign values to them.
- Uses coder.make.ToolchainInfo.addIntrinsicMacros to define macros whose values are specified by the toolchain, outside the scope of your MathWorks software.

C Compiler

```
% ------------------------------
% C Compiler
% ------------------------------

tool = tc.getBuildTool('C Compiler');

tool.setName('Intel C Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.c');
tool.setFileExtension('Header','.h');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

The preceding code:

- Creates a build tool object for the C compiler
- Assigns values to the build tool object properties
- Creates directives and file extensions using name-value pairs
- Sets a command pattern.
- You can use setCommandPattern method to control the use of space characters in commands. For example, the two bars in OUTPUT_FLAG<||>OUTPUT do not permit a space character between the output flag and the output.

C++ Compiler

```
% ------------------------------
% C++ Compiler
% ------------------------------

tool = tc.getBuildTool('C++ Compiler');
```

```
tool.setName('Intel C++ Compiler');
tool.setCommand('icl');
tool.setPath('');

tool.setDirective('IncludeSearchPath','-I');
tool.setDirective('PreprocessorDefine','-D');
tool.setDirective('OutputFlag','-Fo');
tool.setDirective('Debug','-Zi');

tool.setFileExtension('Source','.cpp');
tool.setFileExtension('Header','.hpp');
tool.setFileExtension('Object','.obj');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

The preceding code:

- Creates a build tool object for the C++ compiler
- Is very similar to the build tool object for the C compiler

Linker

```
% ------------------------------
% Linker
% ------------------------------

tool = tc.getBuildTool('Linker');

tool.setName('Intel C/C++ Linker');
tool.setCommand('xilink');
tool.setPath('');

tool.setDirective('Library','-L');
tool.setDirective('LibrarySearchPath','-I');
tool.setDirective('OutputFlag','-out:');
tool.setDirective('Debug','');

tool.setFileExtension('Executable','.exe');
tool.setFileExtension('Shared Library','.dll');

tool.DerivedFileExtensions = horzcat(tool.DerivedFileExtensions,{ ...
                                            ['_' tc.Platform '.lib'],...
                                            ['_' tc.Platform '.exp']});

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

The preceding code:

- Creates a build tool object for the linker
- Assigns values to the `coder.make.BuildTool.DerivedFileExtensions`

Archiver

```
% ------------------------------
% Archiver
% ------------------------------

tool = tc.getBuildTool('Archiver');

tool.setName('Intel C/C++ Archiver');
tool.setCommand('xilib');
tool.setPath('');

tool.setDirective('OutputFlag','-out:');

tool.setFileExtension('Static Library','.lib');

tool.setCommandPattern('|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|');
```

The preceding code:

- Creates a build tool object for the archiver.

Builder

```
% ------------------------------
% Builder
% ------------------------------

tc.setBuilderApplication(tc.Platform);
```

The preceding code:

- Gives the value of `coder.make.ToolchainInfo.Platform` as the argument for setting the value of `BuilderApplication`. This sets the default values of the builder application based on the platform. For example, when `Platform` is `win64`, this line sets the delete command to `'del'`; the display command to `'echo'`, the file separator to `'\'`, and the include directive to `'!include'`.

Build Configurations

```
% --------------------------------------------
% BUILD CONFIGURATIONS
% --------------------------------------------

optimsOffOpts = {'/c /Od'};
optimsOnOpts = {'/c /O2'};
cCompilerOpts    = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL)';
cppCompilerOpts  = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL)';
linkerOpts       = {'$(ldebug) $(conflags) $(LIBS_TOOLCHAIN)'};
sharedLinkerOpts = horzcat(linkerOpts,'-dll -def:$(DEF_FILE)');
archiverOpts     = {'/nologo'};

% Get the debug flag per build tool
debugFlag.CCompiler   = '$(CDEBUG)';
debugFlag.CppCompiler = '$(CPPDEBUG)';
debugFlag.Linker      = '$(LDDEBUG)';
debugFlag.Archiver    = '$(ARDEBUG)';

cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOffOpts));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOffOpts));
cfg.setOption('Linker',linkerOpts);
cfg.setOption('Shared Library Linker',sharedLinkerOpts);
cfg.setOption('Archiver',archiverOpts);

cfg = tc.getBuildConfiguration('Faster Runs');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOnOpts));
cfg.setOption('C++ Compiler',horzcat(cppCompilerOpts,optimsOnOpts));
cfg.setOption('Linker',linkerOpts);
cfg.setOption('Shared Library Linker',sharedLinkerOpts);
cfg.setOption('Archiver',archiverOpts);

cfg = tc.getBuildConfiguration('Debug');
cfg.setOption('C Compiler',horzcat(cCompilerOpts,optimsOffOpts,debugFlag.CCompiler));
cfg.setOption ...
('C++ Compiler',horzcat(cppCompilerOpts,optimsOffOpts,debugFlag.CppCompiler));
cfg.setOption('Linker',horzcat(linkerOpts,debugFlag.Linker));
cfg.setOption('Shared Library Linker',horzcat(sharedLinkerOpts,debugFlag.Linker));
cfg.setOption('Archiver',horzcat(archiverOpts,debugFlag.Archiver));

tc.setBuildConfigurationOption('all','Download','');
tc.setBuildConfigurationOption('all','Execute','');
tc.setBuildConfigurationOption('all','Make Tool','-f $(MAKEFILE)');
```

The preceding code:

- Creates each build configuration object.
- Sets the value of each option for a given build configuration object.

Create and Validate ToolchainInfo Object

This example shows how to create and validate a coder.make.ToolchainInfo object from the toolchain definition file.

Before you create and validate a ToolchainInfo object, create and edit a toolchain definition file, as described in "Create and Edit Toolchain Definition File" on page 23-8.

1. Use the function defined by the toolchain definition file to create a `coder.make.ToolchainInfo` object and assign the object to a handle. For example, the MATLAB Command Window, enter:

   ```
   tc = newtoolchn_tc
   ```
2. Use the `coder.make.ToolchainInfo.validate` method with the `coder.make.ToolchainInfo` object. For example, enter:

   ```
   tc.validate
   ```

   If the `coder.make.ToolchainInfo` object contains errors, the validation method displays error messages in the MATLAB Command Window.
3. Search the toolchain definition file for items named in the error message (without quotes) and update the values.
4. Repeat the process of creating and validating the `ToolchainInfo` object until there are no more errors.

Next, register the custom toolchain, as described in "Register the Custom Toolchain" on page 23-17.

For more information, see "Troubleshooting Custom Toolchain Validation" on page 23-20.

Register the Custom Toolchain

Before you register the custom toolchain, create and validate the `ToolchainInfo` object, as described in “Create and Validate ToolchainInfo Object” on page 23-16.

1. Use the `save` function to create a MATLAB-formatted binary file (MAT-file) from the `coder.make.ToolchainInfo` object in the MATLAB workspace variables. For example, enter:

   ```
   save newtoolchn_tc tc
   ```

   The new `.mat` file appears in the Current Folder.
2. Create a new MATLAB function called `rtwTargetInfo.m`.
3. Copy and paste the following text into `rtwTargetInfo.m`:

```
function rtwTargetInfo(tr)
% RTWTARGETINFO Target info callback

tr.registerTargetInfo(@loc_createToolchain);

end

% -------------------------------------------------------------------------
% Create the ToolchainInfoRegistry entries
% -------------------------------------------------------------------------
function config = loc_createToolchain

  config(1)           = coder.make.ToolchainInfoRegistry;
  config(1).Name      = '<mytoolchain v#.#> | <buildartifact (platform)>';
  config(1).FileName  = fullfile('<yourdir>','<mytoolchain_tc.mat>');
  config(1).TargetHWDeviceType   =  {'<devicetype>'};
  config(1).Platform  = {'<win64>'};

% To register more custom toolchains:
% 1) Copy and paste the five preceding 'config' lines.
% 2) Increment the index of config().
% 3) Replace the values between angle brackets.
% 4) Remove the angle brackets.

end
```

4. Replace the items between angle brackets with real values, and remove the angle brackets:

   - `Name` — Provide a unique name for the toolchain definition file using the recommended format: name, version number, build artifact, and platform.
   - `FileName` — The full path and name of the MAT-file.
   - `TargetHWDeviceType` — The platform or platforms supported by the custom toolchain.

- Platform — The host operating system supported by the custom toolchain. For all platforms, use the following wildcard: '*'

For more information, refer to the corresponding ToolchainInfo properties in “Properties”.

Here are some example entries for an Intel toolchain that uses nmake, based on :

```
config(1)           = coder.make.ToolchainInfoRegistry;
config(1).Name      = 'Intel v12.1 | nmake makefile (64-bit Windows)';
config(1).FileName  = fullfile(fileparts(mfilename('fullpath')),'intel_tc.mat');
config(1).TargetHWDeviceType  = {'ARM9','ARM10','ARM11'};
config(1).Platform  = {computer('arch')};
```

5. Save the new rtwTargetInfo.m file to a folder that is on the MATLAB path.
6. List all of the rtwTargetInfo.m files on the MATLAB path. Using the MATLAB Command Window, enter:

   ```
   which -all rtwTargetInfo
   ```
7. Verify that the rtwTargetInfo.m file you just created appears in the list of files.
8. Reset TargetRegistry so it picks up the custom toolchain from the rtwTargetInfo.m file:

   ```
   RTW.TargetRegistry.getInstance('reset');
   ```

Next, use the custom toolchain, as described in “Use the Custom Toolchain” on page 23-19.

Use the Custom Toolchain

You can use a custom toolchain when generating a static or dynamic library or an executable. You cannot use one to generate MEX functions. To specify which compiler to use for MEX-function generation, see “Setting Up the C or C++ Compiler”).

Before using the custom toolchain, register the custom toolchain, as described in “Register the Custom Toolchain” on page 23-17.

1. Use `coder.config` to create a configuration object. For example:

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

2. Get the value of `config(end).Name` from the `rtwTargetInfo.m` file. Then assign that value to the `cfg.Toolchain` property:

   ```
   cfg.Toolchain = 'mytoolchain v#.#' | 'buildartifact (platform)'
   ```

   With the example, this would look like:

   ```
   cfg.Toolchain = 'Intel v12.1 | nmake makefile (64-bit Windows)';
   ```

3. Perform other steps required to generate code, as described in “Deployment”. For example, specify the path and file name of the source code:

   ```
   cfg.CustomSource = 'filename_main.c';
   cfg.CustomInclude = pwd;
   ```

4. When you generate code using the `codegen` function, specify the configuration object that uses the custom toolchain. For example:

   ```
   codegen -config cfg filename
   ```

You have completed the full workflow of creating and using a custom toolchain described in “Custom Toolchain Registration” on page 23-2.

Troubleshooting Custom Toolchain Validation

In this section...

Build Tool Command Path Incorrect

If the path or command file name are not correct, validation displays:

```
Cannot find file 'path+command'. The file does not exist.
```

Consider the following two lines from an example toolchain definition file:

```
tool.setCommand('abc');
tool.setPath('/toolchain/');
```

To correct this issue:

- Check that the build tool is installed.
- Review the arguments given for the `tool.setCommand` and `tool.setPath` lines in toolchain definition file.

Build Tool Not in System Path

When the build tool’s path is not provided and the command file is not in the system path, validation displays:

```
Cannot find 'command'. It is not in the system path.
```

Consider the following two lines from an example toolchain definition file:

```
tool.setCommand('icl');
tool.setPath('');
```

Because the argument for `setPath()` is `''` instead of an absolute path, the build tool must be on the system path.

To correct this issue:

- Use `coder.make.ToolchainInfo.ShellSetup` property to add the path to the toolchain installation.
- Use your system setup to add the toolchain installation directory to system environment path.

Otherwise, replace `''` with the absolute path of the command file.

Tool Path Does Not Exist

If the path of the build tool path is provided, but does not exist, validation displays:

```
Path 'toolpath' does not exist.
```

To correct this issue:

- Check the actual path of the build tool. Then, update the value of `coder.make.BuildTool.setPath` in the toolchain definition file.
- Use your system setup to add the toolchain installation directory to system environment path. Then, set the value of `coder.make.BuildTool.setPath` to `''`.

Unsupported Platform

If the toolchain is not supported on the host computer platform, validation displays:

```
Toolchain 'tlchn' is supported on a 'pltfrma' platform. However, you are running on a 'pltfrmb' platform.
```

To correct this issue:

- Check the `coder.make.ToolchainInfo.Platform` property in your toolchain definition file for errors.
- Update or replace the toolchain definition file with one that supports your host computer platform.
- Change host computer platforms.

Toolchain is Not installed

If the toolchain is not installed, validation displays:

```
Toolchain is not installed
```

To correct this issue, install the expected toolchain, or verify that you selected the correct toolchain, as described in "Use the Custom Toolchain" on page 23-19.

Project or Configuration is Using the Template Makefile

By default, MATLAB Coder tries to use the selected build toolchain to build the generated code. However, if the makefile configuration options detailed in the following sections are **not** set to their default value, MATLAB Coder cannot use the toolchain and reverts to using the template makefile approach for building the generated code.

MATLAB Coder Project Settings

Project Settings Dialog Box All Settings Parameter Name	Default Setting
Generate makefile	`Yes`
Make command	`make_rtw`
Template makefile	`default_tmf`
Compiler optimization level	`Off`

Command-line Configuration Parameters for the codegen function

coder.CodeConfig or coder.EmbeddedCodeConfig Parameter Name	Default Value
`GenerateMakefile`	`'true'`
`MakeCommand`	`'make_rtw'`
`TemplateMakefile`	`'default_tmf'`
`CCompilerOptimization`	`'Off'`

To use the toolchain approach, reset your configuration options to these default values manually or:

- To reset settings for project `project_name`, at the MATLAB command line, enter:

```
coder.make.upgradeMATLABCoderProject(project_name)
```

- To reset command-line settings for configuration object `config`, create an updated configuration object `new_config` and then use `new_config` with the `codegen` function in subsequent builds. At the MATLAB command line, enter:

  ```
  new_config = coder.make.upgradeCoderConfigObject(config);
  ```

Skipped Validation of Build Tool “Download” or “Execute”

Even though the Validation Report states “Toolchain Validation Result: Passed” it includes one or both of the following notes:

```
### Validation of build tool "Download"
Skipped. No "Download" build tool is specified.
### Validation of build tool "Execute"
Skipped. "Execute" build tool "$(PRODUCT)" cannot be validated.
```

To correct this issue, update the toolchain definition file and re-register the updated toolchain. For more information, see:

- “Create and Edit Toolchain Definition File” on page 23-8
- “Create and Validate ToolchainInfo Object” on page 23-16
- “Register the Custom Toolchain” on page 23-17

Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers

Symptom: During a software build, a build error occurs; variables don't resolve correctly.

If your toolchain uses a one-pass or single-pass linker, prevent circular data dependencies by adding the StartLibraryGroup and EndLibraryGroup linker directives to the toolchain definition file.

For example, if the linker is like GNU gcc, then the directives are `'-Wl,--start-group'` and `'-Wl,--end-group'`, as shown here:

```
% ------------------------------
% Linker
% ------------------------------

tool = tc.getBuildTool('Linker');

tool.setName(           'GNU Linker');
tool.setCommand(        'gcc');
tool.setPath(           '');

tool.setDirective(      'Library',                          '-l');
tool.setDirective(      'LibrarySearchPath',                '-L');
tool.setDirective(      'OutputFlag',                       '-o');
tool.setDirective(      'Debug',                            '-g');
tool.addDirective(      'StartLibraryGroup', {'-Wl,--start-group'});
tool.addDirective(      'EndLibraryGroup',     {'-Wl,--end-group'});

tool.setFileExtension(  'Executable',       '');
tool.setFileExtension(  'Shared Library',   '.so');
```

Deploying Generated Code

Using C/C++ Code That MATLAB Coder Generates

With MATLAB Coder, you can generate C/C++ source code, a static library, a dynamically linked library, or an executable. How you use the generated code depends on your goal.

Goal	See
Package generated files into a zip file for relocation to another development environment.	"Package Code for Other Development Environments" on page 24-46
Call generated code from MATLAB code.	"External Code Integration for Code Generation" on page 27-2
Generate an example C/C++ main function. Use that function to integrate generated code into a C application.	"Use an Example C Main in an Application" on page 24-23
Integrate generated code into a C/C++ application.	"Use a C Dynamic Library in a Microsoft Visual Studio Project" on page 24-11
Integrate generated code that uses emxArrays.	• "Use an Example C Main in an Application" on page 24-23 • "C Code Interface for Arrays" on page 6-15
Generate a C/C++ Executable.	"Generating Standalone C/C++ Executables from MATLAB Code" on page 20-14

C Compiler Considerations for Signed Integer Overflows

The code generator reduces memory usage and enhances performance of code that it produces by assuming that signed integer C operations wrap on overflow. A signed integer overflow occurs when the result of an arithmetic operation is outside the range of values that the output data type can represent. The C programming language does not define the results of such operations. Some C compilers aggressively optimize signed operations for in-range values at the expense of overflow conditions. Other compilers preserve the full wrap-on-overflow behavior. For example, the gcc and MinGW compilers provide an option to reliably wrap overflow on signed integer overflows.

When you generate code, if you use a supported compiler with the default options configured by the code generator, the compiler preserves the full wrap-on-overflow behavior. If you change the compiler options or compile the code in another development environment, it is possible that the compiler does not preserve the full wrap-on-overflow behavior. In this case, the executable program can produce unpredictable results.

If this issue is a concern for your application, consider one or more of the following actions:

- Verify that the compiled code produces the expected results.
- If your compiler has an option to force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, configure the build process to use the compiler option `-fwrapv`.
- Choose a compiler that wraps on integer overflow.
- If you have Embedded Coder installed, develop and apply a custom code replacement library to replace code generated for signed integers. For more information, see "Code Replacement Customization" (Embedded Coder).

More About

- "Setting Up the C or C++ Compiler"
- Supported and Compatible Compilers

Call a Generated C Static Library Function from C Code

This example shows how to call a generated C library function from C code. It uses the C static library function absval described in “Call a C/C++ Static Library Function from MATLAB Code” on page 24-6.

1 Write a main function in C that does the following:

   - Includes the generated header file, which contains the function prototypes for the library function.
   - Calls the initialize function before calling the library function for the first time.
   - Calls the terminate function after calling the library function for the last time.

   Here is an example of a C main function that calls the library function absval:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "absval.h"

int main(int argc, char *argv[])
{
    absval_initialize();

    printf("absval(-2.75)=%g\n", absval(-2.75));

    absval_terminate();

    return 0;
}
```

2 Configure your target to integrate this custom C main function with your generated code, as described in “Specify External File Locations” on page 24-14.

   For example, you can define a configuration object that points to the custom C code:

   a Create a configuration object. At the MATLAB prompt, enter:

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

   b Set custom code properties on the configuration object, as in these example commands:

```
    cfg.CustomSource = 'main.c';
    cfg.CustomInclude = 'c:\myfiles';
```

3. Generate the C executable. Use the `-args` option to specify that the input is a real, scalar double. At the MATLAB prompt, enter:

   ```
   codegen -config cfg  absval -args {0}
   ```

4. Call the executable. For example:

   ```
   absval(-2.75)
   ```

More About

- "Call Generated C/C++ Functions" on page 24-8
- "Generating Standalone C/C++ Executables from MATLAB Code" on page 20-14

Call a C/C++ Static Library Function from MATLAB Code

This example shows how to call a C/C++ library function from MATLAB code that is suitable for code generation.

Suppose you have a MATLAB file `absval.m` that contains the following function:

```
function y = absval(u) %#codegen
  y = abs(u);
end
```

To generate a C static library function and call it from MATLAB code:

1. Generate the C library for `absval.m`.

   ```
   codegen -config:lib absval -args {0.0}
   ```

   Here are key points about this command:

   - The `-config:lib` option instructs MATLAB Coder to generate `absval` as a C static library function.

     The default target language is C. To change the target language to C++, see “Specify a Language for Code Generation” on page 20-28.
   - MATLAB Coder creates the library `absval.lib` (or `absval.a` on Linus Torvalds' Linux) and header file `absval.h` in the folder `/emcprj/rtwlib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the C library.
   - The `-args` option specifies the class, size, and complexity of the primary function input `u` by example, as described in “Define Input Properties by Example at the Command Line” on page 20-51.
2. Write a MATLAB function to call the generated library:

   ```
   %#codegen
   function y = callabsval

   % Call the initialize function before
   % calling the C function for the first time
   coder.ceval('absval_initialize');

   y = -2.75;
   y = coder.ceval('absval',y);
   ```

```
% Call the terminate function after
% calling the C function for the last time
coder.ceval('absval_terminate');
```

The MATLAB function `callabsval` uses the interface `coder.ceval` to call the generated C functions `absval_initialize`, `absval`, and `absval_terminate`. You must use this function to call C functions from generated code. For more information, see "Call Generated C/C++ Functions" on page 24-8.

3 Convert the code in `callabsval.m` to a MEX function so that you can call the C library function `absval` directly from the MATLAB prompt.

a Generate the MEX function using `codegen` as follows:

- Create a code generation configuration object for a MEX function:

```
cfg = coder.config
```

- On Microsoft Windows platforms, use this command:

```
codegen -config cfg callabsval codegen/lib/absval/absval.lib
    codegen/lib/absval/absval.h
```

By default, this command creates, in the current folder, a MEX function named `callabsval_mex`

On the Linus Torvalds' Linux platform, use this command:

```
codegen -config cfg  callabsval codegen/lib/absval/absval.a
   codegen/lib/absval/absval.h
```

b At the MATLAB prompt, call the C library by running the MEX function. For example, on Windows:

```
callabsval_mex
```

Call Generated C/C++ Functions

In this section...

Conventions for Calling Functions in Generated Code

When generating code, MATLAB Coder uses the following calling conventions:

- Passes arrays by reference as inputs.
- Returns arrays by reference as outputs.
- Unless you optimize your code by using the same variable as both input and output, passes scalars by value as inputs. In that case, MATLAB Coder passes the scalar by reference.
- Returns scalars by value for single-output functions.
- Returns scalars by reference:
  - For functions with multiple outputs.
  - When you use the same variable as both input and output.

For more information about optimizing your code by using the same variable as both input and output, see "Eliminate Redundant Copies of Function Inputs" on page 28-7.

How to Call C/C++ Functions from MATLAB Code

You can call the C/C++ functions generated for libraries as custom C/C++ code from MATLAB functions that are suitable for code generation. For static libraries, you must use the `coder.ceval` function to wrap the function calls, as in this example:

```
function y = callmyCFunction %#codegen
```

```
  y = 1.5;
  y = coder.ceval('myCFunction',y);
end
```

Here, the MATLAB function `callmyCFunction` calls the custom C function `myCFunction`, which takes one input argument.

For dynamically-linked libraries, you can also use `coder.ceval`.

There are additional requirements for calling C/C++ functions from the MATLAB code in the following situations:

- You want to call generated C/C++ libraries or executables from a MATLAB function. Call housekeeping functions generated by MATLAB Coder, as described in “Calling Initialize and Terminate Functions” on page 24-9.
- You want to call C/C++ functions that are generated from MATLAB functions that have more than one output, as described in “Calling C/C++ Functions with Multiple Outputs” on page 24-10.
- You want to call C/C++ functions that are generated from MATLAB functions that return arrays, as described in “Calling C/C++ Functions that Return Arrays” on page 24-10.

Calling Initialize and Terminate Functions

When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder automatically generates two housekeeping functions that you must call along with the C/C++ function.

Housekeeping Function	When to Call
primary_function_name`_initialize`	Before you call your C/C++ executable or library function for the first time
primary_function_name`_terminate`	After you call your C/C++ executable or library function for the last time

From C/C++ code, you can call these functions directly. However, to call them from MATLAB code that is suitable for code generation, you must use the `coder.ceval` function. `coder.ceval` is a MATLAB Coder function, but is not supported by the native MATLAB language. Therefore, if your MATLAB code uses this function, use `coder.target` to disable these calls in MATLAB and replace them with equivalent functions.

Calling C/C++ Functions with Multiple Outputs

Although MATLAB Coder can generate C/C++ code from MATLAB functions that have multiple outputs, the generated C/C++ code cannot return multiple outputs directly because the C/C++ language does not support multiple return values. Instead, you can achieve the effect of returning multiple outputs from your C/C++ function by using `coder.wref` with `coder.ceval`.

Calling C/C++ Functions that Return Arrays

Although MATLAB Coder can generate C/C++ code from MATLAB functions that return values as arrays, the generated code cannot return arrays *by value* because the C/C++ language is limited to returning single, scalar values. Instead, you can return arrays from your C/C++ function *by reference* as pointers by using `coder.wref` with `coder.ceval`.

Use a C Dynamic Library in a Microsoft Visual Studio Project

This example shows how to create and configure a simple Microsoft Visual Studio Win32 Console Application project that calls a dynamic library (DLL) that MATLAB Coder generates. This example uses Microsoft Visual Studio 2013. In other versions of Microsoft Visual Studio, you might encounter a different procedure.

Generate a C Dynamic Library

1 Create a MATLAB function `foo`.

```
function c = foo(a) %#codegen
  c = sqrt(a);
end
```

2 Save it as `foo.m` in a local writable folder, for example, `c:\dll_test`.

3 Generate a DLL for the MATLAB function `foo`. Use the `-args` option to specify that the input `a` is a real double.

```
codegen -report -config:dll  foo -args {0}
```

On Microsoft Windows systems, `codegen` generates a C dynamic library, `foo.dll`, and supporting files in the default folder, `codegen/dll/foo`.

Create a Microsoft Visual Studio Project

In Microsoft Visual Studio, create an empty Win32 Console Application project. In Microsoft Visual Studio 2013:

1 On the Start page window, select **File** > **New** > **Project**.

2 In the New Product dialog box, select **Installed** > **Templates** > **Visual C++** > **Win32** > **Win32 Console Application** and enter a name.

3 In the Win32 Application Wizard, select **Application Settings**. Select the **Empty project** check box.

4 Click **Finish**.

Configure the Platform

Verify that the project configuration specifies the architecture that matches your computer. By default, MATLAB Coder builds a DLL for the platform that you are working on, but Microsoft Visual Studio builds for Win32. In Microsoft Visual Studio 2013:

1 Select **Build** > **Configuration Manager**.
2 In the Configuration Manager, set **Active solution platform** to match your platform.

Configure the Solution Version

Configure the project to use the release version of the C run-time library. By default, the Microsoft Visual Studio project uses the debug version of the C run-time library. However, by default, the DLL that MATLAB Coder generates uses the release version. In Microsoft Visual Studio 2013:

1 Select **Build** > **Configuration Manager**.
2 In the Configuration Manager, set **Active solution configuration** to `Release`.

Configure Additional Directories and Dependencies

1 Select **Project** > **Properties**.
2 Under **Configuration Properties** > **C/C++** > **General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Include Directories**.
3 Under **Configuration Properties** > **Linker** > **General**, add the folder `c:\dll_test\codegen\dll\foo` to **Additional Library Directories**.
4 Under **Configuration Properties** > **Linker** > **Input**, add `foo.lib` to **Additional Dependencies**.

Create a main.c File

Create a `main.c` file that calls `foo.dll`. The `main.c` function must:

- Include the generated header file, which contains the function prototypes for the library function.
- Call the initialize function before calling the library function for the first time.
- Call the terminate function after calling the library function for the last time.

For example:

```
#include "foo.h"
#include "foo_initialize.h"
#include "foo_terminate.h"
#include <stdio.h>
```

```
int main()
{
   foo_initialize();
   printf("%f\n", foo(25));
   foo_terminate();
   getchar();
   return 0;
}
```

Add the main.c File to the Project

1. Select **Project** > **Add Existing Item**.
2. Navigate to the folder that contains the main.c file.
3. Select the main.c file.

Build and Run the Executable

1. Build the executable. Select **Build** > **Build Solution**.
2. Make the `.dll` accessible to the executable. Either copy `foo.dll` to the folder containing the executable or add the folder containing `foo.dll` to your path.
3. Run the executable.

More About

- "Call Generated C/C++ Functions" on page 24-8
- "Generating C/C++ Dynamically Linked Libraries from MATLAB Code" on page 20-9

Specify External File Locations

In this section...

External File Locations for External Code Integration

To integrate external code with generated C/C++ code, you must specify the locations of your external source files, header files, and libraries to MATLAB Coder.

You can specify the file locations:

- In a class definition file, when you derive a class from `coder.ExternalDependency`
- In your MATLAB code using the `coder.updateBuildInfo` function
- In the project settings dialog box
- From the command line
- In the configuration object

Specify External Files in a Class Derived from coder.ExternalDependency

When you derive a class from `coder.ExternalDependency`, you write a method `updateBuildInfo` that specifies the locations of the external files required for the build. See coder.ExternalDependency.

Specify External Files in MATLAB Code Using coder.updateBuildInfo

In your MATLAB code, you can call `coder.updateBuildInfo` to specify the locations of external files. See `coder.updateBuildInfo`.

Specify External Files Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow ▼.
2. In the **Generate** dialog box, set the **Build Type** to one of the following:
   - `Source Code`
   - `Static Library`
   - `Dynamic Library`
   - `Executable`
3. Click **More Settings**.
4. On the **Custom Code** tab, under **Custom C-code to include in generated files**, specify **Source file** and **Header file**. **Source file** specifies that the code appear at the top of generated C/C++ source files. **Header file** specifies that the code appear at the top of generated header files.

Custom Code Property	Description
Under **Additional files and directories to be built**, provide an absolute path or a path relative to the project folder.	
Include directories	Specifies a list of folders that contain custom header, source, object, or library files. Separate list items with a semicolon.
Source files	Specifies additional custom C/C++ files to be compiled with the MATLAB file. Separate list items with a semicolon.
Libraries	Specifies the names of object or library files to be linked with the generated code. Separate list items with a semicolon.
Under **Custom C-code to include in generated files**	
Source file	Specifies code to appear at the top of generated C/C++ source files.
Header file	Specifies custom code to appear at the top of generated header files

Specify External Files at the Command Line

When you compile MATLAB function with MATLAB Coder, you can specify custom C/C++ files — such as source, header, and library files — on the command line along with your MATLAB file. For example, suppose you want to generate an embeddable C code executable that integrates a custom C function `myCfcn` with a MATLAB function `myMfcn` that has no input parameters. The custom source and header files for `myCfcn` reside in the folder `C:\custom`. You can use the following command to generate the code:

```
codegen C:\custom\myCfcn.c C:\custom\myCfcn.h myMfcn
```

Specify External Files with Configuration Objects

You can specify custom C/C++ files by setting custom code properties on configuration objects.

1. Define a configuration object, as described in “Creating Configuration Objects” on page 20-35.

   For example:

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

2. Set one or more of the custom code properties.

Custom Code Property	Description
`CustomInclude`	Specifies a list of folders that contain custom header, source, object, or library files. **Note:** If your folder path name contains spaces, you must enclose it in double quotes: `cc.CustomInclude = '"C:\Program Files\MATLAB\work"'`
`CustomSource`	Specifies additional custom C/C++ files to be compiled with the MATLAB file.
`CustomLibrary`	Specifies the names of object or library files to be linked with the generated code.
`CustomSourceCode`	Specifies code to insert at the top of each generated C/C++ source file.
`CustomHeaderCode`	Specifies custom code to insert at the top of each generated C/C++ header file.

   For example:

```
cc.CustomInclude = 'C:\custom\src C:\custom\lib';
cc.CustomSource = 'cfunction.c';
cc.CustomLibrary = 'chelper.obj clibrary.lib';
cc.CustomSourceCode = '#include "cgfunction.h"';
```

3 Compile the MATLAB code specifying the code generation configuration object.

Note: If you generate code for a function that has input parameters, you must specify the inputs. “Specify Properties of Entry-Point Function Inputs” on page 20-46

```
codegen -config cc  myFunc
```

4 Call custom C/C++ functions.

From...	Call...
C/C++ source code	Custom C/C++ functions directly
MATLAB code, compiled on the MATLAB Coder path	Custom C/C++ functions using `coder.ceval`.

For example, from MATLAB code:

```
...
y = 2.5;
y = coder.ceval('myFunc',y);
...
```

Code Generation of Matrices and Arrays

MATLAB and MATLAB Coder software store matrix data and arrays (1-D, 2-D, ...) in column-major format as a vector. Column-major format orders elements in a matrix starting from the first column, top to bottom, and then moving to the next column. For example, in the following 3x3 matrix:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

translates to an array of length 9 in the following order:

```
A(1) = A(1,1) = 1;
A(2) = A(2,1) = 4;
A(3) = A(3,1) = 7;
A(4) = A(1,2) = 2;
A(5) = A(2,2) = 5;
```

and so on.

In column-major format, the software accesses the next element of an array in memory by incrementing the first index of the array. For example, the software stores these element pairs sequentially in memory:

- `A(i)` and `A(i+1)`
- `B(i,j)` and `B(i+1,j)`
- `C(i,j,k)` and `C(i+1,j,k)`

For more information on the internal representation of MATLAB data, see “MATLAB Data” (MATLAB).

The code generator uses column-major format because:

- Much of the software that supports signal and array processing uses column-major format: MATLAB, LAPACK, Fortran90, DSP libraries.
- A column is equivalent to a channel in frame-based processing. In this case, column-major storage is more efficient than row-major storage.
- A column-major array is self-consistent with its component submatrices:
  - A column-major 2-D array is a simple concatenation of 1-D arrays.
  - A column-major 3-D array is a simple concatenation of 2-D arrays.

- The stride is the number of memory locations to index to the next element in the same dimension. The stride of the first dimension is one element. The stride of the nth dimension element is the product of sizes of the lower dimensions.
- Row-major n-D arrays have their stride of 1 for the highest dimension. Submatrix manipulations typically access a scattered data set in memory, which does not allow for efficient indexing.

C typically uses row-major format. MATLAB uses column-major format. You cannot configure the code generator to produce code with row-major ordering. If you are integrating legacy C code with the generated code, consider transposing the row-major data in your legacy C code into column-major format as a 1-D array.

More About

- “MATLAB Data” (MATLAB)
- “How MATLAB Coder Infers C/C++ Data Types” on page 26-9

Incorporate Generated Code Using an Example Main Function

In this section...

When you build an application that uses generated C/C++ code, you must provide a C/C++ main function that calls the generated code.

By default, for code generation of C/C++ source code, static libraries, dynamic libraries, and executables, MATLAB Coder generates an example C/C++ main function. This function is a template that can help you incorporate generated C/C++ code into your application. The example main function declares and initializes data, including dynamically allocated data. It calls entry-point functions but does not use values that the entry point functions return.

MATLAB Coder generates source and header files for the example main function in the `examples` subfolder of the build folder. For C code generation, it generates the files `main.c` and `main.h`. For C++ code generation, it generates the files `main.cpp` and `main.h`.

Do not modify the files `main.c` and `main.h` in the `examples` subfolder. If you do, when you regenerate code, MATLAB Coder does not regenerate the example main files. It warns you that it detects changes to the generated files. Before using the example main function, copy the example main source and header files to a location outside of the build folder. Modify the files in the new location to meet the requirements of your application.

The `packNGo` function and the **Package** option of the MATLAB Coder app do not package the example main source and header files when you generate the files using the default configuration settings. To package the example main files, configure code generation to generate and compile the example main function, generate your code, and then package the build files.

Workflow for Using an Example Main Function

1. Prepare your MATLAB code for code generation.
2. Check for run-time issues.

3. Make sure that example main generation is enabled.
4. Generate C/C++ code for the entry-point functions.
5. Copy the example main files from the `examples` subfolder to a different folder.
6. Modify the example main files in the new folder to meet the requirements of your application.
7. Deploy the example main and generated code for the platform that you want.
8. Build the application.

For an example that shows how to generate an example main and use it to build an executable, see “Use an Example C Main in an Application” on page 24-23.

Control Example Main Generation Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow ▼.
2. In the **Generate** dialog box, set the **Build Type** to one of the following:
   - `Source Code`
   - `Static Library`
   - `Dynamic Library`
   - `Executable`
3. Click **More Settings**.
4. On the **All Settings** tab, under **Advanced**, set **Generate example main** to one of the following:

Set To	For
`Do not generate an example main function`	Not generating an example C/C++ main function
`Generate, but do not compile, an example main function` (default)	Generating an example C/C++ main function but not compiling it
`Generate and compile an example main function`	Generating an example C/C++ main function and compiling it

Control Example Main Generation Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

```
cfg = coder.config('lib'); % or dll or exe
```

2 Set the `GenerateExampleMain` property.

Set To	For
`'DoNotGenerate'`	Not generating an example C/C++ main function
`'GenerateCodeOnly'` (default)	Generating an example C/C++ main function but not compiling it
`'GenerateCodeAndCompile'`	Generating an example C/C++ main function and compiling it

For example:

```
cfg.GenerateExampleMain = 'GenerateCodeOnly';
```

Related Examples

- "Structure of Generated Example C/C++ Main Function" on page 24-51
- "Call a Generated C Static Library Function from C Code" on page 24-4

More About

- "Specifying main Functions for C/C++ Executables" on page 20-24

Use an Example C Main in an Application

This example shows how to build a C executable from MATLAB code that implements a simple Sobel filter to perform edge detection on images. The executable reads an image from the disk, applies the Sobel filtering algorithm, and then saves the modified image.

The example shows how to generate and modify an example main function that you can use when you build the executable.

In this section...

Prerequisites

To complete this example, install the following products:

- MATLAB
- MATLAB Coder
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). For a list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/`.

  You can use `mex -setup` to change the default compiler. See "Change Default Compiler" (MATLAB).

Create a Folder and Copy Relevant Files

The files you use in this example are:

File Name	File Type	Description
sobel.m	Function code	MATLAB implementation of a Sobel filtering algorithm. sobel.m takes an image (represented as a double matrix) and a threshold value as inputs. The algorithm detects edges in the image (based on the threshold value). sobel.m returns a modified image displaying the edges.
hello.jpg	Image file	Image that the Sobel filter modifies.

Contents of File sobel.m

```
function edgeImage = sobel(originalImage, threshold) %#codegen

% edgeImage = sobel(originalImage, threshold)
% Sobel edge detection. Given a normalized image (with double values)
% return an image where the edges are detected w.r.t. threshold value.

assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = [1 2 1; 0 0 0; -1 -2 -1];
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Contents of hello.jpg

To copy the example files to a local working folder:

1. Create a local working folder. For example, `c:\coder\edge_detection`.
2. Navigate to the working folder.
3. Copy the files `sobel.m` and `hello.jpg` from the examples folder `sobel` to your working folder.

```
copyfile(fullfile(docroot, 'toolbox', 'coder', 'examples', 'sobel'))
```

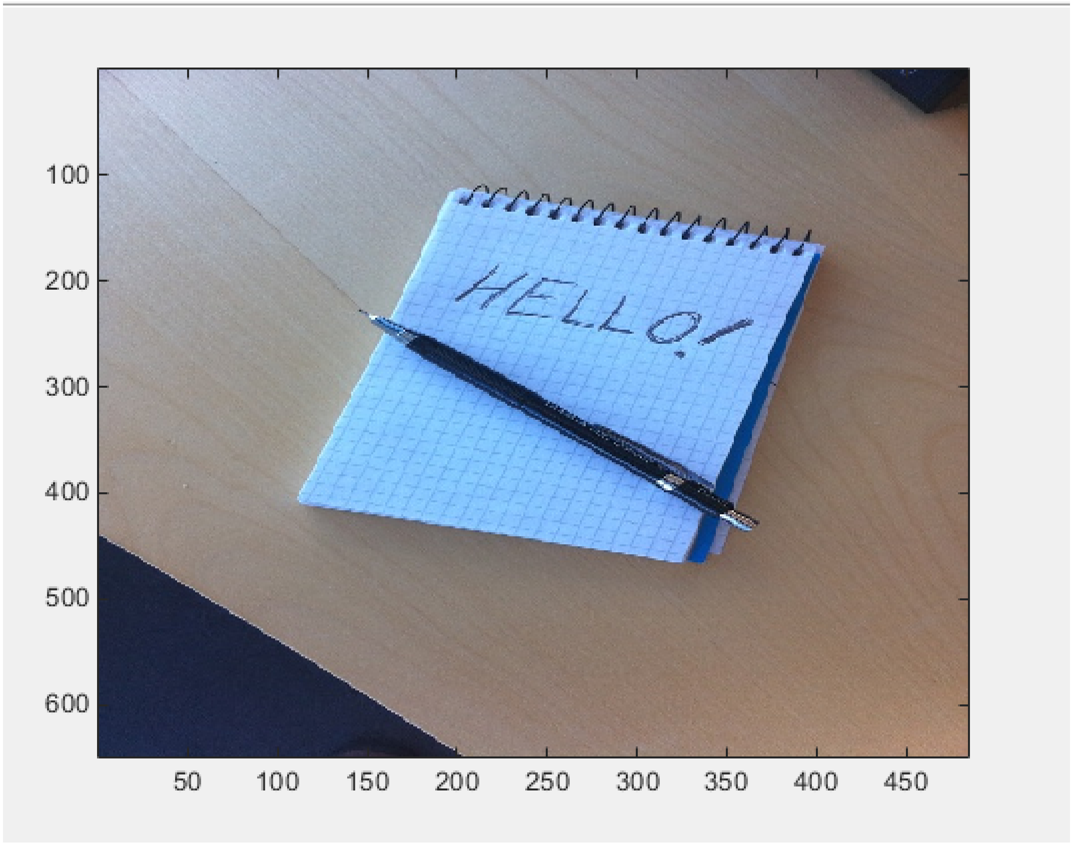
Run the Sobel Filter on the Image

1 Read the original image into a MATLAB matrix and display it.

```
im = imread('hello.jpg');
```

2 Display the image as a basis for comparison to the result of the Sobel filter.

```
image(im);
```



3 The Sobel filtering algorithm operates on grayscale images. Convert the color image to an equivalent grayscale image with normalized values (0.0 for black, 1.0 for white).

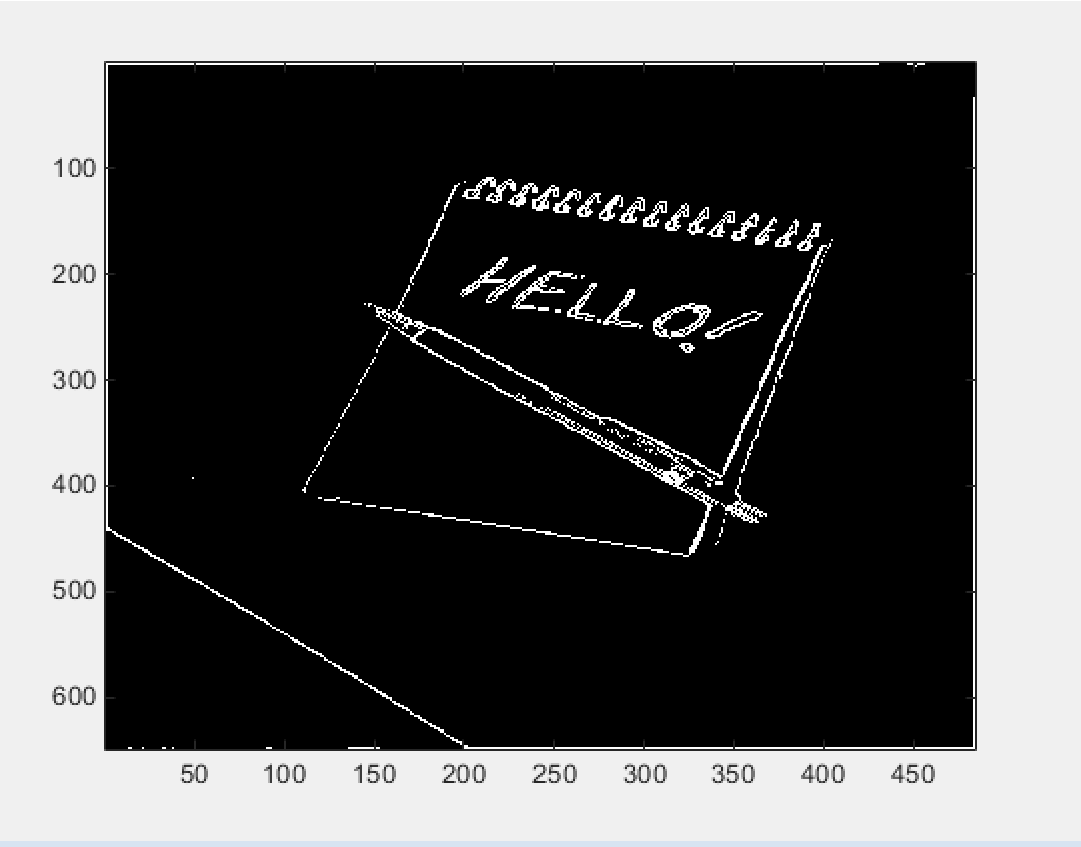
```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
```

4 To run the MATLAB function for the Sobel filter, pass the grayscale image matrix `gray` and a threshold value to the function `sobel`. This example uses 0.7 for a threshold value.

```
edgeIm = sobel(gray, 0.7);
```

5 To display the modified image, reformat the matrix `edgeIm` with the function `repmat` so that you can pass it to the `image` command.

```
im3 = repmat(edgeIm, [1 1 3]);
image(im3);
```



Generate and Test a MEX Function

1. To test that generated code is functionally equivalent to the original MATLAB code and that run-time errors do not occur, generate a MEX function.

   ```
   codegen -report sobel
   ```

   `codegen` generates a MEX function named `sobel_mex` in the current working folder.
2. To run the MEX function for the Sobel filter, pass the grayscale image matrix `gray` and a threshold value to the function `sobel_mex`. This example uses 0.7 for a threshold value.

   ```
   edgeImMex = sobel_mex(gray, 0.7);
   ```
3. To display the modified image, reformat the matrix `edgeImMex` with the function `repmat` so that you can pass it to the `image` command.

   ```
   im3Mex = repmat(edgeImMex, [1 1 3]);
   image(im3Mex);
   ```

   This image is the same as the image created using the MATLAB function.

Generate an Example Main Function for sobel.m

Although you can write a custom main function for your application, an example main function provides a template to help you incorporate the generated code.

To generate an example main function for the Sobel filter:

1. Create a configuration object for a C static library.

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

   For configuration objects for C/C++ source code, static libraries, dynamic libraries, and executables, the setting `GenerateExampleMain` controls generation of the example main function. The setting is set to `'GenerateCodeOnly'` by default, which generates the example main function but does not compile it. For this example, do not change the value of the `GenerateExampleMain` setting.
2. Generate a C static library using the configuration object.

   ```
   codegen -report -config cfg sobel
   ```

The generated files for the static library are in the folder `codegen/lib/sobel`. The example main files are in the subfolder `codegen/lib/sobel/examples`.

Contents of Example Main File main.c

```
/*
 * main.c
 *
 * Code generation for function 'main'
 *
 */

/*************************************************************************/
/* This automatically generated example C main file shows how to call    */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly.     */
/* Instead, make a copy of this file, modify it, and integrate it into   */
/* your development environment.                                         */
/*                                                                       */
/* This file initializes entry-point function arguments to a default     */
/* size and value before calling the entry-point functions. It does      */
/* not store or use any values returned from the entry-point functions.  */
/* If necessary, it does pre-allocate memory for returned values.        */
/* You can use this file as a starting point for a main function that    */
/* you can deploy in your application.                                   */
/*                                                                       */
/* After you copy the file, and before you deploy it, you must make the  */
/* following changes:                                                    */
/* * For variable-size function arguments, change the example sizes to   */
/* the sizes that your application requires.                             */
/* * Change the example values of function arguments to the values that  */
/* your application requires.                                            */
/* * If the entry-point functions return values, store these values or   */
/* otherwise use them as required by your application.                   */
/*                                                                       */
/*************************************************************************/
/* Include files */
#include "rt_nonfinite.h"
#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"
```

```
/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(void);
static double argInit_real_T(void);
static void main_sobel(void);

/* Function Definitions */
static emxArray_real_T *argInit_d1024xd1024_real_T(void)
{
  emxArray_real_T *result;
  static int iv2[2] = { 2, 2 };

  int b_j0;
  int b_j1;

  /* Set the size of the array.
     Change this size to the value that the application requires. */
  result = emxCreateND_real_T(2, iv2);

  /* Loop over the array to initialize each element. */
  for (b_j0 = 0; b_j0 < result->size[0U]; b_j0++) {
    for (b_j1 = 0; b_j1 < result->size[1U]; b_j1++) {
      /* Set the value of the array element.
         Change this value to the value that the application requires. */
      result->data[b_j0 + result->size[0] * b_j1] = argInit_real_T();
    }
  }

  return result;
}

static double argInit_real_T(void)
{
  return 0.0;
}

static void main_sobel(void)
{
  emxArray_uint8_T *edgeImage;
  emxArray_real_T *originalImage;
  emxInitArray_uint8_T(&edgeImage, 2);

  /* Initialize function 'sobel' input arguments. */
  /* Initialize function input argument 'originalImage'. */
  originalImage = argInit_d1024xd1024_real_T();
  /* Call the entry-point 'sobel'. */
  sobel(originalImage, argInit_real_T(), edgeImage);
  emxDestroyArray_uint8_T(edgeImage);
  emxDestroyArray_real_T(originalImage);
}

int main(int argc, const char * const argv[])
{
  (void)argc;
  (void)argv;

  /* Initialize the application.
     You do not need to do this more than one time. */
  sobel_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_sobel();

  /* Terminate the application.
     You do not need to do this more than one time. */
  sobel_terminate();
  return 0;
}

/* End of code generation (main.c) */
```

Copy the Example Main Files

Do not modify the files `main.c` and `main.h` in the `examples` subfolder. If you do, when you regenerate code, MATLAB Coder does not regenerate the example main files. It warns you that it detects changes to the generated files.

Copy the files `main.c` and `main.h` from the folder `codegen/lib/sobel/examples` to another location. For this example, copy the files to the current working folder. Modify the files in the new location.

Modify the Generated Example Main Function

- "Modify the Function main" on page 24-32
- "Modify the Initialization Function argInit_d1024xd1024_real_T" on page 24-34

- “Write the Function saveImage” on page 24-36
- “Modify the Function main_sobel” on page 24-38
- “Modify the Function Declarations” on page 24-39
- “Modify the Include Files” on page 24-39
- “Contents of Modified File main.c” on page 24-40

The example main function declares and initializes data, including dynamically allocated data, to zero values. It calls entry-point functions with arguments set to zero values, but it does not use values returned from the entry-point functions.

The C main function must meet the requirements of your application. This example modifies the example main function to meet the requirements of the Sobel filter application.

This example modifies the file main.c so that the Sobel filter application:

- Reads in the grayscale image from a binary file.
- Applies the Sobel filtering algorithm.
- Saves the modified image to a binary file.

Modify the Function main

Modify the function main to:

- Accept the file containing the grayscale image data and a threshold value as input arguments.
- Call the function main_sobel with the address of the grayscale image data stream and the threshold value as input arguments.

In the function main:

1. Remove the declarations void(argc) and (void)argv.
2. Declare the variable filename to hold the name of the binary file containing the grayscale image data.

   ```
   const char *filename;
   ```
3. Declare the variable threshold to hold the threshold value.

   ```
   double threshold;
   ```

4 Declare the variable fd to hold the address of the grayscale image data that the application reads in from filename.

```
FILE *fd;
```

5 Add an if statement that checks for three arguments.

```
if (argc != 3) {
      printf("Expected 2 arguments: filename and threshold\n");
      exit(-1);
}
```

6 Assign the input argument argv[1] for the file containing the grayscale image data to filename.

```
filename = argv[1];
```

7 Assign the input argument argv[2] for the threshold value to threshold, converting the input from a string to a numeric double.

```
threshold = atof(argv[2]);
```

8 Open the file containing the grayscale image data whose name is specified in filename. Assign the address of the data stream to fd.

```
fd = fopen(filename, "rb");
```

9 To verify that the executable can open filename, write an if-statement that exits the program if the value of fd is NULL.

```
if (fd == NULL) {
   exit(-1);
}
```

10 Replace the function call for main_sobel by calling main_sobel with input arguments fd and threshold.

```
main_sobel(fd, threshold);
```

11 Close the grayscale image file after calling sobel_terminate.

```
fclose(fd);
```

Modified Function main

```
int main(int argc, const char * const argv[])
{
  const char *filename;
  double threshold;
  FILE *fd;

  if (argc != 3) {
      printf("Expected 2 arguments: filename and threshold\n");
      exit(-1);
  }

  filename = argv[1];
  threshold = atof(argv[2]);
  fd = fopen(filename, "rb");
  if (fd == NULL) {
    exit(-1);
  }
  /* Initialize the application.
     You do not need to do this more than one time. */
  sobel_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_sobel(fd, threshold);

  /* Terminate the application.
     You do not need to do this more than one time. */
  sobel_terminate();

  fclose(fd);

  return 0;
}
```

Modify the Initialization Function argInit_d1024xd1024_real_T

In the example main file, the function `argInit_d1024xd1024_real_T` creates a dynamically allocated variable-size array (emxArray) for the image that you pass to the Sobel filter. This function initializes the emxArray to a default size and the elements of the emxArray to 0. It returns the initialized emxArray.

For the Sobel filter application, modify the function to read the grayscale image data from a binary file into the emxArray.

In the function `argInit_d1024xd1024_real_T`:

1 Replace the input argument `void` with the argument `FILE *fd`. This variable points to the grayscale image data that the function reads in.

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
```

2 Change the values of the variable iv2 to match the dimensions of the grayscale image matrix gray. iv2 holds the size values for the dimensions of the emxArray that argInit_d1024xd1024_real_T creates.

```
static int iv2[2] = { 484, 648 };
```

MATLAB stores matrix data in column-major format, while C stores matrix data in row-major format. Declare the dimensions accordingly.

3 Define a variable element to hold the values read in from the grayscale image data.

```
double element;
```

4 Change the for-loop construct to read data points from the normalized image into element by adding an fread command to the inner for-loop.

```
fread(&element, 1, sizeof(element), fd);
```

5 Inside the for-loop, assign element as the value set for the emxArray data.

```
result->data[b_j0 + result->size[0] * b_j1] = element;
```

Modified Initialization Function argInit_d1024xd1024_real_T

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
  emxArray_real_T *result;
  static int iv2[2] = { 484, 648 };

  int b_j0;
  int b_j1;
  double element;

  /* Set the size of the array.
     Change this size to the value that the application requires. */
  result = emxCreateND_real_T(2, iv2);

  /* Loop over the array to initialize each element. */
  for (b_j0 = 0; b_j0 < result->size[0U]; b_j0++) {
    for (b_j1 = 0; b_j1 < result->size[1U]; b_j1++) {
      /* Set the value of the array element.
         Change this value to the value that the application requires. */
      fread(&element, 1, sizeof(element), fd);
      result->data[b_j0 + result->size[0] * b_j1] = element;
    }
  }

  return result;
}
```

Write the Function saveImage

The MATLAB function `sobel.m` interfaces with MATLAB arrays, but the Sobel filter application interfaces with binary files.

To save the image modified by the Sobel filtering algorithm to a binary file, create a function `saveImage`. The function `saveImage` writes data from an emxArray into a binary file. It uses a construction that is similar to the one used by the function `argInit_d1024xd1024_real_T`.

In the file `main.c`:

1. Define the function `saveImage` that takes the address of emxArray `edgeImage` as an input and has output type void.

   ```
   static void saveImage(emxArray_uint8_T *edgeImage)
   {
   }
   ```
2. Define the variables `b_j0` and `b_j1` like they are defined in the function `argInit_d1024xd1024_real_T`.

   ```
   int b_j0;
   int b_j1;
   ```
3. Define the variable `element` to store data read from the emxArray.

   ```
   uint8_T element;
   ```
4. Open a binary file `edge.bin` for writing the modified image. Assign the address of `edge.bin` to `FILE *fd`.

   ```
   FILE *fd = fopen("edge.bin", "wb");
   ```
5. To verify that the executable can open `edge.bin`, write an `if`-statement that exits the program if the value of `fd` is `NULL`.

   ```
   if (fd == NULL) {
      exit(-1);
   }
   ```

6. Write a nested `for`-loop construct like the one in the function `argInit_d1024xd1024_real_T`.

   ```
   for (b_j0 = 0; b_j0 < edgeImage->size[0U]; b_j0++)
   {
       for (b_j1 = 0; b_j1 < edgeImage->size[1U]; b_j1++)
       {
       }
   }
   ```

7. Inside the inner `for`-loop, assign the values from the modified image data to `element`.

   ```
   element = edgeImage->data[b_j0 + edgeImage->size[0] * b_j1];
   ```

8. After the assignment for `element`, write the value from `element` to the file `edge.bin`.

   ```
   fwrite(&element, 1, sizeof(element), fd);
   ```

9. After the `for`-loop construct, close `fd`.

   ```
   fclose(fd);
   ```

Function saveImage

```
static void saveImage(emxArray_uint8_T *edgeImage)
{
  int b_j0;
  int b_j1;
  uint8_T element;

  FILE *fd = fopen("edge.bin", "wb");
 if (fd == NULL) {
   exit(-1);
 }
  /* Loop over the array to save each element. */
  for (b_j0 = 0; b_j0 < edgeImage->size[0U]; b_j0++) {
    for (b_j1 = 0; b_j1 < edgeImage->size[1U]; b_j1++) {
      element = edgeImage->data[b_j0 + edgeImage->size[0] * b_j1];
      fwrite(&element, 1, sizeof(element), fd);
    }
  }
  fclose(fd);
}
```

Modify the Function main_sobel

In the example main function, the function main_sobel creates emxArrays for the data for the grayscale and modified images. It calls the function argInit_d1024xd1024_real_T to initialize the emxArray for the grayscale image. main_sobel passes both emxArrays and the threshold value of 0 that the initialization function argInit_real_T returns to the function sobel. When the function main_sobel ends, it discards the result of the function sobel.

For the Sobel filter application, modify the function main_sobel to:

- Take the address of the grayscale image data and the threshold value as inputs.
- Read the data from the address using argInit_d1024xd1024_real_T.
- Pass the data to the Sobel filtering algorithm with the threshold value threshold.
- Save the result using saveImage.

In the function main_sobel:

1. Replace the input arguments to the function with the arguments FILE *fd and double threshold.

   ```
   static void main_sobel(FILE *fd, double threshold)
   ```
2. Pass the input argument fd to the function call for argInit_d1024xd1024_real_T.

   ```
   originalImage = argInit_d1024xd1024_real_T(fd);
   ```
3. Replace the threshold value input in the function call to sobel with threshold.

   ```
   sobel(originalImage, threshold, edgeImage);
   ```
4. After calling the function sobel, call the function saveImage with the input edgeImage.

   ```
   saveImage(edgeImage);
   ```

Modified Function main_sobel

```
static void main_sobel(FILE *fd, double threshold)
{
  emxArray_uint8_T *edgeImage;
  emxArray_real_T *originalImage;
  emxInitArray_uint8_T(&edgeImage, 2);
```

```
  /* Initialize function 'sobel' input arguments. */
  /* Initialize function input argument 'originalImage'. */
  originalImage = argInit_d1024xd1024_real_T(fd);

  /* Call the entry-point 'sobel'. */
  sobel(originalImage, threshold, edgeImage);

  saveImage(edgeImage);

  emxDestroyArray_uint8_T(edgeImage);
  emxDestroyArray_real_T(originalImage);
}
```

Modify the Function Declarations

To match the changes that you made to the function definitions, make the following changes to the function declarations:

1. Change the input of the function `*argInit_d1024xd1024_real_T` to `FILE *fd`.

   ```
   static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
   ```
2. Change the inputs of the function `main_sobel` to `FILE *fd` and `double threshold`.

   ```
   static void main_sobel(FILE *fd, double threshold);
   ```
3. Add the function `saveImage`.

   ```
   static void saveImage(emxArray_uint8_T *edgeImage);
   ```

Modified Function Declarations

```
/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
static void saveImage(emxArray_uint8_T *edgeImage);
static double argInit_real_T(void);
static void main_sobel(FILE *fd, double threshold);
```

Modify the Include Files

For input/output functions that you use in `main.c`, add the header file `stdio.h` to the included files list.

```
#include <stdio.h>
```

Modified Include Files

```
/* Include Files */
#include <stdio.h>

#include "rt_nonfinite.h"
#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"
```

Contents of Modified File `main.c`

`main.c`

```
/*
 * main.c
 *
 * Code generation for function 'main'
 *
 */

/*************************************************************************/
/* This automatically generated example C main file shows how to call    */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly.     */
/* Instead, make a copy of this file, modify it, and integrate it into   */
/* your development environment.                                         */
/*                                                                       */
/* This file initializes entry-point function arguments to a default     */
/* size and value before calling the entry-point functions. It does      */
/* not store or use any values returned from the entry-point functions.  */
/* If necessary, it does pre-allocate memory for returned values.        */
/* You can use this file as a starting point for a main function that    */
/* you can deploy in your application.                                   */
/*                                                                       */
/* After you copy the file, and before you deploy it, you must make the  */
/* following changes:                                                    */
/* * For variable-size function arguments, change the example sizes to   */
/* the sizes that your application requires.                             */
/* * Change the example values of function arguments to the values that  */
/* your application requires.                                            */
/* * If the entry-point functions return values, store these values or   */
/* otherwise use them as required by your application.                   */
/*                                                                        */
/**************************************************************************/
/* Include Files */
#include <stdio.h>

#include "rt_nonfinite.h"
#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"

/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
static void saveImage(emxArray_uint8_T *edgeImage);
static double argInit_real_T(void);
static void main_sobel(FILE *fd, double threshold);

/* Function Definitions */

static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
  emxArray_real_T *result;
  static int iv2[2] = { 484, 648 };

  int b_j0;
  int b_j1;
  double element;

  /* Set the size of the array.
     Change this size to the value that the application requires. */
  result = emxCreateND_real_T(2, iv2);

  /* Loop over the array to initialize each element. */
  for (b_j0 = 0; b_j0 < result->size[0U]; b_j0++) {
    for (b_j1 = 0; b_j1 < result->size[1U]; b_j1++) {
      /* Set the value of the array element.
         Change this value to the value that the application requires. */
      fread(&element, 1, sizeof(element), fd);
      result->data[b_j0 + result->size[0] * b_j1] = element;
    }
  }

  return result;
```

```
}

static void saveImage(emxArray_uint8_T *edgeImage)
{
  int b_j0;
  int b_j1;
  uint8_T element;

  FILE *fd = fopen("edge.bin", "wb");
 if (fd == NULL) {
   exit(-1);
 }
  /* Loop over the array to save each element. */
  for (b_j0 = 0; b_j0 < edgeImage->size[0U]; b_j0++) {
    for (b_j1 = 0; b_j1 < edgeImage->size[1U]; b_j1++) {
      element = edgeImage->data[b_j0 + edgeImage->size[0] * b_j1];
      fwrite(&element, 1, sizeof(element), fd);
    }
  }
  fclose(fd);
}

/*
 * Arguments    : void
 * Return Type  : double
 */
static double argInit_real_T(void)
{
  return 0.0;
}

static void main_sobel(FILE *fd, double threshold)
{
  emxArray_uint8_T *edgeImage;
  emxArray_real_T *originalImage;
  emxInitArray_uint8_T(&edgeImage, 2);

  /* Initialize function 'sobel' input arguments. */
  /* Initialize function input argument 'originalImage'. */
  originalImage = argInit_d1024xd1024_real_T(fd);

  /* Call the entry-point 'sobel'. */
  sobel(originalImage, threshold, edgeImage);
  saveImage(edgeImage);

  emxDestroyArray_uint8_T(edgeImage);
  emxDestroyArray_real_T(originalImage);
}

int main(int argc, const char * const argv[])
{
  const char *filename;
  double threshold;
  FILE *fd;

  if (argc != 3) {
      printf("Expected 2 arguments: filename and threshold\n");
      exit(-1);
  }

  filename = argv[1];
  threshold = atof(argv[2]);
  fd = fopen(filename, "rb");
  if (fd == NULL) {
    exit(-1);
  }
  /* Initialize the application.
     You do not need to do this more than one time. */
  sobel_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_sobel(fd, threshold);

  /* Terminate the application.
     You do not need to do this more than one time. */
  sobel_terminate();

  fclose(fd);

  return 0;
}

/* End of code generation (main.c) */
```

Generate the Sobel Filter Application

1. Navigate to the working folder if you are not currently in it.
2. Create a configuration object for a C standalone executable.

   ```
   cfg = coder.config('exe');
   ```
3. Generate a C standalone executable for the Sobel filter using the configuration object and the modified main function.

   ```
   codegen -report -config cfg sobel main.c main.h
   ```

By default, if you are running MATLAB on a Windows platform, the executable `sobel.exe` is generated in the current working folder. If you are running MATLAB on a platform other than Windows, the file extension is the corresponding extension for that platform. By default, the code generated for the executable is in the folder `codegen/exe/sobel`.

Run the Sobel Filter Application

1. Create the MATLAB matrix `gray` if it is not currently in your MATLAB workspace:

   ```
   im = imread('hello.jpg');
   gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
   ```
2. Write the matrix `gray` into a binary file using the `fopen` and `fwrite` commands. The application reads in this binary file.

   ```
   fid = fopen('gray.bin', 'w');
   fwrite(fid, gray, 'double');
   fclose(fid);
   ```
3. Run the executable, passing to it the file `gray.bin` and the threshold value 0.7.

   To run the example in MATLAB on a Windows platform:

   ```
   system('sobel.exe gray.bin 0.7');
   ```

   The executable generates the file `edge.bin`.

Display the Resulting Image

1. Read the file `edge.bin` into a MATLAB matrix `edgeImExe` using the `fopen` and `fread` commands.

```
fd = fopen('edge.bin', 'r');
edgeImExe = fread(fd, size(gray), 'uint8');
fclose(fd);
```

2 Pass the matrix `edgeImExe` to the function `repmat` and display the image.

```
im3Exe = repmat(edgeImExe, [1 1 3]);
image(im3Exe);
```

The image matches the images from the MATLAB and MEX functions.

Related Examples

- “Structure of Generated Example C/C++ Main Function” on page 24-51
- “Incorporate Generated Code Using an Example Main Function” on page 24-20
- “Call a Generated C Static Library Function from C Code” on page 24-4

Package Code for Other Development Environments

In this section...

When to Package Code

To relocate the generated code files to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB, use the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

See "Package Generated Code Using the MATLAB Coder App" on page 24-46 and "Package Generated Code at the Command Line" on page 24-48.

Package Generated Code Using the MATLAB Coder App

This example shows how to package generated code into a zip file for relocation using the **Package** option in the MATLAB Coder app. By default, MATLAB Coder creates the zip file in the current working folder.

1 In a local writable folder, for example `c:\work`, write a function `foo` that takes two double inputs.

```
function y = foo(A,B)
  y = A + B;
end
```

2 Open the MATLAB Coder app. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

3 On the **Select Source Files** page, enter the name of the entry-point function `foo`. Click **Next** to go to the **Define Input Types** page.

4 Specify that inputs `A` and `B` are scalar doubles. Click **Next** to go to the **Check for Run-Time Issues** page.

5. Check for run-time issues. In the **Check for Run-Time Issues** dialog box, enter code that calls `foo` with scalar double inputs. For example:

   ```
   foo(1,2)
   ```
   Click **Check for Issues**.

   To check for run-time issues, the app generates and runs a MEX function. The app does not find issues for `foo`. Click **Next** to go to the **Generate Code** page.
6. In the **Generate** dialog box, set the **Build Type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable`. You cannot package the code generated for MEX targets.
7. Click **Generate**. Click **Next** to go to the **Finish Workflow** page.
8. On the **Finish Workflow** page, click **Package**.
9. In the **Package** dialog box, specify the package file name and packaging type. By default, the app derives the name of the package file from the project name. The app saves the file in the current working folder. By default, the app packages the generated files as a single, flat folder. For this example, use the default values, and then click **Save**.

   This zip file contains the C code and header files required for relocation. It does not contain:

   - Compile flags
   - Defines
   - Makefiles
   - Example main files, unless you configure code generation to generate and compile the example main function. See “Incorporate Generated Code Using an Example Main Function” on page 24-20.
10. Inspect the contents of `foo_pkg.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially open and inspect the file without unpacking it.

    You can now relocate the resulting zip file to the desired development environment and unpack the file.

Package Generated Code at the Command Line

This example shows how to package generated code into a zip file for relocation using the packNGo function at the command line.

1. In a local writable folder, for example c:\work, write a function foo that takes two double inputs.

   ```
   function y = foo(A,B)
     y = A + B;
   end
   ```

2. Generate a static library for function foo. (packNGo does not package MEX function code.)

   ```
   codegen -report -config:lib foo -args {0,0}
   ```

   codegen generates code in the c:\work\codegen\lib\foo folder.

3. Load the buildInfo object.

   ```
   load('c:\work\codegen\lib\foo\buildInfo.mat')
   ```

4. Create the zip file.

   ```
   packNGo(buildInfo, 'fileName', 'foo.zip');
   ```

   Alternatively, use the notation:

   ```
   buildInfo.packNGo('fileName', 'foo.zip');
   ```

   The packNGo function creates a zip file, foo.zip, in the current working folder. This zip file contains the C code and header files required for relocation. It does not contain:

   - Compile flags
   - Defines
   - Makefiles
   - Example main files, unless you configure code generation to generate and compile the example main function. See “Incorporate Generated Code Using an Example Main Function” on page 24-20.

   In this example, you specify only the file name. Optionally, you can specify additional packaging options. See “Specify packNGo Options” on page 24-49.

5 Inspect the contents of `foo.zip` to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially open and inspect the file without unpacking it. If you need to unpack the file and you packaged the generated code files as a hierarchical structure, you will need to unpack the primary and secondary zip files. When you unpack the secondary zip files, relative paths of the files are preserved.

   You can now relocate the resulting zip file to the desired development environment and unpack the file.

Specify packNGo Options

You can specify options for the `packNGo` function.

To	Specify
Change the structure of the file packaging to hierarchical	`packNGo(buildInfo, {'packType' 'hierarchical'});`

Change the structure of the file packaging to hierarchical and rename the primary zip file	`packNGo(buildInfo, {'packType' 'hierarchical'...`
	`'fileName' 'zippedsrcs'});`

Include all header files found on the include path in the zip file (rather than the minimal header files required to build the code)	`packNGo(buildInfo, {'minimalHeaders' false});`

Generate warnings for parse errors and missing files	`packNGo(buildInfo, {'ignoreParseError' true...`
	`'ignoreFileMissing' true});`

For more information, see `packNGo` in “Build Information Methods” on page 20-141.

Choose a Structure for the Zip File

Before you generate and package the files, decide whether you want to package the files in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure. This approach is the simplest and might be the optimal choice.

If	Use
You are relocating files to an IDE that does not use the generated makefile, or the code	A single, flat folder structure

If	Use
is not dependent on the relative location of required static files	
The target development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code is dependent on the relative location of files	A hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of zip files. There is a primary zip file, which in turn contains the following secondary zip files:

- `mlrFiles.zip` — files in your *`matlabroot`* folder tree
- `sDirFiles.zip` — files in and under your build folder where you initiated code generation
- `otherFiles.zip` — required files not in the *`matlabroot`* or `start` folder trees

Paths for the secondary zip files are relative to the root folder of the primary zip file, maintaining the source development folder structure.

Structure of Generated Example C/C++ Main Function

In this section...

When you build an application that uses generated C/C++ code, you must provide a C/C++ main function that calls the generated code.

By default, for code generation of C/C++ source code, static libraries, dynamic libraries, and executables, MATLAB Coder generates an example C/C++ main function. This function is a template that can help you incorporate generated C/C++ code into your application. The example main function declares and initializes data, including dynamically allocated data. It calls entry-point functions but does not use values that the entry point functions return. To use the example main function, copy the example main source and header files to a location outside of the build folder, and then modify the files in the new location to meet the requirements of your application.

MATLAB Coder generates source and header files for the example main function in the `examples` subfolder of the build folder. For C code generation, it generates the files `main.c` and `main.h`. For C++ code generation, it generates the files `main.cpp` and `main.h`.

Contents of the File `main.c` or `main.cpp`

For the example main source file `main.c` or `main.cpp`, MATLAB Coder generates the following sections:

- "Include Files" on page 24-52
- "Function Declarations" on page 24-52
- "Argument Initialization Functions" on page 24-52
- "Entry-Point Functions" on page 24-53
- "Main Function" on page 24-53

By default, MATLAB Coder also generates comments in the example main source file that can help you modify the example main function to use in your application.

Include Files

This section includes the header files required to call code that is not in the example main source file. If you call external functions when you modify the example main source file, include any other required header files.

Function Declarations

This section declares the function prototypes for the argument initialization and entry-point functions that are defined in the example main source file. Modify the function prototypes to match modifications that you make in the function definitions. Declare new function prototypes for functions that you define in the example main source file.

Argument Initialization Functions

This section defines an initialization function for each data type that the entry-point functions use as an argument. The argument initialization function initializes the size of the argument to a default value and the values of the data to zero. The function then returns the initialized data. Change these size and data values to meet the requirements of your application.

For an argument with dimensions of size `<dimSizes>` and MATLAB C/C++ data type `<baseType>`, the example main source file defines an initialization function with the name `argInit_<dimSizes>_<baseType>`. For example, for a 5-by-5 array with data of MATLAB type double, the example main source file defines the argument initialization function `argInit_5x5_real_T`.

MATLAB Coder alters the name of the argument initialization functions as follows:

- If any of the dimensions are variable-size, MATLAB Coder designates the size of these dimensions as `d<maxSize>`, where `<maxSize>` is the maximum size of that dimension. For example, for an array with data of MATLAB type double with a first dimension of static size 2 and a second dimension that can vary in size up to 10, the example main source file defines the argument initialization function `argInit_2xd10_real_T`.
- If any of the dimensions are unbounded, MATLAB Coder designates the size of these dimensions as `Unbounded`.
- If the return type of the initialization function is an `emxArray`, MATLAB Coder defines the function as returning a pointer to the `emxArray`.
- If the length of the initialization function name exceeds the maximum number of characters set for function names in the configuration settings, MATLAB Coder

prepends an identifier to the front of the function name. MATLAB Coder then truncates the function name to the maximum allowed number of characters for identifier length.

Note: By default, the maximum number of characters allowed for generated identifiers is 31. To specify the value set for the maximum identifier length using the MATLAB Coder app, select the **Maximum identifier length** value on the **Code Appearance** tab of the code generation settings. To specify the value set for the maximum identifier using the command-line interface, change the value of the `MaxIdLength` configuration object setting.

Entry-Point Functions

This section defines a function for each MATLAB entry-point function. For a MATLAB function `foo.m`, the example main source file defines an entry-point function `main_foo`. This function creates the variables and calls the data initialization functions that the C/C++ source function `foo.c` or `foo.cpp` requires. It calls this C/C++ source function but does not return the result. Modify `main_foo` so that it takes inputs and returns outputs as required by your application.

Main Function

This section defines a `main` function that does the following:

- If your output language is C, it declares and names the variables `argc` and `argv` but casts them to void. If your output language is C++, the generated example main declares, but does not name, the variables `argc` and `argv`.
- Calls the initialize function `foo_initialize`, which is named for the alphabetically first entry-point function `foo` declared for code generation. Call the initialize function only once, even if you have multiple entry-point functions called in the function `main`.
- Calls each of the entry-point functions once.
- Calls the terminate function `foo_terminate`, which is named for the alphabetically first entry-point function `foo` declared for code generation. Call the terminate function only once, even if you have multiple entry-point functions called in the function `main`.
- Returns zero.

Modify the function `main`, including the inputs and outputs of `main` and of the entry-point functions, to meet the requirements of your application.

Contents of the File main.h

For the example main header file main.h, MATLAB Coder generates the following:

- “Include Guard” on page 24-54
- “Include Files” on page 24-54
- “Function Declarations” on page 24-54

By default, MATLAB Coder also generates comments in main.h that can help you modify the example main function to use in your application.

Include Guard

main.h uses an include guard to prevent the contents of the file from being included multiple times. The include guard contains the include files and function declarations within an #ifndef construct.

Include Files

main.h includes the header files required to call code that is not defined within it.

Function Declarations

main.h declares the function prototype for the main function that is defined in the example main source file main.c or main.cpp.

Related Examples

- “Incorporate Generated Code Using an Example Main Function” on page 24-20
- “Use an Example C Main in an Application” on page 24-23

More About

- “How MATLAB Coder Infers C/C++ Data Types” on page 26-9

Troubleshoot Failures in Deployed Code

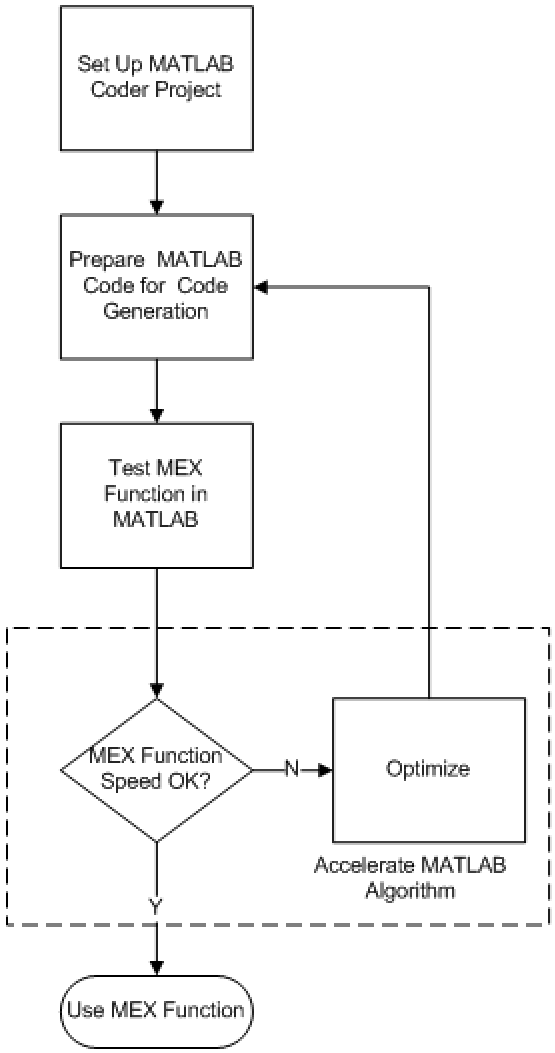
If your deployed code fails, consider regenerating the code with run-time error detection enabled. When you enable run-time error detection, the generated code includes code that detects and reports errors, such as out-of-bounds array indexing. If the code detects one of these errors, it reports a message and terminates the program. Running the code that includes the error checks helps you to see if one of these errors caused the failure.

Run-time error detection can affect the performance of the generated code. If performance is a consideration for your application, when you finish troubleshooting, regenerate the code with run-time error detection disabled.

See “Run-Time Error Detection and Reporting in Standalone C/C++ Code” on page 21-28 and “Generate Standalone Code That Detects and Reports Run-Time Errors” on page 21-30.

Accelerating MATLAB Algorithms

Workflow for Accelerating MATLAB Algorithms


Set Up MATLAB Coder Project
Prepare MATLAB Code for Code Generation
Test MEX Function in MATLAB
MEX Function Speed OK?
N
Optimize
Accelerate MATLAB Algorithm
Y
Use MEX Function

See Also

- “Set Up a MATLAB Coder Project” on page 17-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 18-2
- “Workflow for Testing MEX Functions in MATLAB” on page 19-3
- “Modifying MATLAB Code for Acceleration” on page 25-14

Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms

In this section...

When you choose a section of MATLAB code to accelerate, the following practices are recommended.

Accelerate Code That Dominates Execution Time

Find the section of MATLAB code that dominates run time. Accelerate this section of the code using a MEX function as follows:

1. Place this section of the code inside a separate MATLAB function.
2. From this MATLAB function, generate a MEX function.
3. From your original MATLAB code, call the MEX function.

To find the execution time of each MATLAB instruction, use MATLAB Profiler.

- To open the Profiler from the command line, type `profile viewer`.
- To open Profiler from the MATLAB Editor window, under the **Editor** tab, click **Run and Time**.

For more information about using the Profiler to measure run time of MATLAB code, see “Profile to Improve Performance” (MATLAB).

Include Loops Inside MEX Function

Instead of calling a MEX function inside a loop in the MATLAB code, include the loop inside the MEX function. Including the loop eliminates the overheads in calling the MEX function for every run of the loop.

For example, the following code finds the greatest element in every row of a 1000–by–1000 matrix, `mat`. You can accelerate sections 1,2, and 3 using a MEX function.:

```
% Section 1 begins
for i = 1:10000

   % Section 2 begins
   max = mat(i,0); % Initialize max
   for j = 1:10000

     % Section 3 begins
     if (mat(i,j) > max)
       max = mat(i,j) % Store the current maximum
     end
     % Section 3 ends

   end
   % Section 2 ends

end
% Section 1 ends
```

Accelerate section 1 using a MEX function. Accelerate section 1 first so that the MEX function is called only once.. If you cannot accelerate section 1 first, then accelerate sections 2 or 3, in that order. If section 2 (or 3) is accelerated using a MEX function, the function is called 10000 (or 10000 × 10000) times.

Avoid Generating MEX Functions from Unsupported Functions

Check that the section of MATLAB code that you accelerate does not contain many functions and language features that are unsupported by MATLAB Coder. For a list of supported functions, see "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List" on page 3-2.

Note: In certain situations, you might have to accelerate sections of code even though they contain a few unsupported functions. Declare an unsupported function as extrinsic to invoke the original MATLAB function instead of the code generated for the function. You can declare a function as extrinsic by using `coder.extrinsic` or wrapping it in an `feval` statement. See "Extrinsic Functions" on page 13-9.

Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time

Use MEX functions to accelerate MATLAB code only if user-generated code dominates the run time.

Avoid generating MEX functions if computationally intensive, built-in MATLAB functions dominate the run time. These functions are pre-compiled and optimized, so the MATLAB code is not accelerated significantly using a MEX function. Examples of such functions include `svd`, `eig` ,`fft`, `qr`, `lu`.

Tip: You can invoke computationally intensive, built-in MATLAB functions from your MEX function. Declare the MATLAB function as extrinsic using `coder.extrinsic` or wrap it in an `feval` statement. For more information, see "Extrinsic Functions" on page 13-9.

Minimize MEX Function Calls

Accelerate as much of the MATLAB code as possible using one MEX function instead of several MEX functions called at lower levels. This minimizes the overheads in calling the MEX functions.

For example, consider the function,`testfunc`,which calls two functions,`testfunc_1` and `testfunc_2`:

```
function [y1,y2] = testfunc(x1,x2)
  y1 = testfunc_1(x1,x2);
  y2 = testfunc_2(x1,x2);
end
```

Instead of generating MEX functions individually for `testfunc_1` and `testfunc_2`, and then calling the MEX functions in `testfunc`, generate a MEX function for `testfunc` itself.

Edge Detection on Images

This example shows how to generate a standalone C library from MATLAB code that implements a simple Sobel filter that performs edge detection on images. The example also shows how to generate and test a MEX function in MATLAB prior to generating C code to verify that the MATLAB code is suitable for code generation.

Prerequisites

There are no prerequisites for this example.

Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will only contain the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), you should change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_edge_detection');
```

About the 'sobel' Function

The sobel.m function takes an image (represented as a double matrix) and a threshold value and returns an image with the edges detected (based on the threshold value).

```
type sobel
```

```
% edgeImage = sobel(originalImage, threshold)
% Sobel edge detection. Given a normalized image (with double values)
% return an image where the edges are detected w.r.t. threshold value.
function edgeImage = sobel(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = [1 2 1; 0 0 0; -1 -2 -1];
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Generate the MEX Function

Generate a MEX function using the 'codegen' command.

```
codegen sobel
```

Before generating C code, you should first test the MEX function in MATLAB to ensure that it is functionally equivalent to the original MATLAB code and that no run-time errors occur. By default, 'codegen' generates a MEX function named 'sobel_mex' in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Read in the Original Image

Use the standard 'imread' command.

```
im = imread('hello.jpg');
image(im);
```

Convert Image to a Grayscale Version

Convert the color image (shown above) to an equivalent grayscale image with normalized values (0.0 for black, 1.0 for white).

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:
```

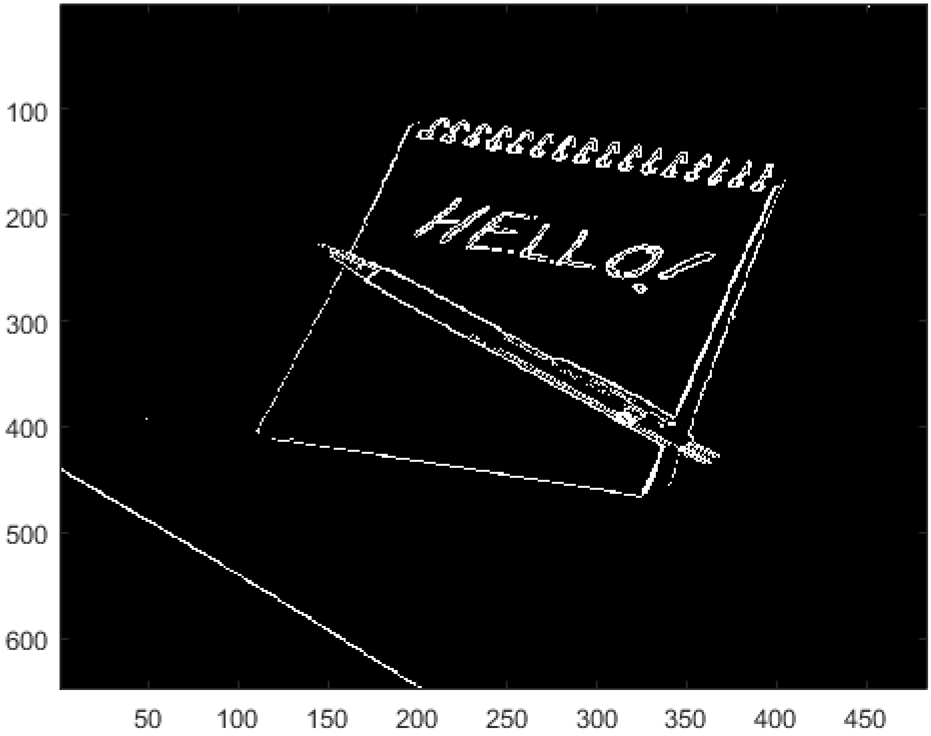
Run the MEX Function (The Sobel Filter)

Pass the normalized image and a threshold value.

```
edgeIm = sobel_mex(gray, 0.7);
```

Display the Result

```
im3 = repmat(edgeIm, [1 1 3]);
image(im3);
```



Generate Standalone C Code

```
codegen -config coder.config('lib') sobel
```

Using 'codegen' with the '-config coder.config('lib')' option produces a standalone C library. By default, the code generated for the library is in the folder codegen/lib/sobel/

Inspect the Generated Function

```
type codegen/lib/sobel/sobel.c
```

```
/*
 * File: sobel.c
 *
 * MATLAB Coder version            : 3.3
 * C/C++ source code generated on  : 15-Feb-2017 04:06:37
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "sobel.h"
#include "sobel_emxutil.h"
#include "sqrt.h"
#include "conv2.h"

/* Function Definitions */

/*
 * Arguments    : const emxArray_real_T *originalImage
 *                double threshold
 *                emxArray_uint8_T *edgeImage
 * Return Type  : void
 */
void sobel(const emxArray_real_T *originalImage, double threshold,
           emxArray_uint8_T *edgeImage)
{
  emxArray_real_T *H;
  emxArray_real_T *V;
  int b_H;
  int c_H;
  emxInit_real_T(&H, 2);
  emxInit_real_T(&V, 2);

  /*  edgeImage = sobel(originalImage, threshold) */
  /*  Sobel edge detection. Given a normalized image (with double values) */
  /*  return an image where the edges are detected w.r.t. threshold value. */
  conv2(originalImage, H);
  b_conv2(originalImage, V);
  b_H = H->size[0] * H->size[1];
  emxEnsureCapacity((emxArray__common *)H, b_H, sizeof(double));
  b_H = H->size[0];
  c_H = H->size[1];
  c_H *= b_H;
  for (b_H = 0; b_H < c_H; b_H++) {
    H->data[b_H] = H->data[b_H] * H->data[b_H] + V->data[b_H] * V->data[b_H];
  }

  emxFree_real_T(&V);
  b_sqrt(H);
  b_H = edgeImage->size[0] * edgeImage->size[1];
  edgeImage->size[0] = H->size[0];
  edgeImage->size[1] = H->size[1];
  emxEnsureCapacity((emxArray__common *)edgeImage, b_H, sizeof(unsigned char));
  c_H = H->size[0] * H->size[1];
  for (b_H = 0; b_H < c_H; b_H++) {
    edgeImage->data[b_H] = (unsigned char)((H->data[b_H] > threshold) * 255U);
  }

  emxFree_real_T(&H);
}

/*
 * File trailer for sobel.c
 *
 * [EOF]
 */
```

Cleanup

Remove files and return to original folder

Run Command: Cleanup

```
cleanup
```

Accelerate MATLAB Algorithms

For many applications, you can generate MEX functions to accelerate MATLAB algorithms. If you have a Fixed-Point Designer license, you can generate MEX functions to accelerate fixed-point MATLAB algorithms. After generating a MEX function, test it in MATLAB to verify that its operation is functionally equivalent to the original MATLAB algorithm. Then compare the speed of execution of the MEX function with that of the MATLAB algorithm. If the MEX function speed is not sufficiently fast, you might improve it using one of the following methods:

- Choosing a different C/C++ compiler.

  It is important that you use a C/C++ compiler that is designed to generate high performance code.

  Note: The default MATLAB compiler for Windows 64-bit platforms, `lcc`, is designed to generate code quickly. It is not designed to generate high performance code.
- “Modifying MATLAB Code for Acceleration” on page 25-14
- “Control Run-Time Checks” on page 25-15

Modifying MATLAB Code for Acceleration

How to Modify Your MATLAB Code for Acceleration

You might improve the efficiency of the generated code using one of the following optimizations:

- “Unroll `for`-Loops” on page 28-37
- “Inline Code” on page 28-10
- “Eliminate Redundant Copies of Function Inputs” on page 28-7

Control Run-Time Checks

In this section...
"Types of Run-Time Checks" on page 25-15
"When to Disable Run-Time Checks" on page 25-15
"How to Disable Run-Time Checks" on page 25-16

Types of Run-Time Checks

The code generated for your MATLAB functions includes the following run-time checks and external calls to MATLAB functions.

- Memory integrity checks

  These checks detect violations of memory integrity in code generated for MATLAB functions and stop execution with a diagnostic message.

  Caution: These checks are enabled by default. Without memory integrity checks, violations result in unpredictable behavior.

- Responsiveness checks in code generated for MATLAB functions

  These checks enable periodic checks for Ctrl+C breaks in code generated for MATLAB functions. Enabling responsiveness checks also enables graphics refreshing.

  Caution: These checks are enabled by default. Without these checks, the only way to end a long-running execution might be to terminate MATLAB.

- Extrinsic calls to MATLAB functions

  Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information about extrinsic functions, see "Declaring MATLAB Functions as Extrinsic Functions" on page 13-10.

When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more generated code and slower MEX function execution than generating code with the checks disabled.

Similarly, extrinsic calls are time consuming and increase memory usage and execution time. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster MEX function execution. The following table lists issues to consider when disabling run-time checks and extrinsic calls.

Consider disabling...	Only if...
Memory integrity checks	You have already verified that array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C.
Extrinsic calls	You are using extrinsic calls only for functions that do not affect application results.

How to Disable Run-Time Checks

You can disable run-time checks explicitly from the project settings dialog box, the command line, or a MEX configuration dialog box.

Disabling Run-Time Checks Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `MEX`.
3. Click **More Settings**.
4. On the **Speed** tab, clear **Ensure memory integrity**, **Enable responsiveness to CTRL+C and graphics refreshing**, or **Keep Extrinsic calls**, as applicable.

Disabling Run-Time Checks From the Command Line

1. In the MATLAB workspace, define the MEX configuration object:

   ```
   mexcfg = coder.config('mex');
   ```
2. At the command line, set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties to false, as applicable:

   ```
   mexcfg.IntegrityChecks = false;
   ```

```
mexcfg.ExtrinsicCalls = false;
mexcfg.ResponsivenessChecks = false;
```

Algorithm Acceleration Using Parallel for-Loops (parfor)

In this section...

Parallel for-Loops (parfor) in Generated Code

To potentially accelerate execution, you can generate MEX functions or C/C++ code from MATLAB code that contains parallel for-loops (`parfor`-loops).

A `parfor`-loop, like the standard MATLAB `for`-loop, executes a series of statements (the loop body) over a range of values. Unlike the `for`-loop, however, the iterations of the `parfor`-loop can run in parallel on multiple cores on the target hardware.

Running the iterations in parallel might significantly improve execution speed of the generated code. For more information, see "How parfor-Loops Improve Execution Speed" on page 25-19.

Note: The parallel execution occurs only in generated MEX functions or C/C++ code; not the original MATLAB code. To accelerate your MATLAB code, generate a MEX function from the `parfor`-loop. Then, call the MEX function from your code. For more information, see "Workflow for Accelerating MATLAB Algorithms" on page 25-2.

MATLAB Coder software uses the Open Multiprocessing (OpenMP) application interface to support shared-memory, multicore code generation. If you want distributed parallelism, use the Parallel Computing Toolbox™ product. By default, MATLAB Coder uses up to as many cores as it finds available. If you specify the number of threads to use, MATLAB Coder uses at most that number of cores for the threads, even if additional cores are available. For more information, see `parfor`.

Because the loop body can execute in parallel on multiple threads, it must conform to certain restrictions. If MATLAB Coder software detects loops that do not conform to `parfor` specifications, it produces an error. For more information, see "parfor Restrictions" on page 25-20.

How parfor-Loops Improve Execution Speed

A `parfor`-loop might provide better execution speed than its analogous `for`-loop because several threads can compute concurrently on the same loop.

Each execution of the body of a `parfor`-loop is called an iteration. The threads evaluate iterations in arbitrary order and independently of each other. Because each iteration is independent, they do not have to be synchronized. If the number of threads is equal to the number of loop iterations, each thread performs one iteration of the loop. If there are more iterations than threads, some threads perform more than one loop iteration.

For example, when a loop of 100 iterations runs on 20 threads, each thread executes five iterations of the loop simultaneously. If your loop takes a long time to run because of the large number of iterations or individual iterations being lengthy, you can reduce the run time significantly using multiple threads. In this example, you might not, however, get 20 times improvement in speed because of parallelization overheads, such as thread creation and deletion.

When to Use parfor-Loops

Use `parfor` when you have:

- Many iterations of a simple calculation. `parfor` divides the loop iterations into groups so that each thread executes one group of iterations.
- A loop iteration that takes a long time to execute. `parfor` executes the iterations simultaneously on different threads. Although this simultaneous execution does not reduce the time spent on an individual iteration, it might significantly reduce overall time spent on the loop.

When Not to Use parfor-Loops

Do not use `parfor` when:

- An iteration of your loop depends on other iterations. Running the iterations in parallel can lead to erroneous results.

  To help you avoid using `parfor` when an iteration of your loop depends on other iterations, MATLAB Coder specifies a rigid classification of variables. For more information, see “Classification of Variables in parfor-Loops” on page 25-26. If MATLAB Coder detects loops that do not conform to the parfor specifications, it does not generate code and produces an error.

Reductions are an exception to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order. For more information, see “Reduction Variables” on page 25-28.

- There are only a few iterations that perform some simple calculations.

Note: For small number of loop iterations, you might not accelerate execution due to parallelization overheads. Such overheads include time taken for thread creation, data synchronization between threads, and thread deletion.

parfor-Loop Syntax

- For a parfor-loop, use this syntax:

```
parfor i = InitVal:EndVal
parfor (i = InitVal:EndVal)
```

- To specify the maximum number of threads, use this syntax:

```
parfor (i = InitVal:EndVal,NumThreads)
```

For more information, see parfor.

parfor Restrictions

- The parfor loop does not support the syntax:

```
parfor (i=initVal:step:endVal)
parfor i=initVal:step:endVal
```

- You must use a compiler that supports the Open Multiprocessing (OpenMP) application interface. See http://www.mathworks.com/support/compilers/current_release/. If you use a compiler that does not support OpenMP, MATLAB Coder treats the parfor-loops as for-loops. In the generated MEX function or C/C++ code, the loop iterations run on a single thread.
- The OpenMP application interface is not compatible with JIT MEX compilation. See “JIT Compilation Does Not Support OpenMP” on page 30-3.

- The type of the loop index must be representable by an integer type on the target hardware. Use a type that does not require a multiword type in the generated code.
- `parfor` for standalone code generation requires the toolchain approach for building executables or libraries. Do not change settings that cause the code generator to use the template makefile approach. See “Project or Configuration is Using the Template Makefile” on page 23-22.
- Do not use the following constructs in the body of a `parfor` loop:

  - **Nested parfor-loops**

    You can have a `parfor` loop inside another `parfor`-loop. However, the inner `parfor` loop will be executed on a single thread as an ordinary `for`-loop.

    Inside a `parfor` loop, you can call a function that contains another `parfor`-loop.
  - **Break and return statements**

    You cannot use `break` or `return` statements inside a `parfor`-loop.
  - **Global variables**

    You cannot write to a global variable inside a `parfor`-loop.
  - **Reductions on MATLAB classes**

    You cannot use reductions on MATLAB classes inside a `parfor`-loop.
  - **Reductions on `char` variables**

    You cannot use reductions on `char` variables inside a `parfor`-loop.

    For example, you cannot generate C code for the following MATLAB code:

    ```
    c = char(0);
    parfor i=1:10
      c = c + char(1);
    end
    ```
    In the `parfor`-loop, MATLAB makes `c` a double. For code generation, `c` cannot change type.
  - **Reductions using external C code**

    You cannot use `coder.ceval` in reductions inside a `parfor`-loop.. For example, you cannot generate code for the following `parfor`-loop:

```
parfor i=1:4
  y=coder.ceval('myCFcn',y,i);
end
```

Instead, write a local function that calls the C code using `coder.ceval` and call this function in the `parfor`-loop. For example:

```
parfor i=1:4
  y = callMyCFcn(y,i);
end
...
function y = callMyCFcn(y,i)
 y = coder.ceval('mCyFcn', y , i);
end
```

- **Extrinsic function calls**

  You cannot call extrinsic functions using `coder.extrinsic` inside a `parfor`-loop. Calls to functions that contain extrinsic calls result in a run-time error.

- **Inlining functions**

  MATLAB Coder does not inline functions into `parfor`-loops, including functions that use `coder.inline('always')`.

- **Unrolling loops**

  You cannot use `coder.unroll` inside a `parfor`-loop.

  If a loop is unrolled inside a `parfor`-loop, MATLAB Coder cannot classify the variable. For example:

```
for j=coder.unroll(3:6)
  y(i,j)=y(i,j)+i+j;
end
```

  This code is unrolled to:

```
y(i,3)=y(i,3)+i+3;
...
y(i,6)=y(i,6)+i+6;
```

  In the unrolled code, MATLAB Coder cannot classify the variable `y` because `y` is indexed in different ways inside the `parfor`-loop.

  MATLAB Coder does not support variables that it cannot classify. For more information, see “Classification of Variables in `parfor`-Loops” on page 25-26.

- **varargin/varargout**

  You cannot use varargin or varargout inside a parfor-loop.

Control Compilation of parfor-Loops

By default, MATLAB Coder generates code that can run the `parfor`-loop on multiple threads. To treat the `parfor`-loops as for-loops that run on a single thread, disable `parfor`:

- By using the `codegen` function with `-O disable:openmp` option at the command line.
- By setting **Enable OpenMP library if possible** to `No` under **All Settings** tab in the project settings dialog box.

When to Disable parfor

Disable parfor if you want to:

- Compare the execution times of the serial and parallel versions of the generated code.
- Investigate failures. If the parallel version of the generated code fails, disable `parfor` and generate a serial version to facilitate debugging.
- Use C compilers that do not support OpenMP.

Reduction Assignments in parfor-Loops

What are Reduction Assignments?

Reduction assignments, or *reductions*, are an exception to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the loop iterations together, but is independent of the iteration order. For a list of supported reduction variables see “Reduction Variables” on page 25-28.

Multiple Reductions in a parfor-Loop

You can perform the same reduction assignment multiple times within a `parfor`-loop provided that you use the same data type each time.

For example, in the following `parfor`-loop, `u(i)` and `v(i)` must be the same type.

```
parfor i = 1:10;
  X = X + u(i);
  X = X + v(i);
end
```

Similarly, the following example is valid provided that `u(i)` and `v(i)` are the same type.

```
parfor i=1:10
  r = foo(r,u(i));
  r = foo(r,v(i));
end
```

Classification of Variables in parfor-Loops

In this section...

Overview

MATLAB Coder classifies variables inside a `parfor`-loop into one of the categories in the following table. It does not support variables that it cannot classify. If a `parfor`-loop contains variables that cannot be uniquely categorized or if a variable violates its category restrictions, the `parfor`-loop generates an error.

Classification	Description
Loop	Serves as a loop index for arrays
Sliced	An array whose segments are operated on by different iterations of the loop
Broadcast	A variable defined before the loop whose value is used inside the loop, but not assigned inside the loop
Reduction	Accumulates a value across iterations of the loop, regardless of iteration order
Temporary	A variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop

Each of these variable classifications appears in this code fragment:

```
a=0;
c=pi;
z=0;
r=rand(1,10);
parfor i=1:10
    a=i;    % 'a' is a temporary variable
    z=z+i;   % 'z' is a reduction variable
    b(i)=r(i);  % 'b' is a sliced output variable;
```

```
                % 'r' a sliced input variable
    if i<=c   % 'c' is a broadcast variable
         d=2*a;  % 'd' is a temporary variable
    end
end
```

Sliced Variables

A *sliced variable* is one whose value can be broken up into segments, or *slices*, which are then operated on separately by different threads. Each iteration of the loop works on a different slice of the array.

In the next example, a slice of A consists of a single element of that array:

```
parfor i = 1:length(A)
   B(i) = f(A(i));
end
```

Characteristics of a Sliced Variable

A variable in a parfor-loop is sliced if it has the following characteristics:

- Type of First-Level Indexing — The first level of indexing is parentheses, ().
- Fixed Index Listing — Within the first-level parenthesis, the list of indices is the same for all occurrences of a given variable.
- Form of Indexing — Within the list of indices for the variable, exactly one index involves the loop variable.
- Shape of Array — In assigning to a sliced variable, the right-hand side of the assignment is not [] or '' (these operators indicate deletion of elements).

Type of First-Level Indexing. For a sliced variable, the first level of indexing is enclosed in parentheses, (). For example, A(...). If you reference a variable using dot notation, A.x, the variable is not sliced.

Variable A on the left is not sliced; variable A on the right is sliced:

```
A.q(i,12)                        A(i,12).q
```

Fixed Index Listing. Within the first-level parentheses of a sliced variable's indexing, the list of indices is the same for all occurrences of a given variable.

Variable B on the left is not sliced because B is indexed by i and i+1 in different places. Variable B on the right is sliced.

```
parfor i = 1:10
  B(i) = B(i+1) + 1;
end
```

```
parfor i = 1:10
  B(i+1) = B(i+1) + 1;
end
```

Form of Indexing. Within the list of indices for a sliced variable, one index is of the form `i`, `i+k`, `i-k`, `k+i`, or `k-i`.

- `i` is the loop variable.
- `k` is a constant or a simple (nonindexed) variable.
- Every other index is a constant, a simple variable, colon, or `end`.

When you use other variables along with the loop variable to index an array, you cannot set these variables inside the loop. These variables are constant over the execution of the entire `parfor` statement. You cannot combine the loop variable with itself to form an index expression.

In the following examples, `i` is the loop variable, `j` and `k` are nonindexed variables.

Variable A Is Not Sliced	Variable A Is Sliced

`A(i+f(k),j,:,3)`
`A(i,20:30,end)`
`A(i,:,s.field1)`
	`A(i+k,j,:,3)`
	`A(i,:,end)`
	`A(i,:,k)`

Shape of Array. A sliced variable must maintain a constant shape. In the following examples, the variable `A` is not sliced:

```
A(i,:) = [];
A(end + 1) = i;
```

Broadcast Variables

A *broadcast variable* is a variable other than the loop variable or a sliced variable that is not modified inside the loop.

Reduction Variables

A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order.

This example shows a `parfor`-loop that uses a scalar reduction assignment. It uses the reduction variable `x` to accumulate a sum across `10` iterations of the loop. The execution order of the iterations on the threads does not matter.

```
x = 0;
parfor i = 1:10
   x = x + i;
end
x
```

Where expr is a MATLAB expression, reduction variables appear on both sides of an assignment statement.

X = X + expr	X = expr + X
X = X - expr	See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 25-32
X = X .* expr	X = expr .* X
X = X * expr	X = expr * X
X = X & expr	X = expr & X
X = X | expr	X = expr | X
X = min(X, expr)	X = min(expr, X)
X = max(X, expr)	X = max(expr, X)
X=f(X, expr) Function f must be a user-defined function.	X = f(expr, X) See “Reduction Assignments, Associativity, and Commutativity of Reduction Functions” on page 25-32

Each of the allowed statements is referred to as a *reduction assignment*. A reduction variable can appear only in assignments of this type.

The following example shows a typical usage of a reduction variable X:

```
X = ...;            % Do some initialization of X
parfor i = 1:n
    X = X + d(i);
end
```

This loop is equivalent to the following, where each d(i) is calculated by a different iteration:

```
X = X + d(1) + ... + d(n)
```

If the loop were a regular for-loop, the variable X in each iteration would get its value either before entering the loop or from the previous iteration of the loop. However, this concept does not apply to parfor-loops.

In a parfor-loop, the value of X is not updated directly inside each thread. Rather, additions of d(i) are done in each thread, with i ranging over the subset of 1:n being performed on that thread. The software then accumulates the results into X.

Similarly, the reduction:

```
r=r<op> x(i)
```

is equivalent to:

```
r=r<op>x(1)] <op>x(2)...<op>x(n)
```

The operation <op> is first applied to x(1)...x(n), then the partial result is applied to r.

If operation <op> takes two inputs, it should meet one of the following criteria:

- Take two arguments of typeof(x(i)) and return typeof(x(i))
- Take one argument of typeof(r) and one of typeof(x(i)) and return typeof(r)

Rules for Reduction Variables

Use the same reduction function or operation in all reduction assignments

For a reduction variable, you must use the same reduction function or operation in all reduction assignments for that variable. In the following example, the parfor-loop on the left is not valid because the reduction assignment uses + in one instance, and * in another.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable

`parfor i = 1:n`
`  if A > 5*k`
`    A = A + 1;`
`  else`
`    A = A * 2;`
`  end`
	`parfor i = 1:n`
	`  if A > 5*k`
	`    A = A * 3;`
	`  else`
	`    A = A * 2;`
	`  end`

Restrictions on reduction function parameter and return types

A reduction r=r<op> x(i), should take arguments of typeof(x(i)) and return typeof(x(i)) or take arguments of typeof(r) and typeof(x(i)) and return typeof(r).

In the following example, in the invalid loop, r is a fixed-point type and 2 is not. To fix this issue, cast 2 to be the same type as r.

<table>
<tr><th>Invalid Use of Reduction Variable</th><th>Valid Use of Reduction Variable</th></tr>
<tr><td><pre>
function r = fiops(in)
r=fi(in,'WordLength',20,...
  'FractionLength',14,...
  'SumMode','SpecifyPrecision',...
  'SumWordLength',20,...
  'SumFractionLength',14,...
  'ProductMode', 'SpecifyPrecision',...
  'ProductWordLength',20,...
  'ProductFractionLength',14);
parfor i = 1:10
    r = r*2;
end
</pre></td><td><pre>
r=fi(in,'WordLength',20,...
  'FractionLength',14,...
  'SumMode','SpecifyPrecision',...
  'SumWordLength',20,...
  'SumFractionLength',14,...
  'ProductMode','SpecifyPrecision',...
  'ProductWordLength',20,...
  'ProductFractionLength',14);
T = r.numerictype;
F = r.fimath;
parfor i = 1:10
    r = r*fi(2,T,F);
end
</pre></td></tr>
</table>

In the following example, the reduction function `fcn` is invalid because it does not handle the case when input `u` is fixed point. (The `+` and `*` operations are automatically polymorphic.) You must write a polymorphic version of `fcn` to handle the expected input types.

Invalid Use of Reduction Variable	Valid Use of Reduction Variable

function [y0, y1, y2] = pfuserfcn(u)
    y0 = 0;
    y1 = 1;
    [F, N] = fiprops();
    y2 = fi(1,N,F);
    parfor (i=1:numel(u),12)
        y0 = y0 + u(i);
        y1 = y1 * u(i);
        y2 = fcn(y2, u(i));
    end
end
 	
function y = fcn(u, v)
  y = u * v;
end
	function [y0, y1, y2] = pfuserfcn(u)
	    y0 = 0;
	    y1 = 1;
	    [F, N] = fiprops();
	    y2 = fi(1,N,F);
	    parfor (i=1:numel(u),12)
	        y0 = y0 + u(i);
	        y1 = y1 * u(i);
	        y2 = fcn(y2, u(i));
	    end
	end
	% fcn handles inputs of type double
	% and fi
	function y = fcn(u, v)
	    if isa(u,'double')
	        y = u * v;
	    else
	        [F, N] = fiprops();
	        y = u * fi(v,N,F);
	    end
	end
	
	function [F, N] = fiprops()
	    N = numerictype(1,96,30);
	    F = fimath('ProductMode',...
	        'SpecifyPrecision',...
	        'ProductWordLength',96);
	end

Reduction Assignments, Associativity, and Commutativity of Reduction Functions

Reduction Assignments. MATLAB Coder does not allow reduction variables to be read anywhere in the `parfor`-loop except in reduction statements. In the following example, the call `foo(r)` after the reduction statement `r=r+i` causes the loop to be invalid.

```
function r = temp %#codegen
  r = 0;
  parfor i=1:10
    r = r + i;
    foo(r);
  end
end
```

Associativity in Reduction Assignments. If you use a user-defined function `f` in the definition of a reduction variable, to get deterministic behavior of `parfor`-loops, the reduction function `f` must be associative.

Note: If `f` is not associative, MATLAB Coder does not generate an error. You must write code that meets this recommendation.

To be associative, the function `f` must satisfy the following for all `a`, `b`, and `c`:

```
f(a,f(b,c)) = f(f(a,b),c)
```

Commutativity in Reduction Assignments. Some associative functions, including `+`, `.`, `min`, and `max`, are also commutative. That is, they satisfy the following for all `a` and `b`:

```
f(a,b) = f(b,a)
```

The function `f` of a reduction assignment must be commutative. If `f` is not commutative, different executions of the loop might result in different answers.

Unless `f` is a known noncommutative built-in, the software assumes that it is commutative.

Temporary Variables

A *temporary variable* is a variable that is the target of a direct, nonindexed assignment, but is not a reduction variable. In the following `parfor`-loop, `a` and `d` are temporary variables:

```
a = 0;
z = 0;
r = rand(1,10);
parfor i = 1:10
   a = i;          % Variable a is temporary
   z = z + i;
   if i <= 5
```

```
      d = 2*a;     % Variable d is temporary
   end
end
```

In contrast to the behavior of a `for`-loop, before each iteration of a `parfor`-loop, MATLAB Coder effectively clears temporary variables. Because the iterations must be independent, the values of temporary variables cannot be passed from one iteration of the loop to another. Therefore, temporary variables must be set inside the body of a `parfor`-loop, so that their values are defined separately for each iteration.

A temporary variable in the context of the `parfor` statement is different from a variable with the same name that exists outside the loop.

Uninitialized Temporaries

Because temporary variables are cleared at the beginning of every iteration, MATLAB Coder can detect certain cases in which an iteration through the loop uses the temporary variable before it is set in that iteration. In this case, MATLAB Coder issues a static error rather than a run-time error, because there is little point in allowing execution to proceed if a run-time error will occur. For example, suppose you write:

```
b = true;
parfor i = 1:n
   if b && some_condition(i)
      do_something(i);
      b = false;
   end
   ...
end
```

This loop is acceptable as an ordinary `for`-loop, but as a `parfor`-loop, `b` is a temporary variable because it occurs directly as the target of an assignment inside the loop. Therefore, it is cleared at the start of each iteration, so its use in the condition of the `if` is uninitialized. (If you change `parfor` to `for`, the value of `b` assumes sequential execution of the loop, so that `do_something(i)` is executed for only the lower values of `i` until `b` is set `false`.)

Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)

This example shows how to generate a MEX function for a MATLAB algorithm that contains a `parfor`-loop.

1 Write a MATLAB function that contains a `parfor`-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
  a(i,:)=real(fft(r(i,:)));
end
```

2 Generate a MEX function for `test_parfor`. At the MATLAB command line, enter:

```
codegen test_parfor
```

`codegen` generates a MEX function, `test_parfor_mex`, in the current folder.

3 Run the MEX function. At the MATLAB command line, enter:

```
test_parfor_mex
```

Because you did not specify the maximum number of threads to use, the generated MEX function executes the loop iterations in parallel on the maximum number of available cores.

Specify Maximum Number of Threads in parfor-Loops

This example shows how to specify the maximum number of threads to use for a parfor-loop. Because you specify the maximum number of threads to use, the generated MEX function executes the loop iterations in parallel on as many cores as available, up to the maximum number that you specify. If you specify more threads than there are cores available, the MEX function uses the available cores.

1 Write a MATLAB function, specify_num_threads, that uses one input to specify the maximum number of threads to execute a parfor-loop in the generated MEX function. For example:

```
function y = specify_num_threads(u) %#codegen
  y = ones(1,100);
  % u specifies maximum number of threads
  parfor (i = 1:100,u)
    y(i) = i;
  end
end
```

2 Generate a MEX function for specify_num_threads. Use -args {0} to specify that input u is a scalar double. Use -report to generate a code generation report. At the MATLAB command line, enter:

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

codegen generates a MEX function, specify_num_threads_mex, in the current folder.

3 Run the MEX function, specifying that it try to run in parallel on four threads. At the MATLAB command line, enter:

```
specify_num_threads_mex(4)
```

The generated MEX function runs on up to four cores. If less than four cores are available, the MEX function runs on the maximum number of cores available at the time of the call.

Troubleshooting parfor-Loops

Global or Persistent Declarations in parfor-Loop

The body of a parfor-loop cannot contain a global or persistent variable declaration.

Compiler Does Not Support OpenMP

The MATLAB Coder software uses the Open Multiprocessing (OpenMP) application interface to support shared-memory, multicore code generation. To generate a loop that runs in parallel on shared-memory, multicore platforms, use a compiler that supports OpenMP. OpenMP is enabled by default. If your compiler does not support OpenMP, MATLAB Coder generates a warning.

Install a compiler that supports OpenMP. See http://www.mathworks.com/support/compilers/current_release/.

Accelerating Simulation of Bouncing Balls

This example shows how to accelerate MATLAB algorithm execution using a generated MEX function. It uses the 'codegen' command to generate a MEX function for a complicated application that uses multiple MATLAB files. You can use 'codegen' to check that your MATLAB code is suitable for code generation and, in many cases, to accelerate your MATLAB algorithm. You can run the MEX function to check for run-time errors.

Prerequisites

There are no prerequisites for this example.

Create a New Folder and Copy Relevant Files

The following code will create a folder in your current working folder (pwd). The new folder will contain only the files that are relevant for this example. If you do not want to affect the current folder (or if you cannot generate files in this folder), change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_bouncing_balls');
```

About the 'run_balls' Function

The run_balls.m function takes a single input to specify the number of bouncing balls to simulate. The simulation runs and plots the balls bouncing until there is no energy left and returns the state (positions) of all the balls.

```
type run_balls
```

```
% balls = run_balls(n)
% Given 'n' number of balls, run a simulation until the balls come to a
% complete halt (or when the system has no more kinetic energy).
function balls = run_balls(n) %#codegen

coder.extrinsic('fprintf');

%   Copyright 2010-2013 The MathWorks, Inc.

% Seeding the random number generator will guarantee that we get
% precisely the same simulation every time we call this function.
old_settings = rng(1283,'V4');
```

```
% The 'cdata' variable is a matrix representing the colordata bitmap which
% will be rendered at every time step.
cdata = zeros(400,600,'uint8');

% Setup figure windows
im = setup_figure_window(cdata);

% Get the initial configuration for 'n' balls.
balls = initialize_balls(cdata, n);

energy = 2; % Something greater than 1
iteration = 1;
while energy > 1
    % Clear the bitmap
    cdata(:,:) = 0;
    % Apply one iteration of movement
    [cdata,balls,energy] = step_function(cdata,balls);
    % Render the current state
    cdata = draw_balls(cdata, balls);
    iteration = iteration + 1;
    if mod(iteration,10) == 0
        fprintf(1, 'Iteration %d\n', iteration);
    end
    refresh_image(im, cdata);
end
fprintf(1, 'Completed iterations: %d\n', iteration);

% Restore RNG settings.
rng(old_settings);
```

Generate the MEX Function

First, generate a MEX function using the command codegen followed by the name of the MATLAB file to compile. Pass an example input (-args 0) to indicate that the generated MEX function will be called with an input of type double.

```
codegen run_balls -args 0
```

The 'run_balls' function calls other MATLAB functions, but you need to specify only the entry-point function when calling 'codegen'.

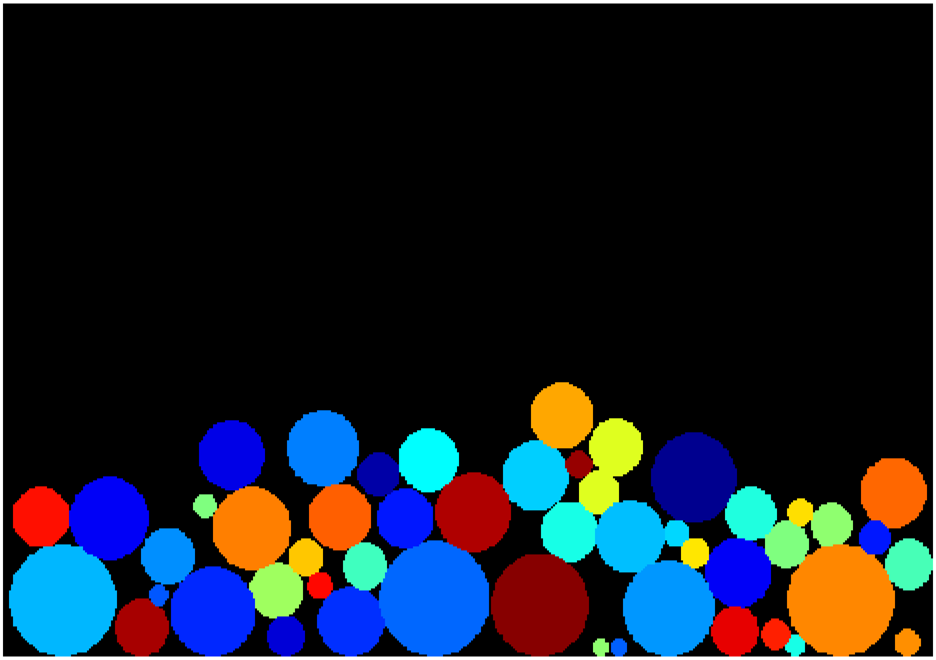
By default, 'codegen' generates a MEX function named 'run_balls_mex' in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Compare Results

Run and time the original 'run_balls' function followed by the generated MEX function.

```
tic, run_balls(50); t1 = toc;
tic, run_balls_mex(50); t2 = toc;
```

```
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 100
Iteration 110
Iteration 120
Iteration 130
Iteration 140
Iteration 150
Iteration 160
Iteration 170
Iteration 180
Iteration 190
Iteration 200
Iteration 210
Iteration 220
Iteration 230
Iteration 240
Iteration 250
Iteration 260
Iteration 270
Iteration 280
Completed iterations: 281
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 100
Iteration 110
Iteration 120
Iteration 130
Iteration 140
Iteration 150
Iteration 160
Iteration 170
Iteration 180
Iteration 190
Iteration 200
Iteration 210
Iteration 220
Iteration 230
Iteration 240
Iteration 250
Iteration 260
Iteration 270
Iteration 280
Completed iterations: 281
```



Estimated speed up is:

```
fprintf(1, 'Speed up: x ~%2.1f\n', t1/t2);
```

```
Speed up: x ~6.2
```

Clean Up

Remove files and return to original folder

Run Command: Cleanup

```
cleanup
```

Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler

This example shows how to use generated code to accelerate an application that you deploy with MATLAB® Compiler. The example accelerates an algorithm by using MATLAB® Coder™ to generate a MEX version of the algorithm. It uses MATLAB Compiler to deploy a standalone application that calls the MEX function. The deployed application uses the MATLAB® Runtime which enables royalty-free deployment to someone who does not have MATLAB.

This workflow is useful when:

- You want to deploy an application to a platform that the MATLAB Runtime supports.
- The application includes a computationally intensive algorithm that is suitable for code generation.
- The generated MEX for the algorithm is faster than the original MATLAB algorithm.
- You do not need to deploy readable C/C++ source code for the application.

The example application uses a DSP algorithm that requires the DSP System Toolbox™.

Create the MATLAB Application

For acceleration, it is a best practice to separate the computationally intensive algorithm from the code that calls it.

In this example, `myRLSFilterSystemIDSim` implements the algorithm. `myRLSFilterSystemIDApp` provides a user interface that calls `myRLSFilterSystemIDSim`.

`myRLSFilterSystemIDSim` simulates system identification by using recursive least-squares (RLS) adaptive filtering. It uses `dsp.VariableBandwidthFIRFilter` to model the unidentified system and `dsp.RLSFilter` to identify the FIR filter.

`myRLSFilterSystemIDApp` provides a user interface that you use to dynamically tune simulation parameters. It runs the simulation for a specified number of time steps or until you stop the simulation. It plots the results on scopes.

For details about this application, see (DSP System Toolbox) in the DSP System Toolbox documentation.

In a writable folder, create `myRLSFilterSystemIDSim` and `myRLSFilterSystemIDApp`. Alternatively, to access these files, click **Open Script**.

myRLSFilterSystemIDSim

```
function [tfe,err,pauseSim,stopSim,cutoffFreq,ff] = ...
    myRLSFilterSystemIDSim()
% myRLSFilterSystemIDSim implements the algorithm used in
% myRLSFilterSystemIDApp.
% This functions instantiates, initializes and steps through the System
% objects used in the algorithm.
%
% You can tune the cutoff frequency of the desired system and the
% forgetting factor of the RLS filter through the GUI that appears when
% myRLSFilterSystemIDApp is executed.

%   Copyright 2013-2016 The MathWorks, Inc.

%#codegen

% Instantiate and initialize System objects. The objects are declared
% persistent so that they are not recreated every time the function is
% called inside the simulation loop.
persistent rlsFilt sine unknownSys transferFunctionEstimator
if isempty(rlsFilt)
    % FIR filter models the unidentified system
    unknownSys = dsp.VariableBandwidthFIRFilter('SampleRate',1e4,...
        'FilterOrder',30,...
        'CutoffFrequency',.48 * 1e4/2);
    % RLS filter is used to identify the FIR filter
    rlsFilt = dsp.RLSFilter('ForgettingFactor',.99,...
        'Length',28);
    % Sine wave used to generate input signal
    sine = dsp.SineWave('SamplesPerFrame',1024,...
        'SampleRate',1e4,...
        'Frequency',50);
    % Transfer function estimator used to estimate frequency responses of
    % FIR and RLS filters.
    transferFunctionEstimator = dsp.TransferFunctionEstimator(...
        'FrequencyRange','centered',...
        'SpectralAverages',10,...
        'FFTLengthSource','Property',...
        'FFTLength',1024,...
        'Window','Kaiser');
end

[paramNew, simControlFlags] = HelperUnpackUDP();

tfe = 0;
err = 0;
cutoffFreq = 0;
ff = 0;
pauseSim = simControlFlags.pauseSim;
stopSim = simControlFlags.stopSim;

if simControlFlags.stopSim
    return;  % Stop the simulation
end
if simControlFlags.pauseSim
    return; % Pause the simulation (but keep checking for commands from GUI)
end

% Generate input signal - sine wave plus Gaussian noise
inputSignal = sine() +  .1 * randn(1024,1);

% Filter input though FIR filter
desiredOutput = unknownSys(inputSignal);

% Pass original and desired signals through the RLS Filter
[rlsOutput , err] = rlsFilt(inputSignal,desiredOutput);

% Prepare system input and output for transfer function estimator
inChans = repmat(inputSignal,1,2);
outChans = [desiredOutput,rlsOutput];

% Estimate transfer function
tfe = transferFunctionEstimator(inChans,outChans);

% Save the cutoff frequency and forgetting factor
cutoffFreq = unknownSys.CutoffFrequency;
ff = rlsFilt.ForgettingFactor;

% Tune FIR cutoff frequency and RLS forgetting factor
if ~isempty(paramNew)
    unknownSys.CutoffFrequency  = paramNew(1);
    rlsFilt.ForgettingFactor = paramNew(2);
    if simControlFlags.resetObj % reset System objects
```

```
        reset(rlsFilt);
        reset(unknownSys);
        reset(transferFunctionEstimator);
        reset(sine);
    end
end

end
```

myRLSFilterSystemIDApp

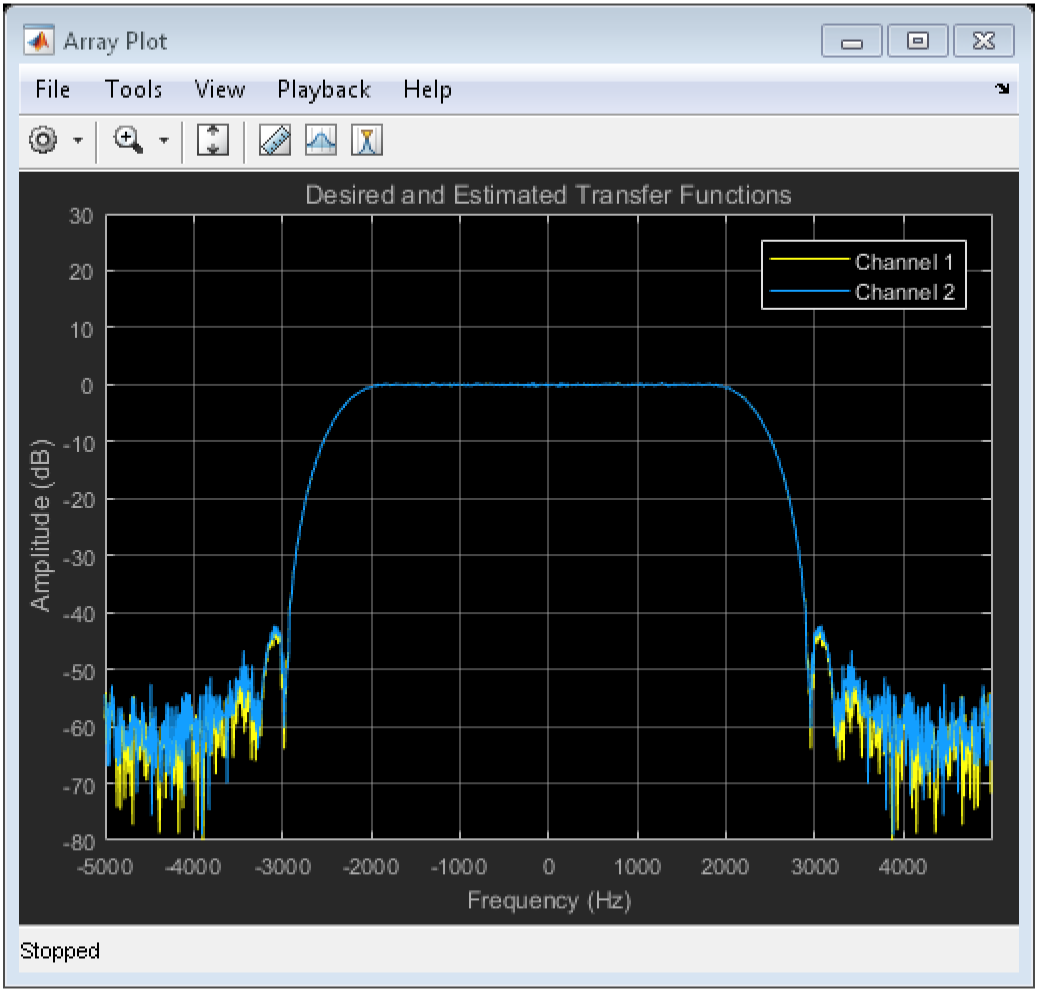
```
function scopeHandles = myRLSFilterSystemIDApp(numTSteps)
% myRLSFilterSystemIDApp initialize and execute RLS Filter
% system identification example. Then, display results using
% scopes. The function returns the handles to the scope and UI objects.
%
% Input:
%   numTSteps - number of time steps
% Outputs:
%   scopeHandles    - Handle to the visualization scopes            .

% Copyright 2013-2016 The MathWorks, Inc.


if nargin == 0
    numTSteps = Inf; % Run until user stops simulation.
end

% Create scopes
tfescope = dsp.ArrayPlot('PlotType','Line',...
    'Position',[8 696 520 420],...
    'YLimits',[-80 30],...
    'SampleIncrement',1e4/1024,...
    'YLabel','Amplitude (dB)',...
    'XLabel','Frequency (Hz)',...
    'Title','Desired and Estimated Transfer Functions',...
    'ShowLegend',true,...
    'XOffset',-5000);

msescope = dsp.TimeScope('SampleRate',1e4,'TimeSpan',.01,...
    'Position',[8 184 520 420],...
    'YLimits',[-300 10],'ShowGrid',true,...
    'YLabel','Mean-Square Error (dB)',...
    'Title','RLSFilter Learning Curve');

screen = get(0,'ScreenSize');
outerSize = min((screen(4)-40)/2, 512);
tfescope.Position = [8, screen(4)-outerSize+8, outerSize+8,...
    outerSize-92];
msescope.Position = [8, screen(4)-2*outerSize+8, outerSize+8, ...
    outerSize-92];

% Create UI to tune FIR filter cutoff frequency and RLS filter
%  forgetting factor
Fs = 1e4;
param = struct([]);
param(1).Name = 'Cutoff Frequency (Hz)';
param(1).InitialValue = 0.48 * Fs/2;
param(1).Limits = Fs/2 * [1e-5, .9999];
param(2).Name = 'RLS Forgetting Factor';
param(2).InitialValue = 0.99;
param(2).Limits = [.3, 1];
hUI = HelperCreateParamTuningUI(param, 'RLS FIR Demo');
set(hUI,'Position',[outerSize+32, screen(4)-2*outerSize+8, ...
    outerSize+8, outerSize-92]);

clear HelperUnpackUDP

% Execute algorithm
while(numTSteps>=0)

    drawnow limitrate;   % needed to process UI callbacks

    [tfe,err,pauseSim,stopSim] = myRLSFilterSystemIDSim();

    if stopSim     % If "Stop Simulation" button is pressed
        break;
    end
    if pauseSim
        continue;
    end

    % Plot transfer functions
    tfescope(20*log10(abs(tfe)));
    % Plot learning curve
    msescope(10*log10(sum(err.^2)));
    numTSteps = numTSteps - 1;
```

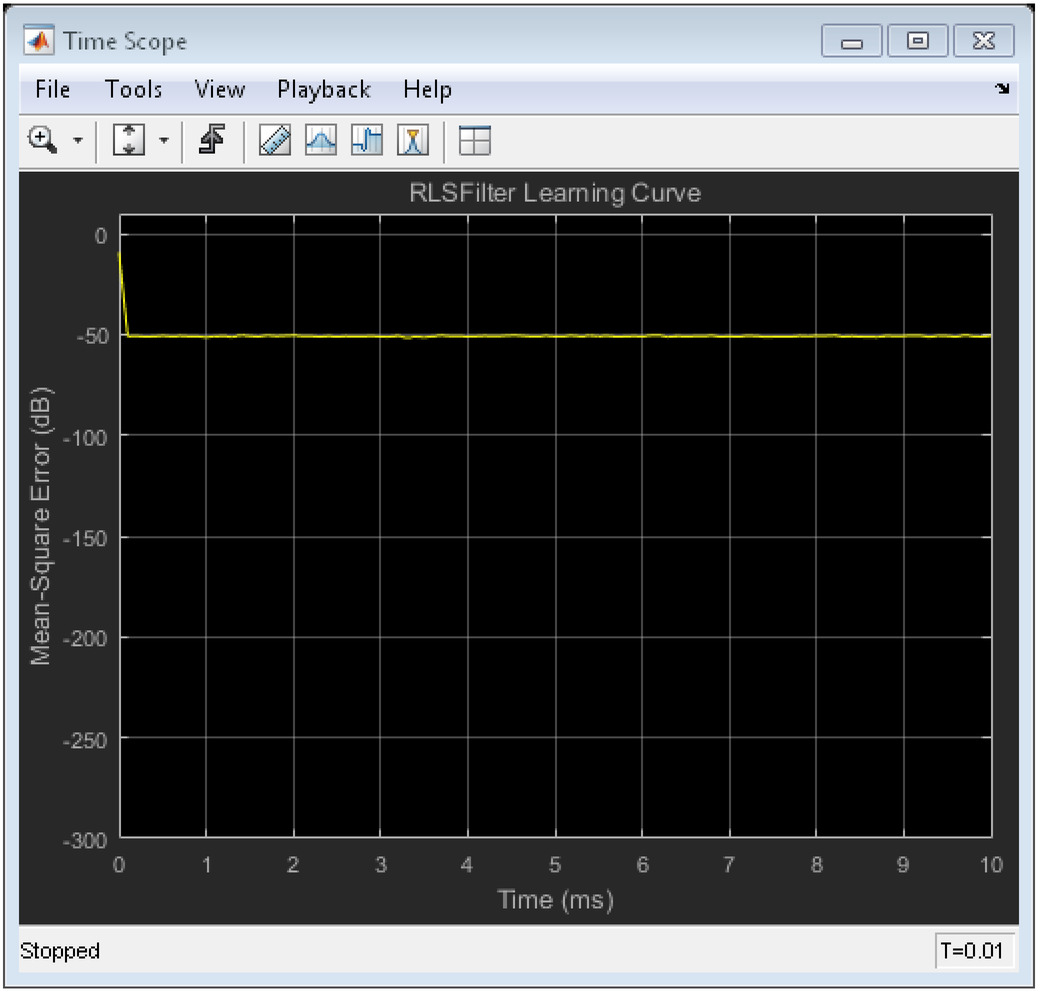
```
end

if ishghandle(hUI)  % If parameter tuning UI is open, then close it.
    delete(hUI);
    drawnow;
    clear hUI
end

scopeHandles.tfescope = tfescope;
scopeHandles.msescope = msescope;
end
```

Test the MATLAB Application

Run the system identification application for 100 time steps. The application runs the simulation for 100 time steps or until you click **Stop Simulation**. It plots the results on scopes.

```
scope1 = myRLSFilterSystemIDApp(100);
release(scope1.tfescope);
release(scope1.msescope);
```


Array Plot
File
Tools
View
Playback
Help
Desired and Estimated Transfer Functions
Channel 1
Channel 2
30
20
10
0
-10
-20
-30
-40
-50
-60
-70
-80
Amplitude (dB)
-5000
-4000
-3000
-2000
-1000
0
1000
2000
3000
4000
Frequency (Hz)
Stopped



Prepare Algorithm for Acceleration

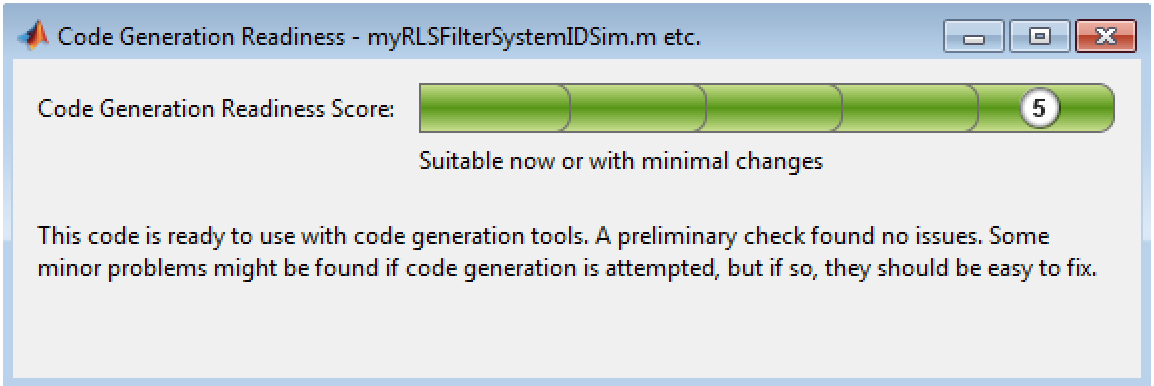
When you use MATLAB Coder to accelerate a MATLAB algorithm, the code must be suitable for code generation.

1. Make sure that `myRLSFilterSystemIDSim.m` includes the `%#codegen` directive after the function signature.

This directive indicates that you intend to generate code for the function. In the MATLAB Editor, it enables the code analyzer to detect code generation issues.

2. Screen the algorithm for unsupported functions or constructs.

```
coder.screener('myRLSFilterSystemIDSim');
```



Code Generation Readiness - myRLSFilterSystemIDSim.m etc.

Code Generation Readiness Score: 5

Suitable now or with minimal changes

This code is ready to use with code generation tools. A preliminary check found no issues. Some minor problems might be found if code generation is attempted, but if so, they should be easy to fix.

The code generation readiness tool does not find code generation issues in this algorithm.

Accelerate the Algorithm

To accelerate the algorithm, this example use the MATLAB Coder `codegen` command. Alternatively, you can use the MATLAB Coder app.

Generate a MEX function for `myRLSFilterSystemIDSim`.

```
codegen myRLSFilterSystemIDSim;
```

`codegen` creates the MEX function `myRLSFilterSystemIDSim_mex` in the current folder.

Compare MEX Function and MATLAB Function Performance

1. Time 100 executions of `myRLSFilterSystemIDSim`.

```
clear myRLSFilterSystemIDSim
disp('Running the MATLAB function ...')
tic
```

```
nTimeSteps = 100;
for ind = 1:nTimeSteps
     myRLSFilterSystemIDSim();
end
tMATLAB = toc;
```

```
Running the MATLAB function ...
```

2. Time 100 executions of `myRLSFilterSystemIDSim_mex`.

```
clear myRLSFilterSystemIDSim
disp('Running the MEX function ...')
tic
for ind = 1:nTimeSteps
    myRLSFilterSystemIDSim_mex();
end
tMEX = toc;

disp('RESULTS:')
disp(['Time for original MATLAB function: ', num2str(tMATLAB),...
     ' seconds']);
disp(['Time for MEX function: ', num2str(tMEX), ' seconds']);
disp(['The MEX function is ', num2str(tMATLAB/tMEX),...
    ' times faster than the original MATLAB function.']);
```

```
Running the MEX function ...
RESULTS:
Time for original MATLAB function: 3.5801 seconds
Time for MEX function: 0.35481 seconds
The MEX function is 10.0902 times faster than the original MATLAB function.
```

Optimize the MEX code

You can sometimes generate faster MEX by using a different C/C++ compiler or by using certain options or optimizations. See “Accelerate MATLAB Algorithms”.

For this example, the MEX is sufficiently fast without further optimization.

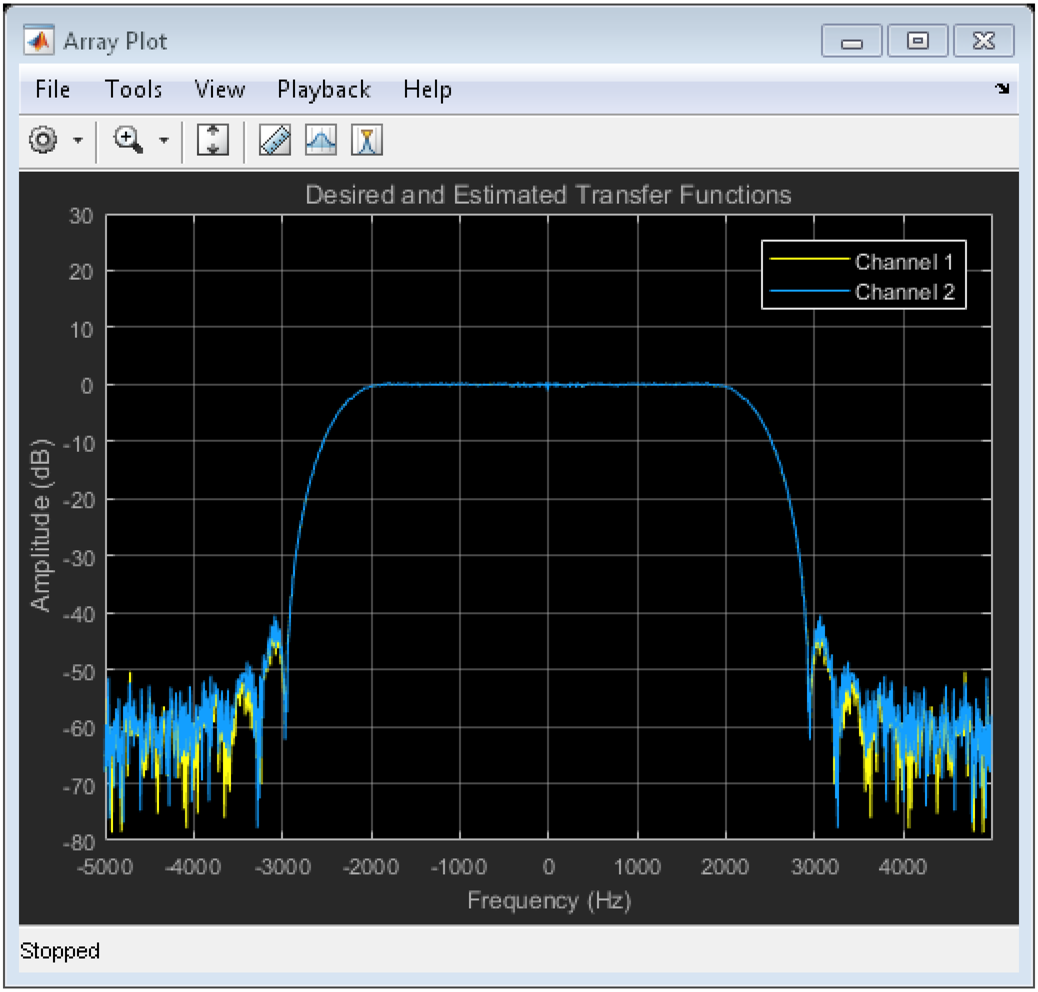
Modify the Application to Call the MEX Function

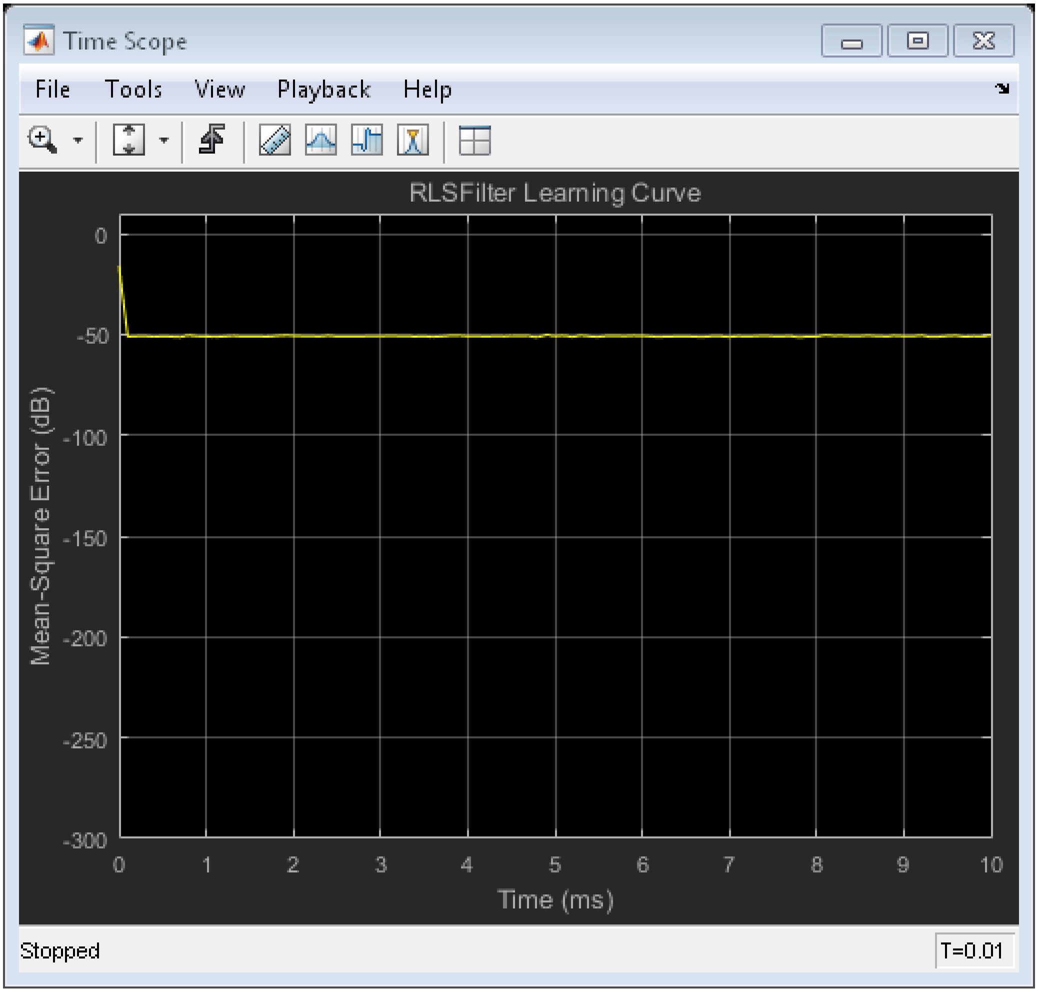
Modify `myRLSFilterSystemIDApp` so that it calls `myRLSFilterSystemIDSim_mex` instead of `myRLSFilterSystemIDSim`.

Save the modified function in `myRLSFilterSystemIDApp_acc.m`.

Test the Application with the Accelerated Algorithm

```
clear myRLSFilterSystemIDSim_mex;
scope2 = myRLSFilterSystemIDApp_acc(100);
release(scope2.tfescope);
release(scope2.msescope);
```





The behavior of the application that calls the MEX function is the same as the behavior of the application that calls the original MATLAB function. However, the plots update more quickly because the simulation is faster.

Create the Standalone Application

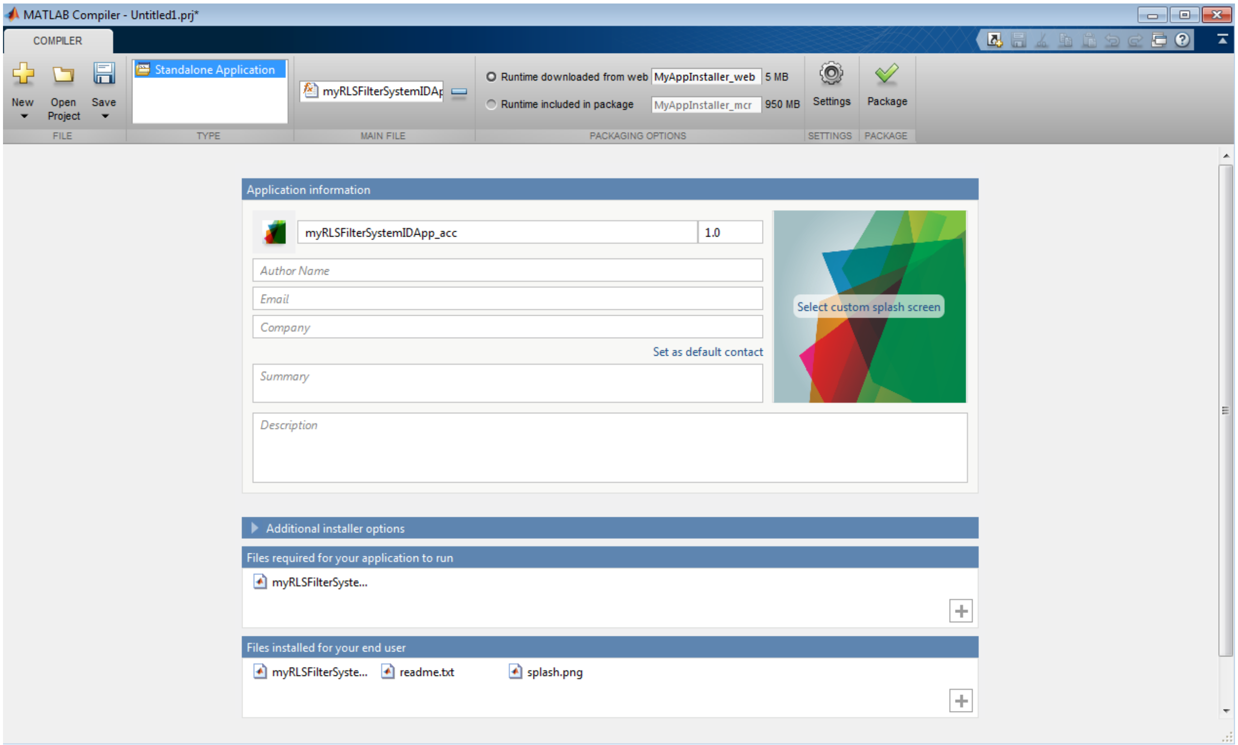
1. To open the Application Compiler App, on the **Apps** tab, under **Application Deployment**, click the app icon.

2. Specify that the main file is `myRLSFilterSystemIDApp_acc`.

The app determines the required files for this application. The app can find the MATLAB files and MEX-files that an application uses. You must add other types of files, such as MAT-files or images, as required files.

3. In the **Packaging Options** section of the toolstrip, make sure that the **Runtime downloaded from web** check box is selected.

This option creates an application installer that downloads and installs the MATLAB Runtime with the deployed MATLAB application.



MATLAB Compiler - Untitled1.prj*

TYPE: Standalone Application

MAIN FILE: myRLSFilterSystemIDAp

PACKAGING OPTIONS:
● Runtime downloaded from web MyAppInstaller_web 5 MB
○ Runtime included in package MyAppInstaller_mcr 950 MB

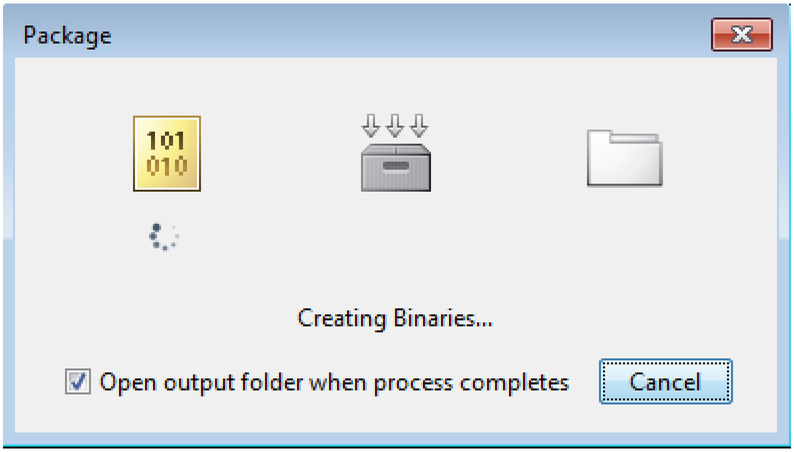
Application information
myRLSFilterSystemIDApp_acc 1.0
Author Name
Email
Company
Summary
Description

Files required for your application to run
myRLSFilterSyste...

Files installed for your end user
myRLSFilterSyste... readme.txt splash.png

4. Click **Package** and save the project.

5. In the Package window, make sure that the **Open output folder when process completes** check box is selected.



Package

Creating Binaries...

☑ Open output folder when process completes

When the packaging is complete, the output folder opens.

Install the Application

1. Open the `for_redistribution` folder.

2. Run `MyAppInstaller_web`.

3. If you connect to the internet by using a proxy server, enter the server settings.

4. Advance through the pages of the installation wizard.

- On the Installation Options page, use the default installation folder.
- On the Required Software page, use the default installation folder.
- On the License agreement page, read the license agreement and accept the license.
- On the Confirmation page, click **Install**.

If the MATLAB Runtime is not already installed, the installer installs it.

5. Click **Finish**.

Run the Application

1. Open a terminal window.

2. Navigate to the folder where the application is installed.

- For Windows®, navigate to `C:\Program Files\myRLSFilterSystemIDApp_acc`.
- For MAC OS x, navigate to `/Applicatons/myRLSFilterSystemIDApp_acc`.
- For Linux, navigate to `/usr/myRLSFilterSystemIDApp_acc`.

3. Run the application by using the appropriate command for your platform.

- For Windows, use `application\myRLSFilterSystemIDApp_acc`.
- For Mac OS x, use `myRLSFilterSystemIDApp_acc.app/Contents/MacOS/myRLSFilterSystemIDApp_acc`.
- For Linux, use `/myRLSFilterSystemIDApp_acc`.

Starting the application takes approximately the same amount of time as starting MATLAB.

More About

- (DSP System Toolbox)
- “Workflow for Accelerating MATLAB Algorithms” on page 25-2
- “Accelerate MATLAB Algorithms” on page 25-13
- “Create Standalone Application from MATLAB” (MATLAB Compiler)
- “About the MATLAB Runtime” (MATLAB Compiler)
- MATLAB Compiler Support for MATLAB and toolboxes.

Calling C/C++ Functions from Generated Code

External Function Calls from Generated Code

In this section...

Calling External Functions from Generated Code

You can call external functions from generated code. The external functions must have a C programming interface. The code generator provides functions for:

- Calling external functions from generated code.
- Passing arguments by reference to external code.
- Manipulating C/C++ data.

By using these functions, you gain unrestricted access to external code. Misuse of these functions or errors in your code can destabilize MATLAB when generating MEX functions.

Why Call External Functions from Generated Code?

Call external functions from generated code when you want to:

- Use legacy code.
- Use your own optimized functions instead of generated code.
- Interface your libraries and hardware with MATLAB functions.

How To Call External Functions

To call external functions, use one of the following methods:

- The `coder.ceval` function in your MATLAB code. `coder.ceval` passes function input and output arguments to C/C++ functions by value or by reference.

- The coder.ExternalDependency class to define methods that call the functions. These methods use the `coder.ceval` function. In your MATLAB code, use these methods to call external functions.

Define the called functions in external C/C++ source files, object files, or libraries. You must then include C/C++ source files, libraries, object files, and header files in the build configuration. See “Specify External File Locations” on page 24-14.

Pass Arguments by Reference to External Functions

By default, `coder.ceval` passes arguments by value to the C/C++ function whenever C/C++ supports passing arguments by value. You can pass MATLAB variables as arguments by reference to external C/C++ functions with the following constructs:

- `coder.ref` — pass value by reference.
- `coder.rref` — pass read-only value by reference.
- `coder.wref` — pass write-only value by reference.

These constructs offer the following benefits:

- Passing values by reference optimizes memory use.

  When you pass arguments by value, MATLAB Coder passes a copy of the value of each argument to the C/C++ function to preserve the original values. When you pass arguments by reference, MATLAB Coder does not copy values. If you need to pass large matrices to the C/C++ function, the memory savings can be significant.

  Passing write-only values by reference allows you to return multiple outputs.

  Use `coder.wref` to return multiple outputs from your C/C++ function, including arrays and matrices. Otherwise, the C/C++ function can return only a single scalar value through its `return` statement.

Do not store pointers that you pass to C/C++ functions because MATLAB Coder optimizes the code based on the assumption that you do not store the addresses of these variables. Storing the addresses might invalidate our optimizations leading to incorrect behavior. For example, if a MATLAB function passes a pointer to an array using `coder.ref`, `coder.rref`, or `coder.wref`, then the C/C++ function can modify the data in the array—but you should not store the pointer for future use.

When you pass arguments by reference using `coder.rref`, `coder.wref`, and `coder.ref`, the corresponding C/C++ function signature must declare these variables as pointers of the same data type. Otherwise, the C/C++ compiler generates a type mismatch error.

For example, suppose your MATLAB function calls an external C function `ctest`:

```
function y = fcn()
u = pi;

y = 0;
y = coder.ceval('ctest',u);
```

Now suppose the C function signature is:

```
double ctest(double *a)
```

When you compile the code, you get a type mismatch error because `coder.ceval` calls `ctest` with an argument of type `double` when `ctest` expects a pointer to a double-precision, floating-point value.

Match the types of arguments in `coder.ceval` with their counterparts in the C function. For instance, you can fix the error in the previous example by passing the argument by reference:

```
y = coder.ceval('ctest', coder.rref(u));
```

You can pass a reference to an element of a matrix. For example, to pass the second element of the matrix `v`, you can use the following code:

```
y = coder.ceval('ctest', coder.ref(v(1,2)));
```

Manipulate C Data

The construct `coder.opaque` allows you to manipulate C/C++ data that a MATLAB function does not recognize. You can store the opaque data in a variable or structure field and pass it to, or return it from, a C/C++ function using `coder.ceval`.

Declaring Opaque Data

The following example uses `coder.opaque` to declare a variable `f` as a `FILE *` type.

```
% This example returns its own source code by using
% fopen/fread/fclose.
function buffer = filetest
%#codegen
```

```
% Declare 'f' as an opaque type 'FILE *'
f = coder.opaque('FILE *', 'NULL');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));

% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
    % By default, MATLAB converts constant values
    % to doubles in generated code
    % so explicit type conversion to in32 is inserted.
    n = coder.ceval('fread', coder.ref(buffer(i)), int32(1), ...
        int32(numel(buffer)), f);
    i = i + n;
end
coder.ceval('fclose',f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB character vector
function y = cstring(x)
    y = [x char(0)];

% Remove character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
j = 1;
for i = 1:numel(buffer)
    if buffer(i) ~= char(13)
        buffer(j) = buffer(i);
        j = j + 1;
    end
end
buffer(i) = 0;
```

Call External Functions with coder.ceval

Workflow for Calling External Functions

To call external C/C++ functions from generated code:

1. Write your C/C++ functions in external source files or libraries.
2. Create header files, if required.

   The header file defines the data types used by the C/C++ functions that MATLAB Coder generates in code, as described in "Mapping MATLAB Types to C/C++ Types" on page 26-9.

   Tip: One way to add these type definitions is to include the header file `tmwtypes.h`, which defines general data types supported by MATLAB. This header file is in *`matlabroot`*`/extern/include`. Check the definitions in `tmwtypes.h` to determine if they are compatible with your target. If not, define these types in your own header files.

   ---
3. In your MATLAB function, add calls to `coder.ceval` to invoke your external C/C++ functions.

   You need one `coder.ceval` statement for each call to a C/C++ function. In your `coder.ceval` statements, use `coder.ref`, `coder.rref`, and `coder.wref` constructs as required (see "Pass Arguments by Reference to External Functions" on page 26-3).
4. Include the custom C/C++ functions in the build. See "Specify External File Locations" on page 24-14.
5. Check for compilation warnings about data type mismatches.

   Perform this check so that you catch type mismatches between C/C++ and MATLAB (see "How MATLAB Coder Infers C/C++ Data Types" on page 26-9).
6. Generate code and fix errors.

7 Run your application.

Best Practices for Calling External Code from Generated Code

The following are recommended practices when calling C/C++ code from generated code.

- **Start small.** — Create a test function and learn how `coder.ceval` and its related constructs work.
- **Use separate files.** — Create a file for each C/C++ function that you call. Make sure that you call the C/C++ functions with suitable types.
- In a header file, declare a function prototype for each function that you call, and include this header file in the generated code. For more information, see “Specify External File Locations” on page 24-14.

Return Multiple Values from C Functions

The C language restricts functions from returning multiple outputs; instead, they return only a single, scalar value. The constructs coder.ref and coder.wref allow MATLAB functions to exchange multiple outputs with the external C functions that they call.

For example, suppose you write a MATLAB function foo that takes two inputs x and y and returns three outputs a, b, and c. In MATLAB, you call this function as follows:

```
[a, b, c] = foo (x, y)
```

If you rewrite foo as a C function, you cannot return a, b, and c through the return statement. You can create a C function with multiple pointer type input arguments, and pass the output parameters by reference. For example:

```
foo(double x, double y, double *a, double *b, double *c)
```

Then you can call the C function with multiple outputs from a MATLAB function using coder.wref constructs:

```
coder.ceval ('foo', x, y, ...
    coder.wref(a), coder.wref(b), coder.wref(c));
```

Similarly, suppose that one of the outputs a is also an input argument. In this case, create a C function with multiple pointer type input arguments, and pass the output parameters by reference. For example:

```
foo(double *a, double *b, double *c)
```

Then call the C function from a MATLAB function using coder.wref and coder.rref constructs:

```
coder.ceval ('foo', coder.ref(a), coder.wref(b), coder.wref(c));
```

How MATLAB Coder Infers C/C++ Data Types

In this section...

Mapping MATLAB Types to C/C++ Types

The C/C++ type associated with a MATLAB variable or expression is based on the following properties:

- Class
- Size
- Complexity

By default, the MATLAB Coder software tries to use built-in C/C++ types in the generated code. If the target hardware supports the built-in C type, the software generates a built-in C type for these MATLAB types.

`int8`	`uint8`	`double`
`int16`	`uint16`	`single`
`int32`	`uint32`	`char`
`int64`	`uint64`	

The built-in C/C++ type that the code generator uses depends on the target hardware. You have the option to use MathWorks C/C++ data types instead of built-in C/C++ types. For information about setting this option, see "Specify Data Types Used in Generated Code" on page 20-38.

The following translation table shows how the MATLAB Coder software maps MATLAB types to MathWorks C/C++ data types.

MATLAB Type	MATLAB C/C++ Data Type	Reference Type for MATLAB C/ C++ Data Type
int8	int8_T	int8_T *
int16	int16_T	int16_T *
int32	int32_T	int32_T *
int64	See “Mapping 64-Bit Integer Types to C/C++” on page 26-10.	
uint8	uint8_T	uint8_T *
uint16	uint16_T	uint16_T *
uint32	uint32_T	uint32_T *
uint64	See “Mapping 64-Bit Integer Types to C/C++” on page 26-10.	
double	real_T	real_T *
single	real32_T	real32_T *
char	char_T	char *
logical	boolean_T	boolean_T *
fi	numerictype also influences the C/C++ type. Integer type varies according to the MATLAB fixed-point type, as described in “Mapping Fixed-Point Types to C/C++” on page 26-11.	
struct	The MATLAB Coder software maps structures to C/C++ types field-by-field. See “Mapping Structures to C/C++ Structures” on page 26-13 .	
complex	See “Mapping Complex Values to C/C++” on page 26-12.	
Multiword types	See “Mapping Multiword Types to C/C++” on page 26-14.	

Mapping 64-Bit Integer Types to C/C++

The C/C++ data type associated with a 64-bit integer MATLAB type depends on the sizes of the integer types on the target hardware. If a type wide enough for a 64-bit type does not exist, then a 64-bit type maps to a multiword type.

By default, MATLAB Coder software tries to map int64 and uint64 types to built-in C types. For a multiword type, the software uses a built-in C type for the array in the struct that represents the multiword type. You have the option to use MATLAB C/C++ data types instead of built-in types. The following table shows how 64 bit integer types map to MATLAB C/C++ data types.

MATLAB Type	MATLAB C/C++ Type	Multiword MATLAB C/C++ Type
int64	int64_T	int64m_T
uint64	uint64_T	uint64m_T
complex int64	cint64_T	cint64m_T
complex uint64	cuint64_T	cuint64m_T

See "Mapping Multiword Types to C/C++" on page 26-14.

Mapping Fixed-Point Types to C/C++

The numerictype properties of a fi object determine the C/C++ data type. By default, the code generator tries to use built-in C/C++ types. However, you can choose to use MATLAB C/C++ data types instead. The following table shows how the Signedness, WordLength, and FractionLength properties determine the MATLAB C/C++ data type. The MATLAB C/C++ data type is the next larger target word size that can store the fixed-point value, based on its word length. The sign of the integer type matches the sign of the fixed-point type.

Signedness	Word Length	Fraction Length	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
1	16	15	int16_T	int16_T *
1	13	10	int16_T	int16_T *
0	19	15	uint32_T	uint32_T *
1	8	7	int8_T	int8_T *

Mapping Arrays to C/C++

By default, the code generator tries to use built-in C/C++ types for arrays in the generated code. However, you can choose to use MATLAB C/C++ data types instead. The following translation table shows how MATLAB Coder software maps arrays to MATLAB C/C++ data types. In the first column, the arrays are specified by the MATLAB function `zeros`:

`zeros(`*`number of rows`*`, `*`number of columns`*`, `*`data type`*`)`
MATLAB array data is laid out in column major order.

Array	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
zeros(10, 5, 'int8')	int8_T	int8_T *
zeros(5, 10, 'int8')	int8_T	int8_T *
zeros(3, 7)	real_T	real_T *
zeros(10, 1, 'single')	real32_T	real32_T *

Mapping Complex Values to C/C++

The following translation table shows how the MATLAB Coder software infers complex values in generated code.

Complex	MATLAB C/C++ Data Type	Reference Type for MATLAB C/C++ Data Type
complex int8	cint8_T	cint8_T *
complex int16	cint16_T	cint16_T *
complex int32	cint32_T	cint32_T *
complex int64	See “Mapping 64-Bit Integer Types to C/C++” on page 26-10.	
complex uint8	cuint8_T	cuint8_T *
complex uint16	cuint16_T	cuint16_T *
complex uint32	cuint32_T	cuint32_T *
complex uint64	See “Mapping 64-Bit Integer Types to C/C++” on page 26-10.	
complex double	creal_T	creal_T *
complex single	creal32_T	creal32_T *

The MATLAB Coder software defines each complex value as a structure with a real component re and an imaginary component im, as in this example from tmwtypes.h:

```
typedef struct {
  real32_T re;/* Real component*/
  real32_T im;/* Imaginary component*/
} creal32_T;
```

MATLAB Coder uses the names re and im in generated code to represent the components of complex numbers. For example, suppose you define a variable x of type creal32_T. The generated code references the real component as x.re and the imaginary component as x.im.

If your C/C++ library requires a different representation, you can define your own versions of MATLAB Coder complex types. However, you *must* use the names re for the real components and im for the imaginary components in your definitions.

The MATLAB Coder software represents a matrix of complex numbers as an array of structures.

Mapping Structures to C/C++ Structures

The MATLAB Coder software maps structures to C/C++ types field-by-field. The order of the field items is preserved as the order in MATLAB. To control the name of the generated C/C++ structure type, or provide a definition, use the coder.cstructname function.

Note: If you are not using dynamic memory allocation, arrays in structures translate into single-dimension arrays, not pointers.

Mapping MATLAB Character Vectors to C/C++ Character Arrays

The MATLAB Coder software maps MATLAB character vectors to C/C++ character arrays. These character arrays are not C/C++ strings because they are not null-terminated. If you pass a MATLAB character vector to external C/C++ code that expects a C/C++ string, the generated C/C++ character array must be null-terminated. To generate a null-terminated C/C++ character array, append a zero to the end of the MATLAB character vector. For example, ['sample text' 0]. Otherwise, the generated code can crash without compiler errors or warnings.

A single character maps to a C/C++ char type, not a C/C++ string.

Mapping Multiword Types to C/C++

The MATLAB Coder software maps multiword types to structure types that contain an array of integers. The array dimensions depend on the size of the widest integer type on the target hardware. For example, for a 128-bit fixed-point type, if the widest integer type on the target hardware is 32-bits, the software generates a structure with an array of four 32-bit integers.

```
typedef struct
{
  unsigned int  chunks[4];
} uint128m_T;
```

If the widest integer type on the target hardware is `long` with a size of 64-bits, MATLAB Coder generates a structure with an array of two 64-bit long integers.

```
typedef struct
{
  unsigned long chunks[2];
} uint128m_T;
```

External Code Integration

External Code Integration for Code Generation

You can integrate external code with MATLAB code intended for code generation. The external code can be external libraries, object files, or C/C++ source code.

The basic workflow is:

1. Create the external code.
2. Call the external code from MATLAB code.
3. Specify the external file locations.
4. Generate code from the MATLAB code.

Call the external code and specify the file locations in one of the following ways:

- Use `coder.ExternalDependency` to encapsulate the interface to the external code. The `updateBuildInfo` method specifies file locations and other build information. Write methods that define the programming interface to the external functions. In your MATLAB code, use these methods to call the external functions.
- Use `coder.ceval` to call external functions from your MATLAB code. When you generate code, define the locations of external files.
- Use `coder.ceval` to call external functions from your MATLAB code. Use `coder.updateBuildInfo` to specify external file locations and update build information.

See Also

`coder.ceval` | coder.ExternalDependency | `coder.updateBuildInfo`

More About

- "Encapsulating the Interface to External Code" on page 27-3
- "Specify External File Locations" on page 24-14
- "External Function Calls from Generated Code" on page 26-2

Encapsulating the Interface to External Code

Use the `coder.ExternalDependency` class to encapsulate the interface between external code and MATLAB code intended for code generation. With the encapsulation, you can separate the details of the interface from your MATLAB code. The methods of `coder.ExternalDependency`:

- specify the location of external files
- update build information
- define the programming interface for external functions

In your MATLAB code, you can call the external code without providing build information.

The workflow is:

1. Write a class definition file for a class that derives from `coder.ExternalDependency`.
2. Store the class definition file in a folder on the MATLAB path.
3. In your MATLAB code, use a method of the class to call an external function.
4. Generate code from your MATLAB code.

See Also

coder.ExternalDependency

Related Examples

- “Encapsulate Interface to an External C Library” on page 27-6

More About

- “Best Practices for Using coder.ExternalDependency” on page 27-4

Best Practices for Using coder.ExternalDependency

In this section...

Terminate Code Generation for Unsupported External Dependency

The `isSupportedContext` method returns true if the external code interface is supported in the build context. If the external code interface is not supported, do not return false. Instead, use `error` to terminate code generation with an error message. For example:

```
function tf = isSupportedContext(ctx)
    if  ctx.isMatlabHostTarget()
        tf = true;
    else
        error('MyLibrary is not available for this target');
    end
end
```

Parameterize Methods for MATLAB and Generated Code

Parameterize methods that call external functions so that the methods run in MATLAB. For example:

```
...
if coder.target('MATLAB')
    % running in MATLAB, use built-in addition
    c = a + b;
else
    % running in generated code, call library function
    coder.ceval('adder_initialize');
end
...
```

Parameterize updateBuildInfo for Multiple Platforms

Parameterize the `updateBuildInfo` method to support multiple platforms. For example, use `coder.BuildConfig.getStdLibInfo` to get the platform-specific library file extensions.

```
...
    [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo()
% Link files
linkFiles =  strcat('adder', linkLibExt);
buildInfo.addLinkObjects(linkFiles, linkPath, linkPriority, ...
    linkPrecompiled, linkLinkOnly, group);
...
```

See Also

coder.BuildConfig | coder.ExternalDependency | `error`

Related Examples

- “Encapsulate Interface to an External C Library” on page 27-6

Encapsulate Interface to an External C Library

Use coder.ExternalDependency to encapsulate the interface to an external C dynamic linked library .

Write a function adder that returns the sum of its inputs.

```
function c = adder(a,b)
    %#codegen
    c = a + b;
end
```

Generate a library that contains adder.

```
codegen('adder','-args', {-2,5}, '-config:dll', '-report');
```

Write the class definition file AdderAPI.m to encapsulate the library interface.

```
%=================================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=================================================================

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end

        function tf = isSupportedContext(ctx)
            if  ctx.isMatlabHostTarget()
                tf = true;
            else
                error('adder library not available for this target');
            end
        end

        function updateBuildInfo(buildInfo, ctx)
            [~, linkLibExt, execLibExt, ~] = ctx.getStdLibInfo();

            % Header files
            hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
            buildInfo.addIncludePaths(hdrFilePath);

            % Link files
            linkFiles = strcat('adder', linkLibExt);
            linkPath = hdrFilePath;
            linkPriority = '';
            linkPrecompiled = true;
            linkLinkOnly = true;
            group = '';
            buildInfo.addLinkObjects(linkFiles, linkPath, ...
                linkPriority, linkPrecompiled, linkLinkOnly, group);

            % Non-build files
            nbFiles = 'adder';
            nbFiles = strcat(nbFiles, execLibExt);
            buildInfo.addNonBuildFiles(nbFiles,'','');
        end

        %API for library function 'adder'
        function c = adder(a, b)
            if coder.target('MATLAB')
                % running in MATLAB, use built-in addition
                c = a + b;
            else
                % running in generated code, call library function
                coder.cinclude('adder.h');

                % Because MATLAB Coder generated adder, use the
                % housekeeping functions before and after calling
                % adder with coder.ceval.
                % Call initialize function before calling adder for the
                % first time.

                coder.ceval('adder_initialize');
                c = 0;
                c = coder.ceval('adder', a, b);


                % Call the terminate function after
                % calling adder for the last time.

                coder.ceval('adder_terminate');
            end
```

```
        end
    end
end
```

Write a function `adder_main` that calls the external library function `adder`.

```
function y = adder_main(x1, x2)
%#codegen
    y = AdderAPI.adder(x1, x2);
end
```

Generate a MEX function for `adder_main`. The MEX Function exercises the `coder.ExternalDependency` methods.

```
codegen('adder_main', '-args', {7,9}, '-report')
```

Copy the library to the current folder using the file extension for your platform.

For Windows, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dll'));
```

For Linux, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.so'));
```

For Mac, use:

```
copyfile(fullfile(pwd, 'codegen', 'dll', 'adder', 'adder.dylib'));
```

Run the MEX function and verify the result.

```
adder_main_mex(2,3)
```

See Also

coder.BuildConfig | coder.ExternalDependency | `error`

More About

- "Encapsulating the Interface to External Code" on page 27-3
- "Build Information Object" on page 20-141
- "Build Information Methods" on page 20-141

Update Build Information from MATLAB code

You can control aspects of the build process that occur after code generation but before compilation. For example, you can specify compiler or linker options.

To customize the build from your MATLAB code:

1. In your MATLAB code, call `coder.updateBuildInfo` to update the build information object. You specify a build information object method and the input arguments for the method.
2. Generate code from your MATLAB code.

See Also

`coder.updateBuildInfo`

Call External Functions Encapsulated by coder.ExternalDependency

When a method of a class derived from `coder.ExternalDependency` defines the interface to an external function, you call the external function by calling the method.

Suppose you define the following method for a class named `AdderAPI`:

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

This method defines the interface to a function `adder` which has two inputs, `a` and `b`. In your MATLAB code, call `adder` this way:

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

See Also

coder.ExternalDependency

Related Examples

- “Encapsulate Interface to an External C Library” on page 27-6

More About

- “Encapsulating the Interface to External Code” on page 27-3

Generate Efficient and Reusable Code

Optimization Strategies

MATLAB Coder introduces certain optimizations when generating C/C++ code or MEX functions from your MATLAB code. For more information, see "MATLAB Coder Optimizations in Generated Code" on page 28-47.

To optimize your generated code further, you can:

- Adapt your MATLAB code.
- Control code generation using the configuration object from the command-line or the project settings dialog box.

To optimize the execution speed of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have `for`-loops whose iterations are independent of each other.	"Generate Code with Parallel for-Loops (parfor)" on page 28-33
You have variable-size arrays in your MATLAB code.	"Minimize Dynamic Memory Allocation" on page 28-20
You have multiple variable-size arrays in your MATLAB code. You want dynamic memory allocation for larger arrays and static allocation for smaller ones.	"Set Dynamic Memory Allocation Threshold" on page 28-28
You want your generated function to be called by reference.	"Eliminate Redundant Copies of Function Inputs" on page 28-7
You are calling small functions in your MATLAB code.	"Inline Code" on page 28-10
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	"Control Inlining" on page 28-12
You do not want to generate code for expressions that contain constants only.	"Fold Function Calls into Constants" on page 28-15
You have loop operations in your MATLAB code that do not depend on the loop index.	"Minimize Redundant Operations in Loops" on page 28-35

Condition	Action
You have integer operations in your MATLAB code. You know beforehand that integer overflow does not occur during execution of your generated code.	"Disable Support for Integer Overflow" on page 28-39
You know beforehand that `Inf`s and `NaN`s do not occur during execution of your generated code.	"Disable Support for Non-Finite Numbers" on page 28-40
You have `for`-loops with few iterations.	"Unroll `for`-Loops" on page 28-37
You already have legacy C/C++ code optimized for your target environment.	"Integrate External/Custom Code" on page 28-41
You want to speed up the code generated for linear algebra functions.	"Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" on page 28-59

To optimize the memory usage of generated code, for these conditions, perform the following actions as necessary:

Condition	Action
You have `if/else/elseif` statements or `switch/case/otherwise` statements in your MATLAB code. You do not require some branches of the statements in your generated code.	"Prevent Code Generation for Unused Execution Paths" on page 28-31
You want your generated function to be called by reference.	"Eliminate Redundant Copies of Function Inputs" on page 28-7
You have limited stack space for your generated code.	"Control Stack Space Usage" on page 28-17
You are calling small functions in your MATLAB code.	"Inline Code" on page 28-10
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	"Control Inlining" on page 28-12
You do not want to generate code for expressions that contain constants only.	"Fold Function Calls into Constants" on page 28-15

Condition	Action
You have loop operations in your MATLAB code that do not depend on the loop index.	“Minimize Redundant Operations in Loops” on page 28-35
You have integer operations in your MATLAB code. You know beforehand that integer overflow does not occur during execution of your generated code.	“Disable Support for Integer Overflow” on page 28-39
You know beforehand that `Inf`-s and `NaN`-s does not occur during execution of your generated code.	“Disable Support for Non-Finite Numbers” on page 28-40
Your MATLAB code has variables that are large arrays or structures. Your variables are not reused in the generated code. They are preserved. You want to see if the extra memory required to preserve the variable names of the large arrays or structures affects performance.	“Reuse Large Arrays and Structures” on page 28-56

Modularize MATLAB Code

For large MATLAB code, streamline code generation by modularizing the code:

1. Break up your MATLAB code into smaller, self-contained sections.
2. Save each section in a MATLAB function.
3. Generate C/C++ code for each function.
4. Call the generated C/C++ functions in sequence from a wrapper MATLAB function using `coder.ceval`.
5. Generate C/C++ code for the wrapper function.

Besides streamlining code generation for the original MATLAB code, this approach also supplies you with C/C++ code for the individual sections. You can reuse the code for the individual sections later by integrating them with other generated C/C++ code using `coder.ceval`.

Eliminate Redundant Copies of Function Inputs

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

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

This coding practice uses a reference parameter optimization. When a variable acts as both input and output, the generated code passes the variable by reference instead of redundantly copying the input to a temporary variable. In the preceding example, input `A` is passed by reference in the generated code because it also acts as an output for function `foo`:

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

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

By contrast, suppose that you rewrite function `foo` without the optimization:

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

The generated code passes the inputs by value and returns the value of the output:

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

In some cases, the output of the function cannot be a modified version of its inputs. If you do not use the inputs later in the function, you can modify your code to operate on the inputs instead of on a copy of the inputs. One method is to create additional return values for the function. For example, consider the code:

```
function y1=foo(u1) %#codegen
  x1=u1+1;
  y1=bar(x1);
end

function y2=bar(u2)
  % Since foo does not use x1 later in the function,
  % it would be optimal to do this operation in place
  x2=u2.*2;
  % The change in dimensions in the following code
  % means that it cannot be done in place
  y2=[x2,x2];
end
```

You can modify this code to eliminate redundant copies.

```
function y1=foo(u1) %#codegen
  u1=u1+1;
  [y1, u1]=bar(u1);
end

function [y2, u2]=bar(u2)
    u2=u2.*2;
  % The change in dimensions in the following code
  % still means that it cannot be done in place
  y2=[u2,u2];
end
```

The reference parameter optimization does not apply to constant inputs. If the same variable is an input and an output, and the input is constant, the code generator treats the output as a separate variable. For example, consider the function `foo`:

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

Generate code in which `A` has a constant value `2`.

```
codegen -config:lib foo -args {coder.Constant(2) 3} -report
```

The generated code defines the constant A and returns the value of the output.

```
...
#define A                              (2.0)
...
double foo(double B)
{
  return A * B;
}
...
```

Related Examples

- “Pass Structure Arguments by Reference or by Value in Generated Code” on page 20-180

Inline Code

Inlining is a technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/ C++ code. Inlining can create opportunities for further optimization of the generated C/ C++ code. The code generator uses internal heuristics to determine whether to inline functions in the generated code. You can use the `coder.inline` directive to fine-tune these heuristics for individual functions. For more information, see `coder.inline`.

In this section...
"Prevent Function Inlining" on page 28-10
"Use Inlining in Control Flow Statements" on page 28-10

Prevent Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
  coder.inline('never');
  y = x;
end
```

Use Inlining in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose you want to generate code for a division function that will be embedded in a system with limited memory. To optimize memory use in the generated code, the following function, `inline_division`, manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
   coder.inline('always');
else
```

```
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
   coder.inline('never');
end

if any(divisor == 0)
   error('Can not divide by 0');
end

y = dividend / divisor;
```

Related Examples

- “Control Inlining” on page 28-12

Control Inlining

Restrict inlining when:

- The size of generated code exceeds desired limits due to excessive inlining of functions. Suppose that you include the statement, `coder.inline('always')`, inside a certain function. You then call that function at many different sites in your code. The generated code can be large due to the function being inlined every time it is called.

  The call sites must be different. For instance, inlining does not lead to large code if the function to be inlined is called several times inside a loop.
- You have limited RAM or stack space.

In this section...

Control Size of Functions Inlined

You can use the MATLAB Coder app or the command-line interface to control the maximum size of functions that can be inlined. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThreshold`, to the maximum size that you want.

  ```
  cfg = coder.config('lib');
  cfg.InlineThreshold = 100;
  ```

  Generate code using this configuration object.

Control Size of Functions After Inlining

You can use the MATLAB Coder app or the command-line interface to control the maximum size of functions after inlining. The function size is measured in terms of an abstract number of instructions, not actual MATLAB instructions or instructions in the target processor. Experiment with this parameter to obtain the inlining behavior that you want.

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the value of the field, **Inline threshold max**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

  ```
  cfg = coder.config('lib');
  cfg.InlineThresholdMax = 100;
  ```

  Generate code using this configuration object.

Control Stack Size Limit on Inlined Functions

Specifying a limit on the stack space constrains the amount of inlining allowed. For out-of-line functions, stack space for variables local to the function is released when the function returns. However, for inlined functions, stack space remains occupied by the local variables even after the function is executed. The value of the property, `InlineStackLimit`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that a certain value of `InlineStackLimit` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

You can use the MATLAB Coder app or the command-line interface to control the stack size limit on inlined functions.

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the value of the field, **Inline stack limit**, to the maximum size that you want.
- At the command line, create a `codegen` configuration object. Set the value of the property, `InlineThresholdMax`, to the maximum size that you want.

  ```
  cfg = coder.config('lib');
  cfg.InlineStackLimit = 2000;
  ```

  Generate code using this configuration object.

Related Examples

- "Inline Code" on page 28-10

Fold Function Calls into Constants

This example shows how to specify constants in generated code using `coder.const`. The code generator folds an expression or a function call in a `coder.const` statement into a constant in generated code. Because the generated code does not have to evaluate the expression or call the function every time, this optimization reduces the execution time of the generated code.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator produces code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int k;
  for (k = 0; k < 10; k++) {
    y[k] = (double)((1 + k) * (1 + k)) + Shift;
  }
}
```

Replace the statement

```
y = (1:10).^2+Shift;
```

with

```
y = coder.const((1:10).^2)+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int i0;
  static const signed char iv0[10] = { 1, 4, 9, 16, 25, 36,
                                       49, 64, 81, 100 };

  for (i0 = 0; i0 < 10; i0++) {
    y[i0] = (double)iv0[i0] + Shift;
  }
}
```

See Also

`coder.const`

Control Stack Space Usage

This example shows how to set the maximum stack space that the generated code uses. Set the maximum stack usage when:

- You have limited stack space, for instance, in embedded targets.
- Your C compiler reports a run-time stack overflow.

The value of the property, `StackUsageMax`, is measured in bytes. Based on information from the target hardware settings, the software estimates the number of stack variables that a certain value of `StackUsageMax` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Control Stack Space Usage Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
3. Click **More Settings**.
4. On the **Memory** tab, set **Stack usage max** to the value that you want.

Control Stack Space Usage at the Command Line

1. Create a configuration object for code generation.

   Use `coder.config` with arguments `'lib'`,`'dll'`, or `'exe'` (depending on your requirements). For example:

   ```
   cfg = coder.config('lib');
   ```
2. Set the property, `StackUsageMax`, to the value that you want.

   ```
   cfg.StackUsageMax=400000;
   ```

More About

- “Stack Allocation and Performance” on page 28-18

Stack Allocation and Performance

By default, local variables are allocated on the stack. Large variables that do not fit on the stack are statically allocated in memory.

Stack allocation typically uses memory more efficiently than static allocation. However, stack space is sometimes limited, typically in embedded processors. MATLAB Coder allows you to manually set a limit on the stack space usage to make your generated code suitable for your target hardware. You can choose this limit based on the target hardware configurations. For more information, see “Control Stack Space Usage” on page 28-17.

For limited stack space, you can choose to allocate large variables on the heap instead of using static allocation. Heap allocation is slower but more memory-efficient than static allocation. To allocate large variables on the heap, do one of the following:

Allocate Heap Space from Command Line

1. Create a configuration object. Set the property, `MultiInstanceCode`, to `true`.

   ```
   cfg = coder.config('exe');
   cfg.MultiInstanceCode = true;
   ```
2. Generate code using this configuration object.

Allocate Heap Space Using the MATLAB Coder App

1. Using the MATLAB Coder app, in the project settings dialog box, on the **Memory** tab, select the **Generate re-entrant code** check box.

- Generate code.

Dynamic Memory Allocation and Performance

To achieve faster execution of generated code, minimize dynamic (or run-time) memory allocation of arrays.

MATLAB Coder does not provide a size for unbounded arrays in generated code. Instead, such arrays are referenced indirectly through pointers. For such arrays, memory cannot be allocated during compilation of generated code. Based on storage requirements for the arrays, memory is allocated and freed at run time as required. This run-time allocation and freeing of memory leads to slower execution of the generated code.

When Dynamic Memory Allocation Occurs

Dynamic memory allocation occurs when the code generator cannot find upper bounds for variable-size arrays. The software cannot find upper bounds when you specify the size of an array using a variable that is not a compile-time constant. An example of such a variable is an input variable (or a variable computed from an input variable).

Instances in the MATLAB code that can lead to dynamic memory allocation are:

- Array initialization: You specify array size using a variable whose value is known only at run time.
- After initialization of an array:
  - You declare the array as variable-size using `coder.varsize` without explicit upper bounds. After this declaration, you expand the array by concatenation inside a loop. The number of loop runs is known only at run time.
  - You use a `reshape` function on the array. At least one of the size arguments to the `reshape` function is known only at run time.

If you know the maximum size of the array, you can avoid dynamic memory allocation. You can then provide an upper bound for the array and prevent dynamic memory allocation in generated code. For more information, see “Minimize Dynamic Memory Allocation” on page 28-20.

Minimize Dynamic Memory Allocation

When possible, minimize dynamic memory allocation because it leads to slower execution of generated code. Dynamic memory allocation occurs when the code generator cannot find upper bounds for variable-size arrays.

If you know the maximum size of a variable-size array, you can avoid dynamic memory allocation. Follow these steps:

1. "Provide Maximum Size for Variable-Size Arrays" on page 28-21.
2. Depending on your requirements, do one of the following:

   - "Disable Dynamic Memory Allocation During Code Generation" on page 28-27.
   - "Set Dynamic Memory Allocation Threshold" on page 28-28

Caution: If a variable-size array in the MATLAB code does not have a maximum size, disabling dynamic memory allocation leads to a code generation error. Before disabling dynamic memory allocation, you must provide a maximum size for variable-size arrays in your MATLAB code.

More About

- "Dynamic Memory Allocation and Performance" on page 28-19

Provide Maximum Size for Variable-Size Arrays

To constrain array size for variable-size arrays, do one of the following:

- **Constrain Array Size Using assert Statements**

  If the variable specifying array size is not a compile-time constant, use an `assert` statement with relational operators to constrain the variable. Doing so helps the code generator to determine a maximum size for the array.

  The following examples constrain array size using `assert` statements:

  - **When Array Size Is Specified by Input Variables**

    Define a function `array_init` which initializes an array `y` with input variable `N`:

    ```
    function y = array_init (N)
      assert(N <= 25); % Generates exception if N > 25
      y = zeros(1,N);
    ```

    The `assert` statement constrains input `N` to a maximum size of 25. In the absence of the `assert` statement, `y` is assigned a pointer to an array in the generated code, thus allowing dynamic memory allocation.

  - **When Array Size Is Obtained from Computation Using Input Variables**

    Define a function, `array_init_from_prod`, which takes two input variables, `M` and `N`, and uses their product to specify the maximum size of an array, `y`.

    ```
    function y = array_init_from_prod (M,N)
       size=M*N;
       assert(size <= 25); % Generates exception if size > 25
       y=zeros(1,size);
    ```

    The `assert` statement constrains the product of `M` and `N` to a maximum of 25.

    Alternatively, if you restrict `M` and `N` individually, it leads to dynamic memory allocation:

    ```
    function y = array_init_from_prod (M,N)
       assert(M <= 5);
       assert(N <= 5);
       size=M*N;
       y=zeros(1,size);
    ```

This code causes dynamic memory allocation because `M` and `N` can both have unbounded negative values. Therefore, their product can be unbounded and positive even though, individually, their positive values are bounded.

Tip: Place the assert statement on a variable immediately before it is used to specify array size.

Tip: You can use `assert` statements to restrict array sizes in most cases. When expanding an array inside a loop, this strategy does not work if the number of loop runs is known only at run time.

- **Restrict Concatenations in a Loop Using coder.varsize with Upper Bounds**

  You can expand arrays beyond their initial size by concatenation. When you concatenate additional elements inside a loop, there are two syntax rules for expanding arrays.

  1. **Array size during initialization is not a compile-time constant**

     If the size of an array during initialization is not a compile-time constant, you can expand it by concatenating additional elements:

     ```
     function out=ExpandArray(in) % Expand an array by five elements
       out = zeros(1,in);
       for i=1:5
          out = [out 0];
       end
     ```

  2. **Array size during initialization is a compile-time constant**

     Before concatenating elements, you have to declare the array as variable-size using `coder.varsize`:

     ```
     function out=ExpandArray() % Expand an array by five elements
       out = zeros(1,5);
       coder.varsize('out');
       for i=1:5
          out = [out 0];
       end
     ```

Either case leads to dynamic memory allocation. To prevent dynamic memory allocation in such cases, use `coder.varsize` with explicit upper bounds. This example shows how to use `coder.varsize` with explicit upper bounds:

Restrict Concatenations Using coder.varsize with Upper Bounds

1 Define a function, `RunningAverage`, that calculates the running average of an `N`-element subset of an array:

```
  function avg=RunningAverage(N)

% Array whose elements are to be averaged
  NumArray=[1 6 8 2 5 3];

% Initialize average:
% These will also be the first two elements of the function output
  avg=[0 0];

% Place a bound on the argument
  coder.varsize('avg',[1 8]);

% Loop to calculate running average
  for i=1:N
    s=0;
    s=s+sum(NumArray(1:i));
    avg=[avg s/i];
  % Increase the size of avg as required by concatenation
  end
```

The output, `avg`, is an array that you can expand as required to accommodate the running averages. As a new running average is calculated, it is added to the array `avg` through concatenation, thereby expanding the array.

Because the maximum number of running averages is equal to the number of elements in `NumArray`, you can supply an explicit upper bound for `avg` in the `coder.varsize` statement. In this example, the upper bound is 8 (the two initial elements plus the six elements of `NumArray`).

2 Generate code for `RunningAverage` with input argument of type `double`:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, avg is assigned an array of size 8 (static memory allocation). The function definition for RunningAverage appears as follows (using built-in C types):

```
void RunningAverage (double N, double avg_data[8], int avg_size[2])
```

3 By contrast, if you remove the explicit upper bound, the generated code dynamically allocates avg.

Replace the statement

```
coder.varsize('avg',[1 8]);
```

with:

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

4 Generate code for RunningAverage with input argument of type double:

```
codegen -config:lib -report RunningAverage -args 2
```

In the generated code, avg is assigned a pointer to an array, thereby allowing dynamic memory allocation. The function definition for RunningAverage appears as follows (using built-in C types):

```
void Test(double N, emxArray_real_T *avg)
```

Note: Dynamic memory allocation also occurs if you precede coder.varsize('avg') with the following assert statement:

```
     assert(N < 6);
```

The assert statement does not restrict the number of concatenations within the loop.

- **Constrain Array Size When Rearranging a Matrix**

  The statement out = reshape(in,m,n,...) takes an array, in, as an argument and returns array, out, having the same elements as in, but reshaped as an m-by-n-by-... matrix. If one of the size variables m,n,.... is not a compile-time constant, then dynamic memory allocation of out takes place.

To avoid dynamic memory allocation, use an assert statement before the reshape statement to restrict the size variables m,n,... to numel(in). This example shows how to use an assert statement before a reshape statement:

Rearrange a Matrix into Given Number of Rows

1 Define a function, ReshapeMatrix, which takes an input variable, N, and reshapes a matrix, mat, to have N rows:

```
  function [out1,out2] = ReshapeMatrix(N)

  mat = [1 2 3 4 5; 4 5 6 7 8]
% Since mat has 10 elements, N must be a factor of 10
% to pass as argument to reshape

  out1 = reshape(mat,N,[]);
% N is not restricted

  assert(N < numel(mat));
% N is restricted to number of elements in mat
  out2 = reshape(mat,N,[]);
```

2 Generate code for ReshapeArray using the codegen command (the input argument does not have to be a factor of 10):

```
codegen -config:lib -report ReshapeArray -args 3
```

While out1 is dynamically allocated, out2 is assigned an array with size 100 (=10 X 10) in the generated code.

Tip: If your system has limited memory, do not use the assert statement in this way. For an n-element matrix, the assert statement creates an n-by-n matrix, which might be large.

Related Examples

- on page 28-20
- on page 28-27
- on page 28-28

More About

- on page 28-19

Disable Dynamic Memory Allocation During Code Generation

To disable dynamic memory allocation using the MATLAB Coder app:

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Memory** tab, under **Variable Sizing Support**, set **Dynamic memory allocation** to `Never`.

To disable dynamic memory allocation at the command line:

1. In the MATLAB workspace, define the configuration object:

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

2. Set the `DynamicMemoryAllocation` property of the configuration object to `Off`:

   ```
   cfg.DynamicMemoryAllocation = 'Off';
   ```

If a variable-size array in the MATLAB code does not have a maximum upper bound, disabling dynamic memory allocation leads to a code generation error. Therefore, you can identify variable-size arrays in your MATLAB code that do not have a maximum upper bound. These arrays are the arrays that are dynamically allocated in the generated code.

Related Examples

- "Minimize Dynamic Memory Allocation" on page 28-20
- "Provide Maximum Size for Variable-Size Arrays" on page 28-21
- "Set Dynamic Memory Allocation Threshold" on page 28-28

More About

- "Dynamic Memory Allocation and Performance" on page 28-19

Set Dynamic Memory Allocation Threshold

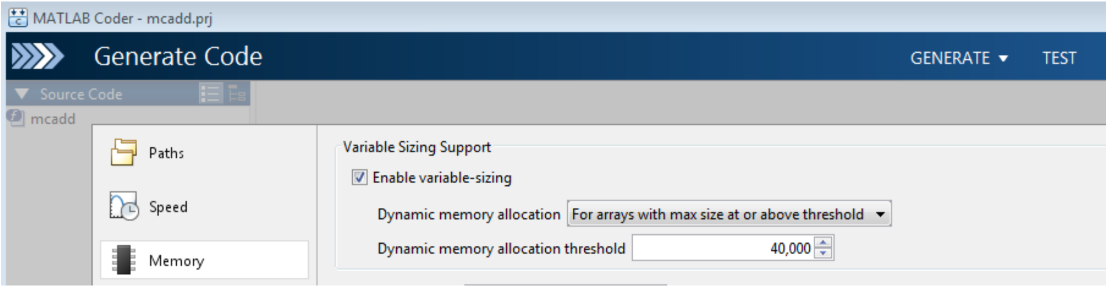
This example shows how to specify a dynamic memory allocation threshold for variable-size arrays. Dynamic memory allocation optimizes storage requirements for variable-size arrays, but causes slower execution of generated code. Instead of disabling dynamic memory allocation for all variable-size arrays, you can disable dynamic memory allocation for arrays less than a certain size.

Specify this threshold when you want to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, static memory allocation can speed up generated code. Static memory allocation can lead to unused storage space. However, you can decide that the unused storage space is not a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, when you use dynamic memory allocation, you can significantly reduce storage requirements.

Set Dynamic Memory Allocation Threshold Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Click **More Settings**.
3. On the **Memory** tab, select the **Enable variable-sizing** check box.
4. Set **Dynamic memory allocation** to `For arrays with max size at or above threshold`.
5. Set **Dynamic memory allocation threshold** to the value that you want.



Variable Sizing Support

☑ Enable variable-sizing

Dynamic memory allocation: For arrays with max size at or above threshold

Dynamic memory allocation threshold: 40,000

The **Dynamic memory allocation threshold** value is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that a certain value of `DynamicMemoryAllocationThreshold` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Set Dynamic Memory Allocation Threshold at the Command Line

1. Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`,`'dll'`, or `'exe'` (depending on your requirements). For example:

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

2. Set `DynamicMemoryAllocation` to `'Threshold'`.

   ```
   cfg.DynamicMemoryAllocation='Threshold';
   ```

3. Set the property, `DynamicMemoryAllocationThreshold`, to the value that you want.

   ```
   cfg.DynamicMemoryAllocationThreshold = 40000;
   ```

   The value stored in `DynamicMemoryAllocationThreshold` is measured in bytes. Based on information from the target hardware settings, the software estimates the size of the array that a certain value of `DynamicMemoryAllocationThreshold` can accommodate. This estimate excludes possible C compiler optimizations such as putting variables in registers.

Related Examples

- “Minimize Dynamic Memory Allocation” on page 28-20
- “Provide Maximum Size for Variable-Size Arrays” on page 28-21
- “Disable Dynamic Memory Allocation During Code Generation” on page 28-27

More About

- “Dynamic Memory Allocation and Performance” on page 28-19

Excluding Unused Paths from Generated Code

In certain situations, you do not need some branches of an: `if, elseif, else` statement, or a `switch, case, otherwise` statement in your generated code. For instance:

- You have a MATLAB function that performs multiple tasks determined by a control-flow variable. You might not need some of the tasks in the code generated from this function.
- You have an `if/elseif/if` statement in a MATLAB function performing different tasks based on the nature (type/value) of the input. In some cases, you know the nature of the input beforehand. If so, you do not need some branches of the `if` statement.

You can prevent code generation for the unused branches of an `if/elseif/else` statement or a `switch/case/otherwise` statement. Declare the control-flow variable as a constant. The code generator produces code only for the branch that the control-flow variable chooses.

Related Examples

- "Prevent Code Generation for Unused Execution Paths" on page 28-31

Prevent Code Generation for Unused Execution Paths

In this section...

If a variable controls the flow of an: `if, elseif, else` statement, or a `switch, case, otherwise` statement, declare it as constant so that code generation takes place for one branch of the statement only.

Depending on the nature of the control-flow variable, you can declare it as constant in two ways:

- If the variable is local to the MATLAB function, assign it to a constant value in the MATLAB code. For an example, see "Prevent Code Generation When Local Variable Controls Flow" on page 28-31.
- If the variable is an input to the MATLAB function, you can declare it as constant using coder.Constant. For an example, see "Prevent Code Generation When Input Variable Controls Flow" on page 28-32.

Prevent Code Generation When Local Variable Controls Flow

1 Define a function `SquareOrCube` which takes an input variable, `in`, and squares or cubes its elements based on whether the choice variable, `ch`, is set to `s` or `c`:

```
function out = SquareOrCube(ch,in) %#codegen
 if ch=='s'
     out = in.^2;
 elseif ch=='c'
     out = in.^3;
 else
     out = 0;
 end
```

2 Generate code for `SquareOrCube` using the `codegen` command:

```
codegen -config:lib SquareOrCube -args {'s',zeros(2,2)}
```

The generated C code squares or cubes the elements of a 2-by-2 matrix based on the input for `ch`.

3 Add the following line to the definition of SquareOrCube:

```
ch = 's';
```

The generated C code squares the elements of a 2-by-2 matrix. The choice variable, ch, and the other branches of the if/elseif/if statement do not appear in the generated code.

Prevent Code Generation When Input Variable Controls Flow

1 Define a function MathFunc, which performs different mathematical operations on an input, in, depending on the value of the input, flag:

```
function out = MathFunc(flag,in) %#codegen
  %# codegen
   switch flag
     case 1
        out=sin(in);
     case 2
        out=cos(in);
     otherwise
        out=sqrt(in);
   end
```

2 Generate code for MathFunc using the codegen command:

```
codegen -config:lib MathFunc -args {1,zeros(2,2)}
```

The generated C code performs different math operations on the elements of a 2-by-2 matrix based on the input for flag.

3 Generate code for MathFunc, declaring the argument, flag, as a constant using coder.Constant:

```
codegen -config:lib MathFunc -args {coder.Constant(1),zeros(2,2)}
```

The generated C code finds the sine of the elements of a 2-by-2 matrix. The variable, flag, and the switch/case/otherwise statement do not appear in the generated code.

More About

- "Excluding Unused Paths from Generated Code" on page 28-30

Generate Code with Parallel for-Loops (parfor)

This example shows how to generate C code for a MATLAB algorithm that contains a `parfor`-loop.

1. Write a MATLAB function that contains a `parfor`-loop. For example:

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
  a(i,:)=real(fft(r(i,:)));
end
```

2. Generate C code for `test_parfor`. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor
```

   Because you did not specify the maximum number of threads to use, the generated C code executes the loop iterations in parallel on the available number of cores.

3. To specify a maximum number of threads, rewrite the function `test_parfor` as follows:

```
function a = test_parfor(u) %#codegen
a=ones(10,256);
r=rand(10,256);
parfor (i=1:10,u)
  a(i,:)=real(fft(r(i,:)));
end
```

4. Generate C code for `test_parfor`. Use `-args 0` to specify that the input, `u`, is a scalar double. At the MATLAB command line, enter:

```
codegen -config:lib test_parfor -args 0
```

   In the generated code, the iterations of the `parfor`-loop run on at most the number of cores specified by the input, `u`. If less than `u` cores are available, the iterations run on the cores available at the time of the call.

More About

- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 25-18
- “Classification of Variables in `parfor`-Loops” on page 25-26

- “Reduction Assignments in parfor-Loops” on page 25-25

Minimize Redundant Operations in Loops

This example shows how to minimize redundant operations in loops. When a loop operation does not depend on the loop index, performing it inside a loop is redundant. This redundancy often goes unnoticed when you are performing multiple operations in a single MATLAB statement inside a loop. For example, in the following code, the inverse of the matrix `B` is being calculated 100 times inside the loop although it does not depend on the loop index:

```
for i=1:100
     C=C + inv(B)*A^i*B;
  end
```

Performing such redundant loop operations can lead to unnecessary processing. To avoid unnecessary processing, move operations outside loops as long as they do not depend on the loop index.

1. Define a function, `SeriesFunc(A,B,n)`, that calculates the sum of `n` terms in the following power series expansion:

   $$C = 1 + B^{-1}AB + B^{-1}A^{2}B + \ldots$$

   ```
     function C=SeriesFunc(A,B,n)

   % Initialize C with a matrix having same dimensions as A
     C=zeros(size(A));

   % Perform the series sum
     for i=1:n
        C=C+inv(B)*A^i*B;
     end
   ```

2. Generate code for `SeriesFunc` with 4-by-4 matrices passed as input arguments for `A` and `B`:

   ```
   X = coder.typeof(zeros(4));
   codegen -config:lib -launchreport SeriesFunc -args {X,X,10}
   ```

   In the generated code, the inversion of `B` is performed `n` times inside the loop. It is more economical to perform the inversion operation once outside the loop because it does not depend on the loop index.

3. Modify `SeriesFunc` as follows:

```
  function C=SeriesFunc(A,B,n)

% Initialize C with a matrix having same dimensions as A
  C=zeros(size(A));

% Perform the inversion outside the loop
  inv_B=inv(B);

% Perform the series sum
  for i=1:n
     C=C+inv_B*A^i*B;
  end
```

This procedure performs the inversion of B only once, leading to faster execution of the generated code.

Unroll for-Loops

When the code generator unrolls a for-loop, instead of producing a for-loop in the generated code, it produces a copy of the loop body for each iteration. For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can significantly increase code generation time and generate inefficient code.

The code generator uses heuristics to determine when to unroll a for-loop. To force loop unrolling, use coder.unroll. For example:

```
function z = call_myloop()
%#codegen
z = myloop(5);
end

function b = myloop(n)
b = zeros(1,n);
coder.unroll();
for i = 1:n
    b(i)=i+n;
end
end
```

Here is the generated code for the for-loop:

```
  z[0] = 6.0;
  z[1] = 7.0;
  z[2] = 8.0;
  z[3] = 9.0;
  z[4] = 10.0;
```

To control when a for-loop is unrolled, use the coder.unroll flag argument. For example, unroll the loop only when the number of iterations is less than 10.

```
function z = call_myloop()
%#codegen
z = myloop(5);
end

function b = myloop(n)
unroll_flag = n < 10;
b = zeros(1,n);
coder.unroll(unroll_flag);
for i = 1:n
```

```
    b(i)=i+n;
end
end
```

To unroll a `for`-loop, the code generator must be able to determine the bounds of the `for`-loop. For example, code generation fails for the following code because the value of `n` is not known at code generation time.

```
function b = myloop(n)
b = zeros(1,n);
coder.unroll();
for i = 1:n
    b(i)=i+n;
end
end
```

See Also

`coder.unroll`

More About

- “Nonconstant Index into vargin or vargout in a for-Loop” on page 30-16

Disable Support for Integer Overflow or Non-Finites

The code generator produces supporting code for the following situations:

- The result of an integer operation falls outside the range that a data type can represent. This situation is known as integer overflow.
- An operation generates non-finite values (`inf` and `NaN`). The supporting code is contained in the files `rt_nonfinite.c`, `rtGetInf.c`, and `rtGetNaN.c` (with corresponding header files).

If you know that these situations do not occur, you can suppress generation of the supporting code. You therefore reduce the size of the generated code and increase its speed. However, if one of these situations occurs, it is possible that the generated code does not match the behavior of the original MATLAB code.

Disable Support for Integer Overflow

You can use the MATLAB Coder app or the command-line interface to disable support for integer overflow. When you disable this support, the overflow behavior of your generated code depends on your target C compiler. Most C compilers wrap on overflow.

- Using the app:

  1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
  2. Click **More Settings**.
  3. On the **Speed** tab, clear `Saturate on integer overflow`.

- At the command line:

  1. Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'`, or `'exe'` (depending on your requirements). For example:

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

  2. Set the `SaturateOnIntegerOverflow` property to `false`.

     ```
     cfg.SaturateOnIntegerOverflow = false;
     ```

Disable Support for Non-Finite Numbers

You can use the MATLAB Coder app or the command-line interface to disable support for non-finite numbers(`inf` and `NaN`).

- Using the app:

  1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
  2. Set **Build type** to `Source Code`, `Static Library`, `Dynamic Library`, or `Executable` (depending on your requirements).
  3. Click **More Settings**.
  4. On the **Speed** tab, clear the **Support non-finite numbers** check box.

- At the command line:

  1. Create a configuration object for code generation. Use `coder.config` with arguments `'lib'`, `'dll'`, or `'exe'` (depending on your requirements). For example:

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

  2. Set the `SupportNonFinite` property to `false`.

     ```
     cfg.SupportNonFinite = false;
     ```

Integrate External/Custom Code

This example shows how to integrate external or custom code to enhance performance of generated code. Although MATLAB Coder generates optimized code for most applications, you might have custom code optimized for your specific requirements. For example:

- You have custom libraries optimized for your target environment.
- You have custom libraries for functions not supported by MATLAB Coder.
- You have custom libraries that meet standards set by your company.

In such cases, you can integrate your custom code with the code generated by MATLAB Coder.

This example illustrates how to integrate the function `cublasSgemm` from the NVIDIA® CUDA® Basic Linear Algebra Subroutines (CUBLAS) library in generated code. This function performs matrix multiplication on a Graphics Processing Unit (GPU).

1 Define a class `ExternalLib_API` that derives from the class `coder.ExternalDependency`. `ExternalLib_API` defines an interface to the `CUBLAS` library through the following methods:

   - `getDescriptiveName`: Returns a descriptive name for `ExternalLib_API` to be used for error messages.
   - `isSupportedContext`: Determines if the build context supports the `CUBLAS` library.
   - `updateBuildInfo`: Adds header file paths and link files to the build information.
   - `GPU_MatrixMultiply`: Defines the interface to the `CUBLAS` library function `cublasSgemm`.

   ExternalLib_API.m

```
classdef ExternalLib_API < coder.ExternalDependency
    %#codegen

    methods (Static)

        function bName = getDescriptiveName(~)
            bName = 'ExternalLib_API';
```

```
end

function tf = isSupportedContext(ctx)
    if  ctx.isMatlabHostTarget()
        tf = true;
    else
        error('CUBLAS library not available for this target');
    end
end

function updateBuildInfo(buildInfo, ctx)
    [~, linkLibExt, ~, ~] = ctx.getStdLibInfo();

    % Include header file path
    % Include header files later using coder.cinclude
    hdrFilePath = 'C:\My_Includes';
    buildInfo.addIncludePaths(hdrFilePath);

    % Include link files
    linkFiles = strcat('libcublas', linkLibExt);
    linkPath = 'C:\My_Libs';
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

    linkFiles = strcat('libcudart', linkLibExt);
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
        linkPriority, linkPrecompiled, linkLinkOnly, group);

end

%API for library function 'cuda_MatrixMultiply'
function C = GPU_MatrixMultiply(A, B)
    assert(isa(A,'single'), 'A must be single.');
    assert(isa(B,'single'), 'B must be single.');

    if(coder.target('MATLAB'))
        C=A*B;
    else

        % Include header files
        %     for external functions and typedefs
        % Header path included earlier using updateBuildInfo
        coder.cinclude('"cuda_runtime.h"');
        coder.cinclude('"cublas_v2.h"');

        % Compute dimensions of input matrices
        m = int32(size(A, 1));
        k = int32(size(A, 2));
        n = int32(size(B, 2));

        % Declare pointers to matrices on destination GPU
        d_A = coder.opaque('float*');
        d_B = coder.opaque('float*');
        d_C = coder.opaque('float*');

        % Compute memory to be allocated for matrices
        % Single = 4 bytes
        size_A = m*k*4;
        size_B = k*n*4;
        size_C = m*n*4;

        % Define error variables
        error = coder.opaque('cudaError_t');
        cudaSuccessV = coder.opaque('cudaError_t', ...
            'cudaSuccess');

        % Assign memory on destination GPU
        error = coder.ceval('cudaMalloc', ...
            coder.wref(d_A), size_A);
        assert(error == cudaSuccessV, ...
            'cudaMalloc(A) failed');
        error = coder.ceval('cudaMalloc', ...
            coder.wref(d_B), size_B);
        assert(error == cudaSuccessV, ...
            'cudaMalloc(B) failed');
        error = coder.ceval('cudaMalloc', ...
            coder.wref(d_C), size_C);
        assert(error == cudaSuccessV, ...
            'cudaMalloc(C) failed');

        % Define direction of copying
        hostToDevice = coder.opaque('cudaMemcpyKind', ...
            'cudaMemcpyHostToDevice');
```

```
% Copy matrices to destination GPU
error = coder.ceval('cudaMemcpy',  ...
    d_A, coder.rref(A), size_A, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(A) failed');

error = coder.ceval('cudaMemcpy',  ...
    d_B, coder.rref(B), size_B, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(B) failed');

% Define type and size for result
C = zeros(m, n, 'single');

error = coder.ceval('cudaMemcpy', ...
    d_C, coder.rref(C), size_C, hostToDevice);
assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

% Define handle variables for external library
handle = coder.opaque('cublasHandle_t');
blasSuccess = coder.opaque('cublasStatus_t', ...
    'CUBLAS_STATUS_SUCCESS');

% Initialize external library
ret = coder.opaque('cublasStatus_t');
ret = coder.ceval('cublasCreate', coder.wref(handle));
assert(ret == blasSuccess, 'cublasCreate failed');


TRANSA = coder.opaque('cublasOperation_t', ...
    'CUBLAS_OP_N');
alpha = single(1);
beta = single(0);

% Multiply matrices on GPU
ret = coder.ceval('cublasSgemm', handle, ...
    TRANSA,TRANSA,m,n,k, ...
    coder.rref(alpha),d_A,m, ...
    d_B,k, ...
    coder.rref(beta),d_C,k);

assert(ret == blasSuccess, 'cublasSgemm failed');

% Copy result back to local host
deviceToHost = coder.opaque('cudaMemcpyKind', ...
    'cudaMemcpyDeviceToHost');
```

```
                    error = coder.ceval('cudaMemcpy', coder.wref(C), ...
                        d_C, size_C, deviceToHost);
                    assert(error == cudaSuccessV, 'cudaMemcpy(C) failed');

                end
            end
        end
    end
```

2 To perform the matrix multiplication using the interface defined in method `GPU_MatrixMultiply` and the build information in `ExternalLib_API`, include the following line in your MATLAB code:

```
C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

For instance, you can define a MATLAB function `Matrix_Multiply` that solely performs this matrix multiplication.

```
function C = Matrix_Multiply(A, B) %#codegen
 C= ExternalLib_API.GPU_MatrixMultiply(A,B);
```

3 Define a MEX configuration object using `coder.config`. For using the CUBLAS libraries, set the target language for code generation to C++.

```
cfg=coder.config('mex');
cfg.TargetLang='C++';
```

4 Generate code for `Matrix_Multiply` using `cfg` as the configuration object and two `2 X 2` matrices of type `single` as arguments. Since `cublasSgemm` supports matrix multiplication for data type `float`, the corresponding MATLAB matrices must have type `single`.

```
codegen -config cfg Matrix_Multiply ...
        -args {ones(2,'single'),ones(2,'single')}
```

5 Test the generated MEX function `Matrix_Multiply_mex` using two `2 X 2` identity matrices of type `single`.

```
Matrix_Multiply_mex(eye(2,'single'),eye(2,'single'))
```

The output is also a `2 X 2` identity matrix.

See Also

coder.BuildConfig | `assert` | `coder.ceval` | coder.ExternalDependency | `coder.opaque` | `coder.rref` | `coder.wref`

Related Examples

- "Encapsulate Interface to an External C Library" on page 27-6

More About

- "Encapsulating the Interface to External Code" on page 27-3

MATLAB Coder Optimizations in Generated Code

In this section...

In order to improve the execution speed and memory usage of generated code, MATLAB Coder introduces the following optimizations:

Constant Folding

When possible, the code generator evaluates expressions in your MATLAB code that involve compile-time constants only. In the generated code, it replaces these expressions with the result of the evaluations. This behavior is known as constant folding. Because of constant folding, the generated code does not have to evaluate the constants during execution.

The following example shows MATLAB code that is constant-folded during code generation. The function `MultiplyConstant` multiplies every element in a matrix by a scalar constant. The function evaluates this constant using the product of three compile-time constants, `a`, `b`, and `c`.

```matlab
function out=MultiplyConstant(in) %#codegen
 a=pi^4;
 b=1/factorial(4);
 c=exp(-1);
 out=in.*(a*b*c);
end
```

The code generator evaluates the expressions involving compile-time constants, `a`,`b`, and `c`. It replaces these expressions with the result of the evaluation in generated code.

Constant folding can occur when the expressions involve scalars only. To explicitly enforce constant folding of expressions in other cases, use the `coder.const` function. For more information, see "Fold Function Calls into Constants" on page 28-15.

Control Constant Folding

You can control the maximum number of instructions that can be constant-folded from the command line or the project settings dialog box.

- At the command line, create a configuration object for code generation. Set the property ConstantFoldingTimeout to the value that you want.

  ```
  cfg=coder.config('lib');
  cfg.ConstantFoldingTimeout = 200;
  ```
- Using the app, in the project settings dialog box, on the **All Settings** tab, set the field **Constant folding timeout** to the value that you want.

Loop Fusion

When possible, the code generator fuses successive loops with the same number of runs into a single loop in the generated code. This optimization reduces loop overhead.

The following code contains successive loops, which are fused during code generation. The function SumAndProduct evaluates the sum and product of the elements in an array Arr. The function uses two separate loops to evaluate the sum y_f_sum and product y_f_prod.

```
function [y_f_sum,y_f_prod] = SumAndProduct(Arr) %#codegen
  y_f_sum = 0;
  y_f_prod = 1;
  for i = 1:length(Arr)
     y_f_sum = y_f_sum+Arr(i);
  end
  for i = 1:length(Arr)
     y_f_prod = y_f_prod*Arr(i);
  end
```

The code generated from this MATLAB code evaluates the sum and product in a single loop.

Successive Matrix Operations Combined

When possible, the code generator converts successive matrix operations in your MATLAB code into a single loop operation in generated code. This optimization reduces excess loop overhead involved in performing the matrix operations in separate loops.

The following example contains code where successive matrix operations take place. The function ManipulateMatrix multiplies every element of a matrix Mat with a factor. To every element in the result, the function then adds a shift:

```
function Res=ManipulateMatrix(Mat,factor,shift)
  Res=Mat*factor;
  Res=Res+shift;
end
```

The generated code combines the multiplication and addition into a single loop operation.

Unreachable Code Elimination

When possible, the code generator suppresses code generation from unreachable procedures in your MATLAB code. For instance, if a branch of an if, elseif, else statement is unreachable, then code is not generated for that branch.

The following example contains unreachable code, which is eliminated during code generation. The function SaturateValue returns a value based on the range of its input x.

```
function y_b = SaturateValue(x) %#codegen
  if x>0
    y_b = x;
  elseif x>10 %This is redundant
    y_b = 10;
  else
    y_b = -x;
 end
```

The second branch of the if, elseif, else statement is unreachable. If the variable x is greater than 10, it is also greater than 0. Therefore, the first branch is executed in preference to the second branch.

MATLAB Coder does not generate code for the unreachable second branch.

memcpy Calls

To optimize generated code that copies consecutive array elements, the code generator tries to replace the code with a memcpy call. A memcpy call can be more efficient than code, such as a for-loop or multiple, consecutive element assignments.

See “memcpy Optimization” on page 28-51.

`memset` Calls

To optimize generated code that assigns a literal constant to consecutive array elements, the code generator tries to replace the code with a `memset` call. A `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

See “memset Optimization” on page 28-53.

memcpy Optimization

To optimize generated code that copies consecutive array elements, the code generator tries to replace the code with a `memcpy` call. A `memcpy` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

Code Generated with the memcpy Optimization	Code Generated without the memcpy Optimization

`memcpy(&C[0], &A[0], 10000U * sizeof(double));`	`for (i0 = 0; i0 < 10000; i0++) {`
	`    C[i0] = A[i0];`

`memcpy(&Z[0], &X[0],1000U * sizeof(double));`	`Z[0] = X[0];`
	`Z[1] = X[1];`
	`Z[2] = X[2];`
	`...`
	`Z[999] = X[999];`

The code generator invokes the `memcpy` optimization if the following conditions are true:

- The `memcpy` optimization is enabled.
- The number of bytes to copy is greater than or equal to the `memcpy` threshold. The number of bytes to copy is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

To enable or disable the `memcpy` optimization:

- At the command line, set the code configuration object property `EnableMemcpy` to `true` or `false`, respectively. The default value is `true`.
- In the MATLAB Coder app, set **Use memcpy for vector assignment** to `Yes` or `No` respectively. The default value is `Yes`.

The default `memcpy` threshold is 64 bytes. To change the threshold:

- At the command line, set the code configuration object property `MemcpyThreshold`.
- In the MATLAB Coder app, set **Memcpy threshold (bytes)**.

The `memset` optimization also uses the `memcpy` threshold.

More About

- “memset Optimization” on page 28-53
- “MATLAB Coder Optimizations in Generated Code” on page 28-47

- “Optimization Strategies” on page 28-3

memset Optimization

To optimize generated code that assigns a literal constant to consecutive array elements, the code generator tries to replace the code with a `memset` call. A `memset` call can be more efficient than code, such as a `for`-loop or multiple, consecutive element assignments.

Code Generated with the memset Optimization	Code Generated without the memset Optimization

`memset(&Y[0], 125, 100U * sizeof(signed char));`	`for (i = 0; i < 100; i++) {`
	`    Y[i] = 125;`

`memset(&Y[0], 0, 10000U * sizeof(double));`	`for (i0 = 0; i0 < 10000; i0++) {`
	`    Y[i0] = 0.0;`

`memset(&Z[0], 0, 1000U * sizeof(double));`	`Z[0] = 0.0;`
	`Z[1] = 0.0;`
	`Z[2] = 0.0;`
	`...`
	`Z[999] = 0.0;`

memset Optimization for an Integer Constant

To assign an integer constant to consecutive array elements, the code generator invokes the `memset` optimization when the following conditions are true:

- The constant is a literal constant. For example, `X[i] = 5`.
- For a nonzero constant:
  - The type of the constant is not multiword.
  - The length in bits of the type of the constant is the same as the length in bits of the C `char` type that the hardware supports.
- The number of bytes to assign is greater than or equal to the `memset` optimization threshold. The number of bytes to assign is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

The `memset` optimization threshold is the same as the `memcpy` optimization threshold. The default threshold is 64 bytes. To change the threshold:

- At the command line, set the code configuration object property `MemcpyThreshold`.
- In the MATLAB Coder app, set **Memcpy threshold (bytes)**.

memset Optimization for Float or Double Zero

To assign a float or double 0 to consecutive array elements, the code generator invokes the `memset` optimization when the following conditions are true:

- The constant is a literal constant. For example, `X[i] = 0.0`.
- The `memset` optimization is enabled for assignment of float or double 0 to consecutive array elements.
- The number of bytes to assign is greater than or equal to the `memset` optimization threshold. The number of bytes to assign is the number of array elements multiplied by the number of bytes required for the C/C++ data type.

To enable or disable the `memset` optimization for assignment of float or double 0 to consecutive array elements:

- At the command line, set the code configuration object property `InitFltsAndDblsToZero` to `true` or `false`, respectively. The default value is `true`.
- In the MATLAB Coder app, set **Use memset to initialize floats and doubles to 0.0** to `Yes` or `No` respectively. The default value is `Yes`.

The `memset` optimization threshold is the same as the `memcpy` optimization threshold. The default threshold is 64 bytes. To change the threshold:

- At the command line, set the code configuration object property `MemcpyThreshold`.
- In the MATLAB Coder app, set **Memcpy threshold (bytes)**.

More About

- “memcpy Optimization” on page 28-51
- “MATLAB Coder Optimizations in Generated Code” on page 28-47
- “Optimization Strategies” on page 28-3

Generate Reusable Code

With MATLAB, you can generate reusable code in the following ways:

- Write reusable functions using standard MATLAB function file names which you can call from different locations, for example, in a Simulink model or MATLAB function library.
- Compile external functions on the MATLAB path and integrate them into generated C code for embedded targets.

See “Resolution of Function Calls for Code Generation” on page 13-2.

Common applications include:

- Overriding generated library function with a custom implementation.
- Implementing a reusable library on top of standard library functions that can be used with Simulink.
- Swapping between different implementations of the same function.

Reuse Large Arrays and Structures

Variable reuse can reduce memory usage or improve execution speed, especially when your code has large structures or arrays. However, variable reuse results in less readable code. If reduced memory usage is more important than code readability, specify that you want the code generator to reuse your variables in the generated code.

The code generator can reuse the name and memory of one variable for another variable when:

- Both variables have the same memory requirements.
- Memory access for one variable does not interfere with memory access for the other variable.

The code generator reuses your variable names for other variables or reuses other variable names for your variables. For example, for code such as:

```
if (s>0)
    myvar1 = 0;
    ...
else
    myvar2 = 0;
    ...
end
```

the generated code can look like this code:

```
 if (s > 0.0) {
   myvar2 = 0.0;
    ...
 } else {
   myvar2 = 0.0;
   ...
 }
```

To specify that you want the code generator to reuse your variables:

- In a code generation configuration object, set the PreserveVariableNames parameter to 'None'.
- In the MATLAB Coder app, set **Preserve variable names** to None.

More About

- “Preserve Variable Names in Generated Code” on page 20-44
- “Optimization Strategies” on page 28-3
- “Configure Build Settings” on page 20-26

LAPACK Calls in Generated Code

To improve the execution speed of code generated for certain linear algebra functions, MATLAB Coder can generate calls to LAPACK functions instead of generating the code for the linear algebra functions. LAPACK is a software library for numerical linear algebra. MATLAB Coder uses the LAPACKE C interface to LAPACK.

For MEX generation, if the input arrays for the linear algebra functions meet certain criteria, the code generator produces LAPACK calls. For standalone code (library or executable program), by default, the code generator does not produce LAPACK calls. If you specify that you want to generate LAPACK calls, and the input arrays for the linear algebra functions meet the criteria, the code generator produces LAPACK calls. See "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" on page 28-59.

For MEX functions, the code generator uses the LAPACK library that is included with MATLAB. MATLAB uses LAPACK in some linear algebra functions such as `eig` and `svd`. For standalone code, the code generator uses the LAPACK library that you specify. See "Specify LAPACK Library" on page 28-59.

More About

- "Optimization Strategies" on page 28-3

External Websites

- www.netlib.org/lapack

Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls

To improve the execution speed of code generated for certain linear algebra functions in standalone (library or executable program) code, specify that you want MATLAB Coder to generate LAPACK calls. LAPACK is a software library for numerical linear algebra. MATLAB Coder uses the LAPACKE C interface to LAPACK. If you specify that you want to generate LAPACK calls, and the input arrays for the linear algebra functions meet certain criteria, the code generator produces the LAPACK calls. Otherwise, the code generator produces code for the linear algebra functions.

For LAPACK calls in standalone code, MATLAB Coder uses the LAPACK library that you specify. Specify a LAPACK library that is optimized for your execution environment. See www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations.

Specify LAPACK Library

To generate LAPACK calls in standalone code, you must have access to a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file for the LAPACK calls. To indicate that you want to generate LAPACK calls and that you want to use a specific LAPACK library, specify the name of the LAPACK callback class.

- At the command line, set the code configuration object property `CustomLAPACKCallback` to the name of the callback class.
- In the MATLAB Coder app, set **Custom LAPACK library callback** to the name of the callback class.

Write LAPACK Callback Class

To specify the locations of a particular LAPACK library and LAPACKE header file, write a LAPACK callback class. Share the callback class with others who want to use this LAPACK library for LAPACK calls in standalone code.

The callback class must derive from the abstract class coder.LAPACKCallback. Use the following example callback class as a template.

```
classdef useMyLAPACK < coder.LAPACKCallback
```

```
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
        function updateBuildInfo(buildInfo, buildctx)
            buildInfo.addIncludePaths(fullfile(pwd,'include'));
            libName = 'mylapack';
            libPath = fullfile(pwd,'lib');
            [~,linkLibExt] = buildctx.getStdLibInfo();
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
                '', true, true);
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
        end
    end
end
```

You must provide the getHeaderFilename and updateBuildInfo methods. The getHeaderFilename method returns the LAPACKE header file name. In the example callback class, replace mylapacke_custom.h with the name of your LAPACKE header file. The updateBuildInfo method provides the information required for the build process to link to the LAPACK library. Use code like the code in the template to specify the location of header files and the full path name of the LAPACK library. In the example callback class, replace mylapack with the name of your LAPACK library.

If your compiler supports only complex data types that are represented as structures, include these lines in the updateBuildInfo method.

```
buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
```

Generate LAPACK Calls by Specifying a LAPACK Callback Class

This example shows how to generate code that calls LAPACK functions in a specific LAPACK library. For this example, assume that the LAPACK callback class useMyLAPACK specifies the LAPACK library that you want to use.

1 Write a MATLAB function that calls a linear algebra function. For example, write a function mysvd that calls the MATLAB function svd.

```
function s = mysvd(A)
    %#codegen
    s = svd(A);
    end
```

2. Define a code configuration object for a static library, dynamically linked library, or executable program. For example, define a configuration object for a dynamically linked library on a Windows platform.

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

3. Specify the LAPACK callback class `useMyLAPACK`.

   ```
   cfg.CustomLAPACKCallback = 'useMyLAPACK';
   ```

   The callback class must be on the MATLAB path.

4. Generate code. Specify that the input `A` is a 500-by-500 array of doubles.

   ```
   codegen test -args {zeros(500)} -config cfg -report
   ```

If `A` is large enough, the code generator produces a LAPACK call for `svd`. Here is an example of a call to the LAPACK library function for `svd`.

```
info_t = LAPACKE_dgesvd(LAPACK_COL_MAJOR, 'N', 'N', (lapack_int)500,
    (lapack_int)500, &A[0], (lapack_int)500, &S[0], NULL, (lapack_int)1, NULL,
    (lapack_int)1, &superb[0]);
```

Locate LAPACK Library in Execution Environment

The LAPACK library must be available in your execution environment. If your LAPACK library is shared, use environment variables or linker options to specify the location of the LAPACK library.

- On a Windows platform, modify the PATH environment variable.
- On a Linux platform, modify the LD_LIBRARY_PATH environment variable or use the `rpath` linker option.
- On a Mac OS X, modify the DYLD_LIBRARY_PATH environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, you can use the build information `addLinkFlags` method in the `updateBuildInfo` method of your coder.LAPACKCallback class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

See Also

coder.LAPACKCallback

More About

- “LAPACK Calls in Generated Code” on page 28-58
- “Optimization Strategies” on page 28-3

External Websites

- www.netlib.org/lapack
- www.netlib.org/lapack/
faq.html#_what_and_where_are_the_lapack_vendors_implementations

Speed Up MEX Generation by Using JIT Compilation

In this section...

To speed up generation of a MEX function, specify use of just-in-time (JIT) compilation technology. When you iterate between modifying MATLAB code and testing the MEX code, using this option can save time.

By default, MATLAB Coder creates a *C/C++ MEX* function by generating and compiling C/C++ code. When you specify JIT compilation, MATLAB Coder creates a *JIT MEX* function that contains an abstract representation of the MATLAB code. When you run the JIT MEX function, MATLAB generates the executable code in memory.

JIT compilation is incompatible with certain code generation features or options. See “JIT Compilation Incompatibilities” on page 28-64. If JIT compilation is enabled, the absence of warning or error messages during code generation indicates successful JIT compilation. The **C code** tab of the code generation report indicates the use of JIT compilation.

Specify Use of JIT Compilation in the MATLAB Coder App

1. To open the **Generate** dialog box, click the **Generate** arrow ▼.
2. Set **Build type** to `MEX`.
3. Select the **Use JIT compilation** check box.

Specify Use of JIT Compilation at the Command Line

Use the `-jit` option of the `codegen` command. For example, specify JIT compilation for `myfunction`:

```
codegen -config:mex myfunction -jit -report
```

Alternatively, use the `EnableJIT` code configuration parameter.

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

```
cfg.EnableJIT = true;
codegen -config cfg myfunction -report
```

JIT Compilation Incompatibilities

The following table summarizes code generation features or options that are incompatible with JIT compilation.

Incompatibility	Message Type	Generated MEX	Action
Custom Code	Warning	C/C++ MEX	To avoid the warning, disable JIT compilation.
Updating build information (`coder.updateBuildInfo`)	Warning	C/C++ MEX	To avoid the warning, disable JIT compilation.
Use of OpenMP application interface for parallelization of `for`-loops (`parfor`)	Warning	• JIT MEX • No parallelization	If you want parallelization of `for`-loops, disable JIT compilation.
Generation of C/C++ source code only	Error	None	Specify either JIT compilation or generation of C/C ++ code only.

See Also

coder.MexCodeConfig | `codegen` | `coder.updateBuildInfo` | `parfor`

More About

- “JIT MEX Incompatibility Warning” on page 30-2
- “JIT Compilation Does Not Support OpenMP” on page 30-3
- “Speed Up Compilation by Generating Only Code” on page 20-80
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 25-18

Generating Reentrant C Code from MATLAB Code

Generate Reentrant C Code from MATLAB Code

In this section...

About This Tutorial

Learning Objectives

This tutorial shows you how to:

- Generate reentrant code from MATLAB code that does not use persistent or global data.
- Automatically generate C code from your MATLAB code.
- Define function input properties at the command line.
- Specify code generation properties.
- Generate a code generation report that you can use to view and debug your MATLAB code.

Note: This example runs on Windows only.

Prerequisites

To complete this example, install the following products:

- MATLAB
- MATLAB Coder
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). MATLAB Coder locates and uses a supported installed compiler. For the current list of supported compilers, see `http://www.mathworks.com/support/compilers/current_release/` on the MathWorks website.

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

Required Files

Type	Name	Description
Function code	`matrix_exp.m`	MATLAB function that computes matrix exponential of the input matrix using Taylor series and returns the computed output.
C main function	`main.c`	Calls the reentrant code.

Copying Files Locally

Copy the tutorial files to a local working folder.

1. Create a local working folder, for example, `c:\coder\work`.
2. Change to the `matlabroot\help\toolbox\coder\examples` folder. At the MATLAB command prompt, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
   ```
3. Copy the `reentrant_win` folder to your local working folder.

   Your work folder now contains the files for the tutorial.
4. Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command prompt, enter:

   ```
   cd work
   ```

   `work` is the full path of the work folder containing your files.

About the Example

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a simple, multithreaded example that does not use persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

Contents of matrix_exp.m

```
function Y = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential of
    % the input matrix using Taylor series and returns the
    % computed output.
    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E;
```

When you generate reusable, reentrant code, MATLAB Coder supports dynamic allocation of:

- Function variables that are too large for the stack
- Persistent variables
- Global variables

MATLAB Coder generates a header file, `*primary_function_name*_types.h`, that you must include when using the generated code. This header file contains the following structures:

- `*primary_function_name*StackData`

  Contains the user allocated memory. Pass a pointer to this structure as the first parameter to functions that use it:

  - Directly (the function uses a field in the structure)
  - Indirectly (the function passes the structure to a called function)

  If the algorithm uses persistent or global data, the `*primary_function_name*StackData` structure also contains a pointer to the *primary_function_name*PersistentData structure. If you include this pointer, you have to pass only one parameter to each calling function.

- *primary_function_name*PersistentData

  If your algorithm uses persistent or global variables, MATLAB Coder provides a separate structure for them. The memory allocation structure contains a pointer to this persistent data structure.Because you have a separate structure for persistent and global variables, you can allocate memory for these variables once and share them with all threads. However, if the threads do not communicate, you can allocate memory for these variables per thread or per application.

Providing a C main Function

To call the reentrant code, provide a main function that:

- Includes the generated header file matrix_exp.h. This file includes the generated header file, matrix_exp_types.h.
- For each thread, allocates memory for stack data.
- Calls the matrix_exp_initialize housekeeping function. For more information, see "Calling Initialize and Terminate Functions" on page 24-9.
- Calls matrix_exp.
- Calls matrix_exp_terminate.
- Frees up the for stack data memory.

Contents of main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    matrix_expStackData* spillData;
} IODATA;

/* The thread_function calls the matrix_exp function written in MATLAB */
DWORD WINAPI thread_function(PVOID dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize();
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
    matrix_exp_terminate();
    return 0;
}

void main() {
  HANDLE thread1, thread2;
  IODATA data1;
  IODATA data2;
  int32_T i;

  /*Initializing data for passing to the 2 threads*/
  matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
  matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

  data1.spillData = sd1;
  data2.spillData = sd2;

  for (i=0;i<NUMELEMENTS;i++) {
      data1.in[i] = 1;
      data1.out[i] = 0;
      data2.in[i] = 1.1;
      data2.out[i] = 0;
  }

  /*Initializing the 2 threads and passing data to the thread functions*/
  printf("Starting thread 1...\n");
  thread1 = CreateThread(NULL , 0, thread_function, (PVOID) &data1, 0, NULL);
  if (thread1 == NULL){
      perror( "Thread 1 creation failed.");
     exit(EXIT_FAILURE);
   }

  printf("Starting thread 2...\n");
  thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
  if (thread2 == NULL){
      perror( "Thread 2 creation failed.");
      exit(EXIT_FAILURE);
  }

  /*Wait for both the threads to finish execution*/
  if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
      perror( "Thread 1 join failed.");
    exit(EXIT_FAILURE);
   }

  if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
    perror( "Thread 2 join failed.");
    exit(EXIT_FAILURE);
   }

  free(sd1);
  free(sd2);

  printf("Finished Execution!\n");
  exit(EXIT_SUCCESS);
  }
```

Configuring Build Parameters

You can enable generation of reentrant code using a code generation configuration object.

1. Create a configuration object.

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

2. Enable reentrant code generation.

   ```
   cfg.MultiInstanceCode = true;
   ```

Generating the C Code

Call the codegen function to generate C code, with the following options:

- -config to pass in the code generation configuration object cfg.
- main.c to include this file in the compilation.
- -report to create a code generation report.
- -args to specify the class, size, and complexity of input arguments using example data.

```
codegen -config cfg main.c -report matrix_exp.m -args ones(160,160)
```

codegen generates a C executable, matrix_exp.exe, in the current folder and C code in the /codegen/exe/matrix_exp subfolder. Because you selected report generation, codegen provides a link to the report.

Viewing the Generated C Code

codegen generates a header file matrix_exp_types.h, which defines the matrix_expStackData global structure. This structure contains local variables that are too large to fit on the stack.

To view this header file:

1. Click the View report link to open the code generation report.
2. In the report, click the **C code** tab.
3. On this tab, click the link to matrix_exp_types.h.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */
```

```
#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
  struct {
    double F[25600];
    double Y[25600];
    double X[25600];
  } f0;
} matrix_expStackData;

#endif                                 /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

Running the Code

Verify that the example is running on Windows platforms and call the code.

```
% This example can only be run on Windows platforms
if ~ispc
 error('This example requires Windows-specific libraries and can only be run on Windows.');
end
system('matrix_exp.exe')
```

The executable runs and reports completion.

Key Points to Remember

- Create a `main` function that:
  - Includes the generated header file, *`primary_function_name`*`_types.h`. This file defines the *`primary_function_name`*`StackData` global structure. This structure contains local variables that are too large to fit on the stack.
  - For each thread, allocates memory for stack data.
  - Calls *`primary_function_name`*`_initialize`.
  - Calls *`primary_function_name`*.

  - Calls `primary_function_name_terminate`.
  - Frees the stack data memory.
- Use the `-config` option to pass the code generation configuration object to the `codegen` function.
- Use the `-args` option to specify input parameters at the command line.
- Use the `-report` option to create a code generation report.

Learn More

To	See
Learn more about the generated code API	"API for Generated Reusable Code" on page 29-14
Call reentrant code without persistent or global data on UNIX	"Call Reentrant Code with No Persistent or Global Data (UNIX Only)" on page 29-17
Call reentrant code with persistent data on Windows	"Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)" on page 29-22
Call reentrant code with persistent data on UNIX	"Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)" on page 29-27

Reentrant Code

Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Operating systems and other system software that use multithreading to handle concurrent events use reentrant code. In a concurrent environment, multiple threads or processes can attempt to read and write static data simultaneously. Therefore, sharing code that uses persistent or static data is difficult. Reentrant code does not contain static data. Calling programs maintain their state variables and pass them into the function. Therefore, any number of threads or processes can share one copy of a reentrant routine.

Generate reentrant code when you want to:

- Deploy your code in multithreaded environments.
- Use an algorithm with persistent data belonging to different processes or threads.
- Compile code that uses function variables that are too large to fit on the stack.

If you do not specify reentrant code, MATLAB Coder generates code that uses statically allocated memory for:

- Function variables that are too large to fit on the stack
- Global variables
- Persistent variables

If the generated code uses static memory allocation for these variables, you cannot deploy the generated code in environments that require reentrant code. If you cannot adjust the static memory allocation size, the generated code can result in static memory size overflow.

When you generate reentrant code, MATLAB Coder creates input data structures for:

- Function variables that are too large to fit on the stack
- Persistent variables
- Global variables

You can then dynamically allocate memory for these input structures. The use of dynamic memory allocation means that you can deploy the code in reentrant environments.

Related Examples

- “Specify Generation of Reentrant Code” on page 29-12
- “Generate Reentrant C Code from MATLAB Code” on page 29-2
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 29-17
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 29-22
- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 29-27

Specify Generation of Reentrant Code

In this section...

Specify Generation of Reentrant Code Using the MATLAB Coder App

1. On the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to one of the following:

   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Memory** tab, select the **Generate re-entrant code** check box.

Specify Generation of Reentrant Code Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

   ```
   cfg = coder.config('lib'); % or dll or exe
   ```
2. Set the `MultiInstanceCode` property to `true`. For example:

   ```
   cfg.MultiInstanceCode = true;
   ```

Related Examples

- “Generate Reentrant C Code from MATLAB Code” on page 29-2
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 29-17
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 29-22

- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 29-27

More About

- “Reentrant Code” on page 29-10

API for Generated Reusable Code

When you generate reusable code, MATLAB Coder supports dynamic allocation of:

- Function variables that are too large for the stack
- Persistent variables
- Global variables

It generates a header file, `*primary_function_name*_types.h`, that you must include when using the generated code. This header file contains the following structures:

- *`primary_function_name`*`StackData`

  This structure contains the user-allocated memory. You must pass a pointer to this structure as the first parameter to all functions that use it:

  - Directly, because the function uses a field in the structure.
  - Indirectly, because the function passes the structure to a called function.

  If the algorithm uses persistent or global data, the *`primary_function_name`*`StackData` structure also contains a pointer to the *`primary_function_name`*`PersistentData` structure. If you include this pointer, you only have to pass one parameter to each calling function.
- *`primary_function_name`*`PersistentData`

  If your algorithm uses persistent or global variables, MATLAB Coder provides a separate structure for them. The memory allocation structure contains a pointer to this structure. Because you have a separate structure for persistent and global variables, you can allocate memory for these variables once and share them with all threads. However, if there is no communication between threads, you can choose to allocate memory for these variables per thread or per application.

For more information on using these global structures, see “Multithreaded Examples” on page 29-16.

Call Reentrant Code in a Single-Threaded Environment

To call reentrant code in a single-threaded environment, create a `main` function that:

- Includes the header file *`primary_function_name`*.h.
- Allocates memory for the global memory allocation structure *`primary_function_name`*`StackData`.
- If the algorithm uses persistent or global data, allocates memory for the global structure *`primary_function_name`*`PersistentData`.
- Calls these functions:

  - *`primary_function_name`*`_initialize`.
  - *`primary_function_name`*.
  - *`primary_function_name`*`_terminate`.

  When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder generates two housekeeping functions. Call these housekeeping functions in the code that calls the generated C/C++ function. For more information, see “Calling Initialize and Terminate Functions” on page 24-9.
- Frees the memory used for global structures.

Call Reentrant Code in a Multithreaded Environment

To call reentrant code, create a `main` function that:

- Includes the header file *`primary_function_name`*.h.
- For each thread, allocates memory for the global memory allocation structure *`primary_function_name`*`StackData`.
- If the algorithm uses persistent or global data, allocates memory for the global structure *`primary_function_name`*`PersistentData`. If the threads communicate, allocate this memory once for the application. Otherwise, you can choose to allocate memory per thread or per application.
- Contains a thread function that calls these functions:
  - *`primary_function_name`*`_initialize`.
  - *`primary_function_name`*.
  - *`primary_function_name`*`_terminate`.

  When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, MATLAB Coder generates two housekeeping functions. Call these functions in the code that calls the generated C/C++ function. For more information, see “Calling Initialize and Terminate Functions” on page 24-9.
- Initializes each thread and passes in a pointer to the memory allocation structure as the first parameter to the thread function.
- Frees up the memory used for global structures.

Multithreaded Examples

Type of Reentrant Code	Platform	Reference
Multithreaded without persistent or global data	Windows	“Generate Reentrant C Code from MATLAB Code” on page 29-2
	UNIX	“Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 29-17
Multithreaded with persistent or global data	Windows	“Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 29-22
	UNIX	“Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 29-27

Call Reentrant Code with No Persistent or Global Data (UNIX Only)

In this section...

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a simple multithreaded example that uses no persistent or global data. Two threads call the MATLAB function matrix_exp with different sets of input data.

Provide a Main Function

To call the reentrant code, provide a main function that:

- Includes the header file matrix_exp.h.
- For each thread, allocates memory for stack data.
- Calls the matrix_exp_initialize housekeeping function. For more information, see "Calling Initialize and Terminate Functions" on page 24-9.
- Calls matrix_exp.
- Calls matrix_exp_terminate.
- Frees the memory used for stack data.

For this example, main.c contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    matrix_expStackData* spillData;
} IODATA;
```

```
/* The thread_function calls the matrix_exp function written in MATLAB */
void *thread_function(void *dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize();
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
    matrix_exp_terminate();
}

int main() {
    pthread_t thread1, thread2;
    int  iret1, iret2;
    IODATA data1;
    IODATA data2;
    int32_T i;

    /*Initializing data for passing to the 2 threads*/
    matrix_expStackData* sd1=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
    matrix_expStackData* sd2=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

    data1.spillData = sd1;
    data2.spillData = sd2;

    for (i=0;i<NUMELEMENTS;i++) {
        data1.in[i] = 1;
        data1.out[i] = 0;
        data2.in[i] = 1.1;
        data2.out[i] = 0;
    }

    /*Initializing the 2 threads and passing required data to the thread functions*/
    printf("Starting thread 1...\n");
    iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
    if (iret1 != 0){
  perror( "Thread 1 creation failed.");
  exit(EXIT_FAILURE);
 }

    printf("Starting thread 2...\n");
    iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
    if (iret2 != 0){
        perror( "Thread 2 creation failed.");
        exit(EXIT_FAILURE);
    }

    /*Wait for both the threads to finish execution*/
    iret1 = pthread_join(thread1, NULL);
    if (iret1 != 0){
  perror( "Thread 1 join failed.");
  exit(EXIT_FAILURE);
 }

    iret2 = pthread_join(thread2, NULL);
    if (iret2 != 0){
  perror( "Thread 2 join failed.");
  exit(EXIT_FAILURE);
 }

    free(sd1);
    free(sd2);

    printf("Finished Execution!\n");
    exit(EXIT_SUCCESS);

}
```

Generate Reentrant C Code

To generate code, run the following script at the MATLAB command prompt.

```
% This example can only be run on Unix platforms
if ~isunix
  error('This example requires pthread libraries and can only be run on Unix.');
end

% Setting the options for the Config object

% Create a code gen configuration object
cfg = coder.config('exe');

% Enable reentrant code generation
cfg.MultiInstanceCode = true;

% Set the post code generation command to be the 'setbuildargs' function
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config cfg main.c matrix_exp.m -report -args ones(160,160)
```

This script:

- Generates an error message if the example is not running on a UNIX platform.
- Creates a code configuration object for generation of an executable.
- Enables the `MultiInstanceCode` option to generate reusable, reentrant code.
- Uses the `PostCodeGenCommand` option to set the post code generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

  ```
  function setbuildargs(buildInfo)
  % The example being compiled requires pthread support.
  % The -lpthread flag requests that the pthread library
  % be included in the build
      linkFlags = {'-lpthread'};
  ```

```
addLinkFlags(buildInfo, linkFlags);
```

For more information, see “Customize the Post-Code-Generation Build Process” on page 20-140.

- Invokes `codegen` with the following options:
  - `-config` to pass in the code generation configuration object `cfg`.
  - `main.c` to include this file in the compilation.
  - `-report` to create a code generation report.
  - `-args` to specify an example input with class, size, and complexity.

Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines the `matrix_expStackData` global structure. This structure contains local variables that are too large to fit on the stack.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
  struct {
    double F[25600];
    double Y[25600];
    double X[25600];
  } f0;
} matrix_expStackData;

#endif                                 /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

Run the Code

Call the code using the command:

```
system('./matrix_exp')
```

The executable runs and reports completion.

Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)

In this section...

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code for This Example

```
function [Y,numTimes] = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential
    % of the input matrix using Taylor series and returns
    % the computed output. It also returns the number of
    % times this function has been called.
    %
    persistent count;
    if isempty(count)
        count = 0;
    end
    count = count+1;

    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E ;

    numTimes = count;
```

Provide a Main Function

To call reentrant code that uses persistent data, provide a main function that:

- Includes the header file matrix_exp.h.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the matrix_exp_initialize housekeeping function. For more information, see “Calling Initialize and Terminate Functions” on page 24-9.
- Calls matrix_exp.
- Calls matrix_exp_terminate.
- Frees the memory used for stack and persistent data.

For this example, main.c contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    real_T numTimes;
    matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
DWORD WINAPI thread_function(PVOID dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize(myIOData->spillData);
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData->numTimes);
    printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
    matrix_exp_terminate();
    return 0;
}

void main() {
    HANDLE thread1, thread2;
    IODATA data1;
    IODATA data2;
    int32_T i;
```

```
   /*Initializing data for passing to the 2 threads*/
   matrix_expPersistentData* pd1 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
   matrix_expPersistentData* pd2 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
   matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
   matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

   sd1->pd = pd1;
   sd2->pd = pd2;
   data1.spillData = sd1;
   data2.spillData = sd2;

   for (i=0;i<NUMELEMENTS;i++) {
       data1.in[i] = 1;
       data1.out[i] = 0;
       data2.in[i] = 1.1;
       data2.out[i] = 0;
   }

   data1.numTimes = 0;
   data2.numTimes = 0;

   /*Initializing the 2 threads and passing required data to the thread functions*/
   printf("Starting thread 1...\n");
   thread1 = CreateThread(NULL, 0, thread_function, (PVOID) &data1, 0, NULL);
   if (thread1 == NULL){
 perror( "Thread 1 creation failed.");
 exit(EXIT_FAILURE);
}

   printf("Starting thread 2...\n");
   thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
   if (thread2 == NULL){
       perror( "Thread 2 creation failed.");
       exit(EXIT_FAILURE);
   }

   /*Wait for both the threads to finish execution*/
   if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
 perror( "Thread 1 join failed.");
 exit(EXIT_FAILURE);
}

   if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
 perror( "Thread 2 join failed.");
 exit(EXIT_FAILURE);
}

   free(sd1);
   free(sd2);
   free(pd1);
   free(pd2);

   printf("Finished Execution!\n");
   exit(EXIT_SUCCESS);
```

```
}
```

Generate Reentrant C Code

Run the following script at the MATLAB command prompt to generate code.

```
% This example can only be run on Windows platforms
if ~ispc
 error...
   ('This example requires Windows-specific libraries and can only be run on Windows.');
end

% Setting the options for the Config object
% Create a code gen configuration object
cfg = coder.config('exe');

% Enable reentrant code generation
cfg.MultiInstanceCode = true;

% Compiling
codegen -config cfg main.c -report matrix_exp.m -args ones(160,160)
```

This script:

- Generates an error message if the example is not running on a Windows platform.
- Creates a code generation configuration object for generation of an executable.
- Enables the `MultiInstanceCode` option to generate reusable, reentrant code.
- Invokes `codegen` with the following options:
  - `-config` to pass in the code generation configuration object `cfg`.
  - `main.c` to include this file in the compilation.
  - `-report` to create a code generation report.
  - `-args` to specify an example input with class, size, and complexity.

Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, that defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expPersistentData
#define typedef_matrix_expPersistentData

typedef struct {
  double count;
} matrix_expPersistentData;

#endif                                 /*typedef_matrix_expPersistentData*/

#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
  struct {
    double F[25600];
    double Y[25600];
    double X[25600];
  } f0;

  matrix_expPersistentData *pd;
} matrix_expStackData;

#endif                                 /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

Run the Code

Call the code using the command:

```
system('matrix_exp.exe')
```

The executable runs and reports completion.

Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)

In this section...

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code for This Example

```
function [Y,numTimes] = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential
    % of the input matrix using Taylor series and returns
    % the computed output. It also returns the number of
    % times this function has been called.
    %

    persistent count;
    if isempty(count)
        count = 0;
    end
    count = count+1;

    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E ;

    numTimes = count;
```

Provide a Main Function

To call reentrant code that uses persistent data, provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions” on page 24-9.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    real_T numTimes;
    matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
void *thread_function(void *dummyPtr) {
  IODATA *myIOData = (IODATA*)dummyPtr;
  matrix_exp_initialize(myIOData->spillData);
  matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData>numTimes);
  printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
  matrix_exp_terminate();
}

int main() {
  pthread_t thread1, thread2;
  int  iret1, iret2;
  IODATA data1;
  IODATA data2;
  int32_T i;
  /*Initializing data for passing to the 2 threads*/
  matrix_expPersistentData* pd1 =
     (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
  matrix_expPersistentData* pd2 =
     (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
  matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
  matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

  sd1->pd = pd1;
  sd2->pd = pd2;
  data1.spillData = sd1;
  data2.spillData = sd2;

  for (i=0;i<NUMELEMENTS;i++) {
      data1.in[i] = 1;
      data1.out[i] = 0;
      data2.in[i] = 1.1;
      data2.out[i] = 0;
  }

  data1.numTimes = 0;
  data2.numTimes = 0;

  /*Initializing the 2 threads and passing required data to the thread functions*/
  printf("Starting thread 1...\n");
  iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
  if (iret1 != 0){
   perror("Thread 1 creation failed.");
exit(EXIT_FAILURE);
 }

  printf("Starting thread 2...\n");
  iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
  if (iret2 != 0){
      perror( "Thread 2 creation failed.");
      exit(EXIT_FAILURE);
  }

  /*Wait for both the threads to finish execution*/
  iret1 = pthread_join(thread1, NULL);
  if (iret1 != 0){
   perror( "Thread 1 join failed.");
exit(EXIT_FAILURE);
 }

  iret2 = pthread_join(thread2, NULL);
  if (iret2 != 0){
   perror( "Thread 2 join failed.");
exit(EXIT_FAILURE);
 }

  free(sd1);
  free(sd2);
  free(pd1);
  free(pd2);
  printf("Finished Execution!\n");
  return(0);

}
```

Generate Reentrant C Code

To generate code, run the following script at the MATLAB command prompt.

```
% This example can only be run on Unix platforms
if ~isunix
  error('This example requires pthread libraries and can only be run on Unix.');
end

% Setting the options for the Config object

% Specify an ERT target
cfg = coder.config('exe');

% Enable reentrant code generation
cfg.MultiInstanceCode = true;

% Set the post code generation command to be the 'setbuildargs' function
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config cfg main.c  -report matrix_exp.m -args ones(160,160)
```

This script:

- Generates an error message if the example is not running on a UNIX platform.
- Creates a code generation configuration object for generation of an executable.
- Enables the `MultiInstanceCode` option to generate reusable, reentrant code.
- Uses the `PostCodeGenCommand` option to set the post-code-generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

  ```
  function setbuildargs(buildInfo)
  % The example being compiled requires pthread support.
  % The -lpthread flag requests that the pthread library
  % be included in the build
      linkFlags = {'-lpthread'};
      addLinkFlags(buildInfo, linkFlags);
  ```

  For more information, see “Customize the Post-Code-Generation Build Process” on page 20-140.
- Invokes `codegen` with the following options:

- `-config` to pass in the code generation configuration object `cfg`.
- `main.c` to include this file in the compilation.
- `-report` to create a code generation report.
- `-args` to specify an example input with class, size, and complexity.

Examine the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * Code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Include files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef typedef_matrix_expPersistentData
#define typedef_matrix_expPersistentData

typedef struct {
  double count;
} matrix_expPersistentData;

#endif                                 /*typedef_matrix_expPersistentData*/

#ifndef typedef_matrix_expStackData
#define typedef_matrix_expStackData

typedef struct {
  struct {
    double F[25600];
    double Y[25600];
    double X[25600];
  } f0;
  matrix_expPersistentData *pd;
} matrix_expStackData;

#endif                                  /*typedef_matrix_expStackData*/
#endif

/* End of code generation (matrix_exp_types.h) */
```

Run the Code

Call the code using the command:

```
system('./matrix_exp')
```

Troubleshooting Code Generation Problems

JIT MEX Incompatibility Warning

Issue

When you generate a MEX function, you see a warning message that starts with:

```
JIT compilation is incompatible with
```

MATLAB Coder generates a C/C++ MEX function instead of a JIT MEX function.

Cause

JIT compilation is incompatible with certain code generation features and options. If you include custom code or update the build information, you cannot generate a JIT MEX function. In these cases, MATLAB Coder generates a C/C++ MEX function instead of a JIT MEX function.

Solution

To eliminate the warning, disable JIT compilation.

More About

- “Speed Up MEX Generation by Using JIT Compilation” on page 28-63

JIT Compilation Does Not Support OpenMP

Issue

When you generate a MEX function for code that contains `parfor`, you see this warning message:

```
JIT technology does not support using OpenMP library, this loop will not be
parallelized.
```

MATLAB Coder generates a JIT MEX function and treats the `parfor`-loop as a `for`-loop.

Cause

JIT compilation and use of the OpenMP application interface are enabled. JIT compilation is incompatible with the OpenMP application interface.

Solution

If you want to parallelize `for`-loops, disable JIT compilation.

See Also

`parfor`

More About

- “Speed Up MEX Generation by Using JIT Compilation” on page 28-63
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 25-18

Output Variable Must Be Assigned Before Run-Time Recursive Call

Issue

You see this error message:

```
All outputs must be assigned before any run-time recursive call. Output 'y' is
not assigned here.
```

Cause

Run-time recursion produces a recursive function in the generated code. The code generator is unable to use run-time recursion for a recursive function in your MATLAB code because an output is not assigned before the first recursive call.

Solution

Rewrite the code so that it assigns the output before the recursive call.

Direct Recursion Example

In the following code, the statement `y = A(1)` assigns a value to the output `y`. This statement occurs after the recursive call `y = A(1)+ mysum(A(2:end))`.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) > 1
    y = A(1)+ mysum(A(2:end));

else
    y = A(1);
end
end
```

Rewrite the code so that assignment `y = A(1)` occurs in the `if` block and the recursive call occurs in the `else` block.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');

if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Alternatively, before the `if` block, add an assignment, for example, `y = 0`.

```
function z = call_mysum(A)
B = A;
coder.varsize('B');
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
y = 0;
if size(A,2) > 1
    y = A(1)+ mysum(A(2:end));

else
    y = A(1);
end
end
```

Indirect Recursion Example

In the following code, `rec1` calls `rec2` before the assignment `y = 0`.

```
function y = rec1(x)
%#codegen

if x >= 0
    y = rec2(x-1)+1;
else
```

```
    y = 0;
end
end

function y = rec2(x)
y = rec1(x-1)+2;
end
```

Rewrite this code so that in `rec1`, the assignment `y = 0` occurs in the `if` block and the recursive call occurs in the `else` block.

```
function y = rec1(x)
%#codegen

if x < 0
    y = 0;
else
    y = rec2(x-1)+1;
end
end

function y = rec2(x)
y = rec1(x-1)+2;
end
```

More About

- “Code Generation for Recursive Functions” on page 13-18

Compile-Time Recursion Limit Reached

Issue

You see a message such as:

```
Compile-time recursion limit reached. Size or type of input #1 of function 'foo'
may change at every call.
```

```
Compile-time recursion limit reached. Size of input #1 of function 'foo' may
change at every call.
```

```
Compile-time recursion limit reached. Value of input #1 of function 'foo' may
change at every call.
```

Cause

With compile-time recursion, the code generator produces multiple versions of the recursive function instead of producing a recursive function in the generated code. These versions are known as function specializations. The code generator is unable to use compile-time recursion for a recursive function in your MATLAB code because the number of function specializations exceeds the limit.

Solutions

To address the issue, try one of these solutions:

- “Force Run-Time Recursion” on page 30-7
- “Increase the Compile-Time Recursion Limit” on page 30-10

Force Run-Time Recursion

- For this message:

  ```
  Compile-time recursion limit reached. Value of input #1 of function 'foo' may
  change at every call.
  ```

  Use this solution:

  “Force Run-Time Recursion by Treating the Input Value as Nonconstant” on page 30-8.

- For this message:

  ```
  Compile-time recursion limit reached. Size of input #1 of function 'foo' may
  change at every call.
  ```

  Use this solution:

  “Force Run-Time Recursion by Making the Input Variable-Size” on page 30-9.
- For this message:

  ```
  Compile-time recursion limit reached. Size or type of input #1 of function 'foo'
  may change at every call.
  ```

  In the code generation report, look at the function specializations. If you can see that the size of an argument is changing for each function specialization, then try this solution:

  “Force Run-Time Recursion by Making the Input Variable-Size” on page 30-9.

Force Run-Time Recursion by Treating the Input Value as Nonconstant

Consider this function:

```
function y = call_recfcn(n)
A = ones(1,n);
x = 100;
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

The second input to `recfcn` has the constant value 100. The code generator determines that the number of recursive calls is finite and tries to produce 100 copies of `recfcn`. This number of specializations exceeds the compile-time recursion limit. To force run-time recursion, instruct the code generator to treat the second input as a nonconstant value by using `coder.ignoreConst`.

```
function y = call_recfcn(n)
A = ones(1,n);
x = coder.ignoreConst(100);
y = recfcn(A,x);
end

function y = recfcn(A,x)
if size(A,2) == 1 || x == 1
    y = A(1);
else
    y = A(1)+recfcn(A(2:end),x-1);
end
end
```

If the code generator cannot determine that the number of recursive calls is finite, it produces a run-time recursive function.

Force Run-Time Recursion by Making the Input Variable-Size

Consider this function:

```
function z = call_mysum(A)
%#codegen
z = mysum(A);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

If the input to `mysum` is fixed-size, the code generator uses compile-time recursion. If `A` is large enough, the number of function specializations exceeds the compile-time limit. To cause the code generator to use run-time conversion, make the input to `mysum` variable-size by using `coder.varsize`.

```
function z = call_mysum(A)
%#codegen
B = A;
coder.varsize('B');
z = mysum(B);
```

```
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

Increase the Compile-Time Recursion Limit

The default compile-time recursion limit of 50 is large enough for most recursive functions that require compile-time recursion. Usually, increasing the limit does not fix the issue. However, if you can determine the number of recursive calls and you want compile-time recursion, increase the limit. For example, consider this function:

```
function z = call_mysum()
%#codegen
B = 1:125;
z = mysum(B);
end

function y = mysum(A)
coder.inline('never');
if size(A,2) == 1
    y = A(1);
else
    y = A(1)+ mysum(A(2:end));
end
end
```

You can determine that the code generator produces 125 copies of the `mysum` function. In this case, if you want compile-time recursion, increase the compile-time recursion limit to 125.

To increase the compile-time recursion limit:

- At the command line, in a code generation configuration object, increase the value of the `CompileTimeRecursionLimit` configuration parameter.
- In the MATLAB Coder app, increase the value of the **Compile-time recursion limit** setting.

More About

- “Code Generation for Recursive Functions” on page 13-18
- “Configure Build Settings” on page 20-26

Unable to Determine That Every Element of Cell Array Is Assigned

Issue

You see one of these messages:

```
Unable to determine that every element of 'y' is assigned before this line.
```

```
Unable to determine that every element of 'y' is assigned before exiting the
function.
```

```
Unable to determine that every element of 'y' is assigned before exiting the
recursively called function.
```

Cause

For code generation, before you use a cell array element, you must assign a value to it. When you use `cell` to create a variable-size cell array, for example, `cell(1,n)`, MATLAB assigns an empty matrix to each element. However, for code generation, the elements are unassigned. For code generation, after you use `cell` to create a variable-size cell array, you must assign all elements of the cell array before any use of the cell array.

The code generator analyzes your code to determine whether all elements are assigned before the first use of the cell array. The code generator detects that all elements are assigned when the code follows this pattern:

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

Here is the pattern for a multidimensional cell array:

```
function z = mycell(m,n,p)
%#codegen
x = cell(m,n,p);
for i = 1:m
    for j =1:n
```

```
        for k = 1:p
            x{i,j,k} = i+j+k;
        end
    end
end
z = x{m,n,p};
end
```

If the code generator detects that some elements are not assigned, code generation fails. Sometimes, even though your code assigns all elements of the cell array, code generation fails because the analysis does not detect that all elements are assigned.

Here are examples where the code generator is unable to detect that elements are assigned:

- Elements are assigned in different loops

  ```
  ...
  x = cell(1,n)
  for i = 1:5
      x{i} = 5;
  end
  for i = 6:n
      x{i} = 7;
  end
  ...
  ```

- The variable that defines the loop end value is not the same as the variable that defines the cell dimension.

  ```
  ...
  x = cell(1,n);
  m = n;
  for i = 1:m
      x{i} = 2;
  end
  ...
  ```

For more information, see “Definition of Variable-Size Cell Array by Using cell” on page 8-11.

Solution

Try one of these solutions:

- "Use recognized pattern for assigning elements" on page 30-14
- "Use repmat" on page 30-14
- "Use coder.nullcopy" on page 30-15

Use recognized pattern for assigning elements

If possible, rewrite your code to follow this pattern:

```
...
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
...
```

Use `repmat`

Sometimes, you can use `repmat` to define the variable-size cell array.

Consider this code that defines a variable-size cell array. It assigns the value 1 to odd elements and the value 2 to even elements.

```
function z = mycell2(n, j)
%#codegen
c =cell(1,n);
for i = 1:2:n-1
    c{i} = 1;
end
for i = 2:2:n
    c{i} = 2;
end
z = c{j};
```

Code generation does not allow this code because:

- More than one loop assigns the elements.
- The loop counter does not increment by 1.

Rewrite the code to first use `cell` to create a 1-by-2 cell array whose first element is 1 and whose second element is 2. Then, use `repmat` to create a variable-size cell array whose element values alternate between 1 and 2.

```
function z = mycell2(n, j)
%#codegen
c = cell(1,2);
c{1} = 1;
c{2} = 2;
c1= repmat(c,1,n);
z = c1{j};
end
```

Use `coder.nullcopy`

As a last resort, you can use `coder.nullcopy` to indicate that the code generator can allocate the memory for your cell array without initializing the memory. For example:

```
function z = mycell3(n, j)
%#codegen
c =cell(1,n);
c1 = coder.nullcopy(c);
for i = 1:4
    c1{i} = 1;
end
for i = 5:n
    c1{i} = 2;
end
z = c1{j};
end
```

Use `coder.nullcopy` with caution. If you access uninitialized memory, results are unpredictable.

See Also

`cell` | `coder.nullcopy` | `repmat`

More About

- "Cell Array Limitations for Code Generation" on page 8-10

Nonconstant Index into `vargin` or `vargout` in a `for`-Loop

Issue

Your MATLAB code contains a `for`-loop that indexes into `varargin` or `varagout`. When you generate code, you see this error message:

```
Non-constant expression or empty matrix. This expression must be constant
because its value determines the size or class of some expression.
```

Cause

At code generation time, the code generator must be able to determine the value of an index into `varargin` or `varagout`. When `varargin` or `varagout` are indexed in a `for`-loop, the code generator determines the index value for each loop iteration by unrolling the loop. Loop unrolling makes a copy of the loop body for each loop iteration. In each iteration, the code generator determines the value of the index from the loop counter.

The code generator is unable to determine the value of an index into `varargin` or `varagout` when:

- The number of copies of the loop body exceeds the limit for loop unrolling.
- Heuristics fail to identify that loop unrolling is warranted for a particular `for`-loop. For example, consider the following function:

  ```
  function [x,y,z] = fcn(a,b,c)
  %#codegen

  [x,y,z] = subfcn(a,b,c);

  function varargout = subfcn(varargin)
  j = 0;
  for i = 1:nargin
      j = j+1;
      varargout{j} = varargin{j};
  end
  ```

  The heuristics do not detect the relationship between the index `j` and the loop counter `i`. Therefore, the code generator does not unroll the `for`-loop.

Solution

Use one of these solutions:

- “Force Loop Unrolling” on page 30-17
- “Rewrite the Code” on page 30-17

Force Loop Unrolling

Force loop unrolling by using `coder.unroll`. For example:

```
function [x,y,z] = fcn(a,b,c)
%#codegen
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
j = 0;

coder.unroll();
for i = 1:nargin
    j = j + 1;
    varargout{j} = varargin{j};
end
```

Rewrite the Code

Rewrite the code so that the code generator can detect the relationship between the index and the loop counter. For example:

```
function [x,y,z] = fcn(a,b,c)
%#codegen
[x,y,z] = subfcn(a,b,c);

function varargout = subfcn(varargin)
for i = 1:nargin
    varargout{i} = varargin{i};
end
```

See Also

`coder.unroll`

More About

- “Code Generation for Variable Length Argument Lists” on page 12-2

- “Unroll `for`-Loops” on page 28-37
- “Optimization Strategies” on page 28-3