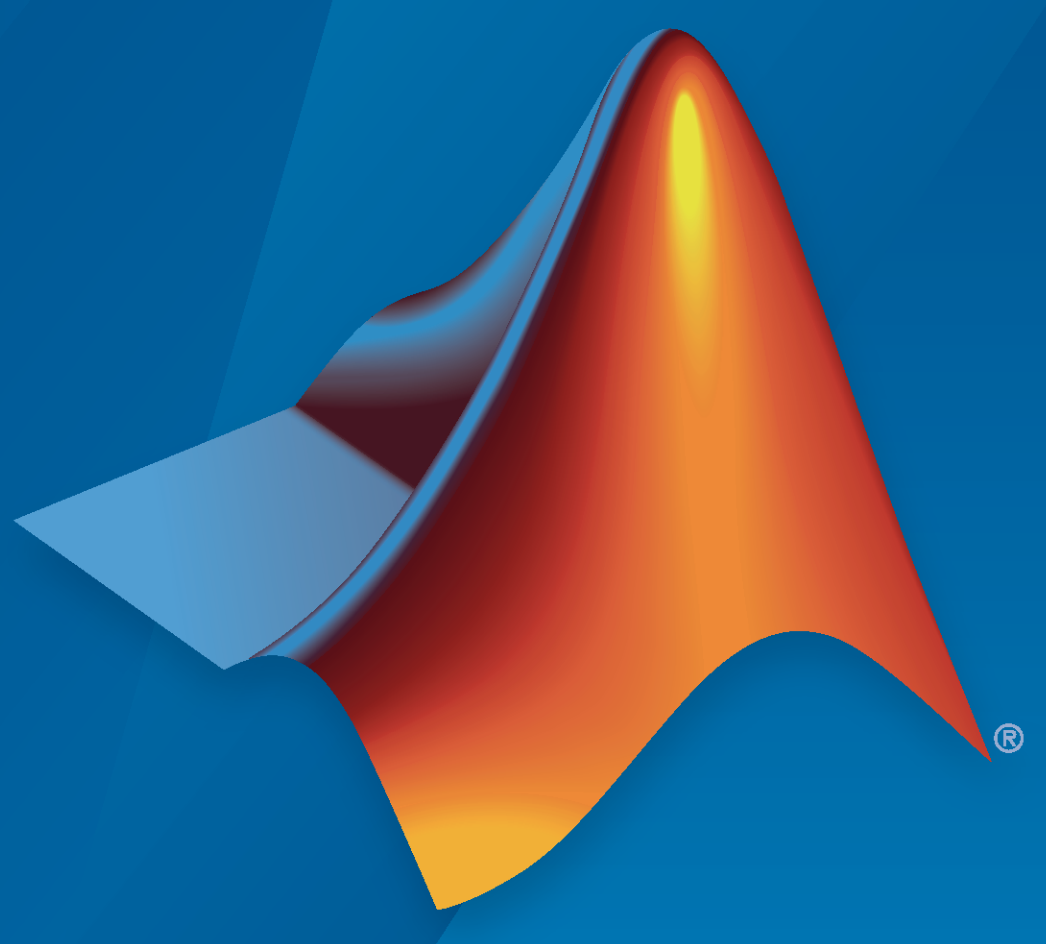
MATLAB® Coder™

User's Guide



MATLAB®

R2021a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Coder™ User's Guide

© COPYRIGHT 2011–2021 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)
September 2017	Online only	Revised for Version 3.4 (Release 2017b)
March 2018	Online only	Revised for Version 4.0 (Release 2018a)
September 2018	Online only	Revised for Version 4.1 (Release 2018b)
March 2019	Online only	Revised for Version 4.2 (Release 2019a)
September 2019	Online only	Revised for Version 4.3 (Release 2019b)
March 2020	Online only	Revised for Version 5.0 (Release 2020a)
September 2020	Online only	Revised for Version 5.1 (Release 2020b)
March 2021	Online only	Revised for Version 5.2 (Release 2021a)

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

MATLAB Coder Product Description **1-2**

Product Overview **1-3**
When to Use MATLAB Coder **1-3**
Code Generation for Embedded Software Applications **1-3**
Code Generation for Fixed-Point Algorithms **1-3**

2 Design Considerations for C/C++ Code Generation

When to Generate Code from MATLAB Algorithms **2-2**
When Not to Generate Code from MATLAB Algorithms **2-2**

Which Code Generation Feature to Use **2-3**

Prerequisites for C/C++ Code Generation from MATLAB **2-4**

MATLAB Code Design Considerations for Code Generation **2-5**
See Also **2-5**

Differences Between Generated Code and MATLAB Code **2-6**
Functions that have Multiple Possible Outputs **2-6**
Writing to ans Variable **2-7**
Logical Short-Circuiting **2-7**
Loop Index Overflow **2-8**
Indexing for Loops by Using Single Precision Operands **2-9**
Index of an Unentered for Loop **2-10**
Character Size **2-10**
Order of Evaluation in Expressions **2-10**
Name Resolution While Constructing Function Handles **2-11**
Termination Behavior **2-13**
Size of Variable-Size N-D Arrays **2-13**
Size of Empty Arrays **2-13**
Size of Empty Array That Results from Deleting Elements of an Array . . . **2-13**
Binary Element-Wise Operations with Single and Double Operands **2-14**
Floating-Point Numerical Results **2-15**
NaN and Infinity **2-15**
Negative Zero **2-15**
Code Generation Target **2-16**
MATLAB Class Property Initialization **2-16**

MATLAB Classes in Nested Property Assignments That Have Set Methods 2-16
MATLAB Handle Class Destructors 2-16
Variable-Size Data 2-17
Complex Numbers 2-17
Converting Strings with Consecutive Unary Operators to double 2-17

Potential Differences Reporting **2-18**
Addressing Potential Differences Messages 2-18
Disabling and Enabling Potential Differences Reporting 2-18

Potential Differences Messages **2-20**
Automatic Dimension Incompatibility 2-20
mtimes No Dynamic Scalar Expansion 2-20
Matrix-Matrix Indexing 2-21
Vector-Vector Indexing 2-21
Loop Index Overflow 2-22

MATLAB Language Features Supported for C/C++ Code Generation **2-24**
MATLAB Features That Code Generation Supports 2-24
MATLAB Language Features That Code Generation Does Not Support 2-25

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

Functions and Objects Supported for C/C++ Code Generation **3-2**

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

Variables Definition for Code Generation **4-2**

Best Practices for Defining Variables for C/C++ Code Generation **4-3**
Define Variables By Assignment Before Using Them 4-3
Use Caution When Reassigning Variables 4-5
Use Type Cast Operators in Variable Definitions 4-5
Define Matrices Before Assigning Indexed Variables 4-5

Eliminate Redundant Copies of Variables in Generated Code **4-6**
When Redundant Copies Occur 4-6
How to Eliminate Redundant Copies by Defining Uninitialized Variables 4-6
Defining Uninitialized Variables 4-6

Reassignment of Variable Properties **4-8**

Reuse the Same Variable with Different Properties **4-9**
When You Can Reuse the Same Variable with Different Properties 4-9

When You Cannot Reuse Variables 4-9
Limitations of Variable Reuse 4-10

Supported Variable Types 4-11

Edit and Represent Coder Type Objects and Properties 4-12
Object Properties 4-12
Legacy Representation of Coder Type Objects 4-13

5 Defining Data for Code Generation

Data Definition for Code Generation 5-2

Code Generation for Complex Data 5-3
Restrictions When Defining Complex Variables 5-3
Code Generation for Complex Data with Zero-Valued Imaginary Parts 5-3
Results of Expressions That Have Complex Operands 5-5
Results of Complex Multiplication with Nonfinite Values 5-6

Encoding of Characters in Code Generation 5-7

Array Size Restrictions for Code Generation 5-8

Code Generation for Constants in Structures and Arrays 5-9

Code Generation for Strings 5-11
Limitations 5-11
Differences Between Generated Code and MATLAB Code 5-11

Define String Scalar Inputs 5-12
Define String Scalar Types at the Command Line 5-12
Define String Scalar Inputs in the MATLAB Coder App 5-13

Code Generation for Sparse Matrices 5-14
Sparse Data Types in Generated Code 5-14
Input Definition 5-14
Code Generation Guidelines 5-15
Code Generation Limitations 5-16

Specify Array Layout in Functions and Classes 5-17
Specify Array Layout in a Function 5-17
Query Array Layout of a Function 5-18
Specify Array Layout in a Class 5-18

Code Design for Row-Major Array Layout 5-21
Understand Potential Inefficiencies Caused by Array Layout 5-21
Linear Indexing Uses Column-Major Array Layout 5-23

6 Code Generation for Variable-Size Data

Code Generation for Variable-Size Arrays **6-2**
Memory Allocation for Variable-Size Arrays **6-2**
Enabling and Disabling Support for Variable-Size Arrays **6-3**
Variable-Size Arrays in a Code Generation Report **6-3**

Control Memory Allocation for Variable-Size Arrays **6-4**
Provide Upper Bounds for Variable-Size Arrays **6-4**
Disable Dynamic Memory Allocation **6-4**
Configure Code Generator to Use Dynamic Memory Allocation for Arrays Bigger Than a Threshold **6-4**

Specify Upper Bounds for Variable-Size Arrays **6-6**
Specify Upper Bounds for Variable-Size Inputs **6-6**
Specify Upper Bounds for Local Variables **6-6**

Define Variable-Size Data for Code Generation **6-8**
Use a Matrix Constructor with Nonconstant Dimensions **6-8**
Assign Multiple Sizes to the Same Variable **6-8**
Define Variable-Size Data Explicitly by Using coder.varsize **6-9**

Diagnose and Fix Variable-Size Data Errors **6-12**
Diagnosing and Fixing Size Mismatch Errors **6-12**
Diagnosing and Fixing Errors in Detecting Upper Bounds **6-13**

Incompatibilities with MATLAB in Variable-Size Support for Code Generation **6-15**
Incompatibility with MATLAB for Scalar Expansion **6-15**
Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays **6-16**
Incompatibility with MATLAB in Determining Size of Empty Arrays **6-17**
Incompatibility with MATLAB in Determining Class of Empty Arrays **6-18**
Incompatibility with MATLAB in Matrix-Matrix Indexing **6-18**
Incompatibility with MATLAB in Vector-Vector Indexing **6-19**
Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation **6-19**
Incompatibility with MATLAB in Concatenating Variable-Size Matrices . . **6-20**
Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements **6-20**

Variable-Sizing Restrictions for Code Generation of Toolbox Functions **6-22**
Common Restrictions **6-22**
Toolbox Functions with Restrictions for Variable-Size Data **6-23**

7 Code Generation for MATLAB Structures

Structure Definition for Code Generation **7-2**

Structure Operations Allowed for Code Generation **7-3**

Define Scalar Structures for Code Generation **7-4**
Restrictions When Defining Scalar Structures by Assignment **7-4**
Adding Fields in Consistent Order on Each Control Flow Path **7-4**
Restriction on Adding New Fields After First Use **7-4**

Define Arrays of Structures for Code Generation **7-6**
Ensuring Consistency of Fields **7-6**
Using repmat to Define an Array of Structures with Consistent Field Properties **7-6**
Defining an Array of Structures by Using struct **7-6**
Defining an Array of Structures Using Concatenation **7-7**

Index Substructures and Fields **7-8**

Assign Values to Structures and Fields **7-10**

8 Code Generation for Categorical Arrays

Code Generation for Categorical Arrays **8-2**
Define Categorical Arrays for Code Generation **8-2**
Allowed Operations on Categorical Arrays **8-2**
MATLAB Toolbox Functions That Support Categorical Arrays **8-3**

Define Categorical Array Inputs **8-6**
Define Categorical Array Inputs at the Command Line **8-6**
Define Categorical Array Inputs in the MATLAB Coder App **8-6**
Representation of Categorical Arrays **8-7**

Categorical Array Limitations for Code Generation **8-9**

9 Code Generation for Cell Arrays

Code Generation for Cell Arrays **9-2**
Homogeneous vs. Heterogeneous Cell Arrays **9-2**
Controlling Whether a Cell Array Is Homogeneous or Heterogeneous **9-2**
Naming the Structure Type That Represents a Heterogeneous Cell Array in the Generated Code **9-3**
Cell Arrays in Reports **9-3**

Control Whether a Cell Array Is Variable-Size **9-5**

Define Cell Array Inputs **9-7**

Cell Array Limitations for Code Generation **9-8**
Cell Array Element Assignment **9-8**

Variable-Size Cell Arrays . . . 9-9
Definition of Variable-Size Cell Array by Using cell . . . 9-9
Cell Array Indexing . . . 9-12
Growing a Cell Array by Using {end + 1} . . . 9-12
Cell Array Contents . . . 9-13
Passing Cell Arrays to External C/C++ Functions . . . 9-13

10 Code Generation for Datetime Arrays

Code Generation for Datetime Arrays . . . **10-2**
Define Datetime Arrays for Code Generation . . . 10-2
Allowed Operations on Datetime Arrays . . . 10-2
MATLAB Toolbox Functions That Support Datetime Arrays . . . 10-3

Define Datetime Array Inputs . . . **10-5**
Define Datetime Array Inputs at the Command Line . . . 10-5
Define Datetime Array Inputs in the MATLAB Coder App . . . 10-5
Representation of Datetime Arrays . . . 10-6

Datetime Array Limitations for Code Generation . . . **10-7**

11 Code Generation for Duration Arrays

Code Generation for Duration Arrays . . . **11-2**
Define Duration Arrays for Code Generation . . . 11-2
Allowed Operations on Duration Arrays . . . 11-2
MATLAB Toolbox Functions That Support Duration Arrays . . . 11-3

Define Duration Array Inputs . . . **11-6**
Define Duration Array Inputs at the Command Line . . . 11-6
Define Duration Array Inputs in the MATLAB Coder App . . . 11-6
Representation of Duration Arrays . . . 11-7

Duration Array Limitations for Code Generation . . . **11-8**

12 Code Generation for Tables

Code Generation for Tables . . . **12-2**
Define Tables for Code Generation . . . 12-2
Allowed Operations on Tables . . . 12-2
MATLAB Toolbox Functions That Support Tables . . . 12-3

Define Table Inputs **12-5**
Define Table Inputs at the Command Line **12-5**
Define Table Inputs in the MATLAB Coder App **12-5**
Representation of Tables **12-6**

Table Limitations for Code Generation **12-8**
Creating Tables Limitations **12-8**
Modifying Tables Limitations **12-8**
Using Table Functions Limitations **12-10**

13 Code Generation for Timetables

Code Generation for Timetables **13-2**
Define Timetables for Code Generation **13-2**
Allowed Operations on Timetables **13-2**
MATLAB Toolbox Functions That Support Timetables **13-3**

Define Timetable Inputs **13-5**
Define Timetable Inputs at the Command Line **13-5**
Define Timetable Inputs in the MATLAB Coder App **13-5**
Representation of Timetables **13-6**

Timetable Limitations for Code Generation **13-8**
Creating Timetables Limitations **13-8**
Modifying Timetables Limitations **13-9**
Using Timetable Functions Limitations **13-11**

14 Code Generation for Enumerated Data

Code Generation for Enumerations **14-2**
Define Enumerations for Code Generation **14-2**
Allowed Operations on Enumerations **14-4**
MATLAB Toolbox Functions That Support Enumerations **14-5**

Customize Enumerated Types in Generated Code **14-7**
Specify a Default Enumeration Value **14-7**
Specify a Header File **14-8**
Include Class Name Prefix in Generated Enumerated Type Value Names
. . . . **14-8**

15 Code Generation for MATLAB Classes

MATLAB Classes Definition for Code Generation **15-2**
Language Limitations **15-2**
Code Generation Features Not Compatible with Classes **15-3**
Defining Class Properties for Code Generation **15-3**
Inheritance from Built-In MATLAB Classes Not Supported **15-6**

Classes That Support Code Generation **15-7**

Generate Code for MATLAB Value Classes **15-8**

Generate Code for MATLAB Handle Classes and System Objects **15-12**

Code Generation for Handle Class Destructors **15-15**
Guidelines and Restrictions **15-15**
Behavioral Differences of Objects in Generated Code and in MATLAB . . **15-16**

Class Does Not Have Property **15-18**
Solution **15-18**

Passing By Reference Not Supported for Some Properties **15-20**

Handle Object Limitations for Code Generation **15-21**
A Variable Outside a Loop Cannot Refer to a Handle Object Allocated Inside a Loop **15-21**
A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object **15-22**
References to Handle Objects Can Appear Undefined **15-23**

System Objects in MATLAB Code Generation **15-24**
Usage Rules and Limitations for System Objects for Generating Code . . **15-24**
System Objects in codegen **15-26**
System Objects in the MATLAB Function Block **15-26**
System Objects in the MATLAB System Block **15-26**
System Objects and MATLAB Compiler Software **15-26**

Specify Objects as Inputs at the Command Line **15-27**
Consistency Between coder.ClassType Object and Class Definition File **15-28**
Limitations for Using Objects as Entry-Point Function Inputs **15-28**

Specify Objects as Inputs in the MATLAB Coder App **15-30**
Automatically Define an Object Input Type **15-30**
Provide an Example **15-30**
Consistency Between the Type Definition and Class Definition File **15-31**
Limitations for Using Objects as Entry-Point Function Inputs **15-31**

16 Generating C++ Classes

Generate C++ Classes for MATLAB Classes **16-2**
Example: Generate Code for a Handle Class That Has Private and Public Members **16-2**
Additional Usage Notes and Limitations **16-5**

17 Code Generation for Function Handles

Function Handle Limitations for Code Generation **17-2**

18 Code Generation for Deep Learning Arrays

Code Generation for dlarray **18-2**
Define dlarray for Code Generation **18-2**
dlarray Object Functions with Code Generation Support **18-3**
Deep Learning Toolbox Functions with dlarray Code Generation Support **18-4**
MATLAB Functions with dlarray Code Generation Support **18-4**

dlarray Limitations for Code Generation **18-8**
Recommended Usage **18-8**
Limitations **18-8**

19 Defining Functions for Code Generation

Code Generation for Variable Length Argument Lists **19-2**

Specify Number of Entry-Point Function Input or Output Arguments to Generate **19-3**
Control Number of Input Arguments **19-3**
Control the Number of Output Arguments **19-4**

Code Generation for Anonymous Functions **19-6**
Anonymous Function Limitations for Code Generation **19-6**

Code Generation for Nested Functions **19-7**
Nested Function Limitations for Code Generation **19-7**

20 Calling Functions for Code Generation

Resolution of Function Calls for Code Generation **20-2**
Key Points About Resolving Function Calls **20-4**
Compile Path Search Order **20-4**
When to Use the Code Generation Path **20-4**

Resolution of File Types on Code Generation Path **20-5**

Compilation Directive %#codegen **20-7**

Use MATLAB Engine to Execute a Function Call in Generated Code . . . **20-8**
When To Declare a Function as Extrinsic **20-8**
Using the coder.extrinsic Construct **20-9**
Calling MATLAB Functions Using feval **20-11**
Working with mxArrays **20-11**
Restrictions on Using Extrinsic Functions **20-12**

Code Generation for Recursive Functions **20-14**
Compile-Time Recursion **20-14**
Run-Time Recursion **20-15**
Disallow Recursion **20-15**
Disable Run-Time Recursion **20-15**
Recursive Function Limitations for Code Generation **20-16**

Force Code Generator to Use Run-Time Recursion **20-17**
Treat the Input to the Recursive Function as a Nonconstant **20-17**
Make the Input to the Recursive Function Variable-Size **20-18**
Assign Output Variable Before the Recursive Call **20-19**

Avoid Duplicate Functions in Generated Code **20-20**
Issue **20-20**
Cause **20-20**
Solution **20-20**

21 Fixed-Point Conversion

Detect Dead and Constant-Folded Code **21-2**
What Is Dead Code? **21-2**
Detect Dead Code **21-2**
Fix Dead Code **21-3**

Convert MATLAB Code to Fixed-Point C Code **21-5**

Propose Fixed-Point Data Types Based on Simulation Ranges **21-6**

Propose Fixed-Point Data Types Based on Derived Ranges **21-17**

Specify Type Proposal Options **21-29**

Detect Overflows **21-32**

Replace the exp Function with a Lookup Table **21-40**

Replace a Custom Function with a Lookup Table **21-47**

Enable Plotting Using the Simulation Data Inspector **21-53**

Visualize Differences Between Floating-Point and Fixed-Point Results **21-54**

View and Modify Variable Information **21-64**
View Variable Information **21-64**
Modify Variable Information **21-64**
Revert Changes **21-65**
Promote Sim Min and Sim Max Values **21-65**

Automated Fixed-Point Conversion **21-67**
Automated Fixed-Point Conversion Capabilities **21-67**
Code Coverage **21-67**
Proposing Data Types **21-70**
Locking Proposed Data Types **21-71**
Viewing Functions **21-72**
Viewing Variables **21-79**
Log Data for Histogram **21-81**
Function Replacements **21-83**
Validating Types **21-83**
Testing Numerics **21-84**
Detecting Overflows **21-84**

Convert Fixed-Point Conversion Project to MATLAB Scripts **21-85**

Generated Fixed-Point Code **21-87**
Location of Generated Fixed-Point Files **21-87**
Minimizing fi-casts to Improve Code Readability **21-87**
Avoiding Overflows in the Generated Fixed-Point Code **21-88**
Controlling Bit Growth **21-88**
Avoiding Loss of Range or Precision **21-89**
Handling Non-Constant mpower Exponents **21-90**

Fixed-Point Code for MATLAB Classes **21-92**
Automated Conversion Support for MATLAB Classes **21-92**
Unsupported Constructs **21-92**
Coding Style Best Practices **21-92**

Automated Fixed-Point Conversion Best Practices **21-94**
Create a Test File **21-94**
Prepare Your Algorithm for Code Acceleration or Code Generation **21-95**
Check for Fixed-Point Support for Functions Used in Your Algorithm . . **21-95**
Manage Data Types and Control Bit Growth **21-96**
Convert to Fixed Point **21-96**
Use the Histogram to Fine-Tune Data Type Settings **21-97**
Optimize Your Algorithm **21-97**
Avoid Explicit Double and Single Casts **21-99**

Replacing Functions Using Lookup Table Approximations **21-100**

MATLAB Language Features Supported for Automated Fixed-Point Conversion **21-101**
MATLAB Language Features Supported for Automated Fixed-Point Conversion **21-101**
MATLAB Language Features Not Supported for Automated Fixed-Point Conversion **21-102**

Inspecting Data Using the Simulation Data Inspector **21-103**
What Is the Simulation Data Inspector? **21-103**
Import Logged Data **21-103**
Export Logged Data **21-103**
Group Signals **21-103**
Run Options **21-103**
Create Report **21-104**
Comparison Options **21-104**
Enabling Plotting Using the Simulation Data Inspector **21-104**
Save and Load Simulation Data Inspector Sessions **21-104**

Custom Plot Functions **21-105**

Data Type Issues in Generated Code **21-106**
Enable the Highlight Option in the MATLAB Coder App **21-106**
Enable the Highlight Option at the Command Line **21-106**
Stowaway Doubles **21-106**
Stowaway Singles **21-106**
Expensive Fixed-Point Operations **21-106**

22 Automated Fixed-Point Conversion Using Programmatic Workflow

Convert MATLAB Code to Fixed-Point C Code **22-2**

Propose Fixed-Point Data Types Based on Simulation Ranges **22-4**

Propose Fixed-Point Data Types Based on Derived Ranges **22-9**

Detect Overflows **22-16**

Replace the exp Function with a Lookup Table **22-19**

Replace a Custom Function with a Lookup Table **22-21**

Enable Plotting Using the Simulation Data Inspector **22-23**

Visualize Differences Between Floating-Point and Fixed-Point Results **22-24**

Generate Single-Precision C Code at the Command Line **23-2**
Prerequisites **23-2**
Create a Folder and Copy Relevant Files **23-2**
Determine the Type of the Input Argument **23-4**
Generate and Run Single-Precision MEX to Verify Numerical Behavior **23-4**
Generate Single-Precision C Code **23-4**
View the Generated Single-Precision C Code **23-4**
View Potential Data Type Issues **23-5**

Generate Single-Precision C Code Using the MATLAB Coder App **23-6**
Prerequisites **23-6**
Create a Folder and Copy Relevant Files **23-6**
Open the MATLAB Coder App **23-8**
Select the Source Files **23-8**
Enable Single-Precision Conversion **23-8**
Define Input Types **23-9**
Check for Run-Time Issues **23-9**
Generate Single-Precision C Code **23-10**
View the Generated C Code **23-10**
View Potential Data Type Issues **23-10**

Generate Single-Precision MATLAB Code **23-11**
Prerequisites **23-11**
Create a Folder and Copy Relevant Files **23-11**
Set Up the Single-Precision Configuration Object **23-12**
Generate Single-Precision MATLAB Code **23-13**
View the Type Proposal Report **23-13**
View Generated Single-Precision MATLAB Code **23-14**
View Potential Data Type Issues **23-14**
Compare the Double-Precision and Single-Precision Variables **23-15**
Optionally Generate Single-Precision C Code **23-16**

Choose a Single-Precision Conversion Workflow **23-18**

Single-Precision Conversion Best Practices **23-19**
Use Integers for Index Variables **23-19**
Limit Use of assert Statements **23-19**
Initialize MATLAB Class Properties in Constructor **23-19**
Provide a Test File That Calls Your MATLAB Function **23-19**
Prepare Your Code for Code Generation **23-20**
Verify Double-Precision Code Before Single-Precision Conversion **23-20**
Best Practices for Generation of Single-Precision C/C++ Code **23-20**
Best Practices for Generation of Single-Precision MATLAB Code **23-21**

Warnings from Conversion to Single-Precision C/C++ Code **23-22**
Function Uses Double-Precision in the C89/C90 Standard **23-22**
Built-In Function Is Implemented in Double-Precision **23-22**
Built-In Function Returns Double-Precision **23-23**

Combining Integers and Double-Precision Numbers **23-24**

MATLAB Language Features Supported for Single-Precision Conversion **23-25**
MATLAB Language Features Supported for Single-Precision Conversion **23-25**
MATLAB Language Features Not Supported for Single-Precision Conversion **23-26**

24 Setting Up a MATLAB Coder Project

Set Up a MATLAB Coder Project **24-2**
Create a Project **24-2**
Open an Existing Project **24-2**

Specify Properties of Entry-Point Function Inputs Using the App **24-3**
Why Specify Input Properties? **24-3**
Specify an Input Definition Using the App **24-3**

Automatically Define Input Types by Using the App **24-4**

Make Dimensions Variable-Size When They Meet Size Threshold **24-5**

Define Input Parameter by Example by Using the App **24-6**
Define an Input Parameter by Example **24-6**
Specify Input Parameters by Example **24-7**
Specify a String Scalar Input Parameter by Example **24-8**
Specify a Structure Type Input Parameter by Example **24-8**
Specify a Cell Array Type Input Parameter by Example **24-9**
Specify an Enumerated Type Input Parameter by Example **24-10**
Specify an Object Input Type Parameter by Example **24-11**
Specify a Fixed-Point Input Parameter by Example **24-12**
Specify an Input from an Entry-Point Function Output Type **24-13**

Define or Edit Input Parameter Type by Using the App **24-14**
Define or Edit an Input Parameter Type **24-14**
Specify a String Scalar Input Parameter **24-15**
Specify an Enumerated Type Input Parameter **24-15**
Specify a Fixed-Point Input Parameter **24-16**
Specify a Structure Input Parameter **24-16**
Specify a Cell Array Input Parameter **24-18**

Define Constant Input Parameters Using the App **24-23**

Define Inputs Programmatically in the MATLAB File **24-24**

Add Global Variables by Using the App **24-25**

Specify Global Variable Type and Initial Value Using the App **24-26**
Why Specify a Type Definition for Global Variables? **24-26**
Specify a Global Variable Type **24-26**
Define a Global Variable by Example **24-26**
Define or Edit Global Variable Type **24-27**

Define Global Variable Initial Value 24-27
Define Global Variable Constant Value 24-28
Remove Global Variables 24-28

Undo and Redo Changes to Type Definitions in the App **24-29**

Code Generation Readiness Screening in the MATLAB Coder App **24-30**

Slow Operations in MATLAB Coder App **24-31**

Unable to Open a MATLAB Coder Project **24-32**

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

Workflow for Preparing MATLAB Code for Code Generation **25-2**
See Also 25-2

Fixing Errors Detected at Design Time **25-3**
See Also 25-3

Using the Code Analyzer **25-4**

Check Code with the Code Analyzer **25-5**

Check Code by Using the Code Generation Readiness Tool **25-7**
Run Code Generation Readiness Tool at the Command Line 25-7
Run Code Generation Readiness Tool from the Current Folder Browser 25-7
Run the Code Generation Readiness Tool Using the MATLAB Coder App 25-7

Code Generation Readiness Tool **25-8**
Summary Tab 25-9
Code Structure Tab 25-10

Unable to Determine Code Generation Readiness **25-13**

Generate MEX Functions by Using the MATLAB Coder App **25-14**
Workflow for Generating MEX Functions Using the MATLAB Coder App 25-14
Generate a MEX Function Using the MATLAB Coder App 25-14
Configure Project Settings 25-16
Build a MATLAB Coder Project 25-16
See Also 25-17

Generate MEX Functions at the Command Line **25-18**
Command-line Workflow for Generating MEX Functions 25-18
Generate a MEX Function at the Command Line 25-18

Fix Errors Detected at Code Generation Time **25-19**
See Also 25-19

Running MEX Functions 25-20
Debug MEX Functions 25-20
Debug MEX Functions by Using a C/C++ Debugger 25-20

Debugging Strategies 25-21

Collect and View Line Execution Counts for Your MATLAB Code 25-22

26 Testing MEX Functions in MATLAB

Why Test MEX Functions in MATLAB? 26-2

Workflow for Testing MEX Functions in MATLAB 26-3
See Also 26-3

Running MEX Functions 26-4
Debug MEX Functions 26-4
Debug MEX Functions by Using a C/C++ Debugger 26-4

Check for Run-Time Issues by Using the App 26-5
Collect MATLAB Line Execution Counts 26-5
Disable JIT Compilation for Parallel Loops 26-5

Verify MEX Functions in the MATLAB Coder App 26-7

Verify MEX Functions at the Command Line 26-8

Debug Run-Time Errors 26-9
Viewing Errors in the Run-Time Stack 26-9
Handling Run-Time Errors 26-10

Using MEX Functions That MATLAB Coder Generates 26-11

27 Generating C/C++ Code from MATLAB Code

Code Generation Workflow 27-3
See Also 27-3

Generating Standalone C/C++ Executables from MATLAB Code 27-4
Generate a C Executable Using the MATLAB Coder App 27-4
Generate a C Executable at the Command Line 27-10
Specifying main Functions for C/C++ Executables 27-11
Specify main Functions 27-11

Configure Build Settings 27-13
Specify Build Type 27-13
Specify a Language for Code Generation 27-15

Specify Output File Name **27-16**
Specify Output File Locations **27-16**
Parameter Specification Methods **27-17**
Specify Build Configuration Parameters **27-17**

Specify Configuration Parameters in Command Line Workflow Interactively **27-22**
Using the Dialog Box **27-22**
Additional Functionalities in the Dialog Box **27-22**

Specify Data Types Used in Generated Code **27-24**
Specify Data Type Using the MATLAB Coder App **27-24**
Specify Data Type at the Command Line **27-24**

Use Generated Initialize and Terminate Functions **27-25**
Initialize Function **27-25**
Terminate Function **27-27**

Change the Standard Math Library **27-29**

Convert codegen Command to Equivalent MATLAB Coder Project **27-30**
Example: Convert a Complete codegen Command to a Project File **27-30**
Example: Convert an Incomplete codegen Command to a Template Project File **27-31**
Limitations **27-31**

Share Build Configuration Settings **27-33**
Export Settings **27-33**
Import Settings **27-34**

Convert MATLAB Coder Project to MATLAB Script **27-35**
Convert a Project Using the MATLAB Coder App **27-35**
Convert a Project Using the Command-Line Interface **27-35**
Run the Script **27-35**
Special Cases That Generate Additional MAT-File **27-36**

Preserve Variable Names in Generated Code **27-38**

Reserved Keywords **27-39**
C Reserved Keywords **27-39**
C++ Reserved Keywords **27-39**
Keywords Reserved for Code Generation **27-40**
Reserved Prefixes **27-41**
MATLAB Coder Code Replacement Library Keywords **27-41**

Specify Properties of Entry-Point Function Inputs **27-43**
Why You Must Specify Input Properties **27-43**
Properties to Specify **27-43**
Rules for Specifying Properties of Primary Inputs **27-46**
Methods for Defining Properties of Primary Inputs **27-46**
Define Input Properties by Example at the Command Line **27-47**
Specify Constant Inputs at the Command Line **27-49**
Specify Variable-Size Inputs at the Command Line **27-49**

Specify Cell Array Inputs at the Command Line **27-52**
Specify Cell Array Inputs by Example **27-52**
Specify the Type of the Cell Array Input **27-53**
Make a Homogeneous Copy of a Type **27-53**
Make a Heterogeneous Copy of a Type **27-54**
Specify Variable-Size Cell Array Inputs **27-55**
Specify Type Name for Heterogeneous Cell Array Inputs **27-56**
Specify Constant Cell Array Inputs **27-56**

Constant Input Checking in MEX Functions **27-57**
Control Whether a MEX Function Checks the Value of a Constant Input **27-58**

Define Input Properties Programmatically in the MATLAB File **27-60**
How to Use assert with MATLAB Coder **27-60**
Rules for Using assert Function **27-64**
Specifying General Properties of Primary Inputs **27-65**
Specifying Properties of Primary Fixed-Point Inputs **27-66**
Specifying Properties of Cell Arrays **27-66**
Specifying Class and Size of Scalar Structure **27-67**
Specifying Class and Size of Structure Array **27-68**

Create and Edit Input Types by Using the Coder Type Editor **27-69**
Open the Coder Type Editor **27-69**
Common Editor Actions **27-69**
Type Browser Pane **27-70**
Type Properties Pane **27-71**
MATLAB Code Pane **27-72**

Speed Up Compilation by Generating Only Code **27-74**

Disable Creation of the Code Generation Report **27-75**

Paths and File Infrastructure Setup **27-76**
Compile Path Search Order **27-76**
Specify Folders to Search for Custom Code **27-76**
Naming Conventions **27-76**

Generate Code for Multiple Entry-Point Functions **27-78**
Generating Code for Multiple Entry-Point Functions **27-78**
Call a Single Entry-Point Function from a MEX Function **27-79**
Generate Code for More Than One Entry-Point Function Using the MATLAB Coder App **27-79**

Generate One MEX Function for Multiple Signatures **27-82**
Generate Multisignature MEX Function for a Single Entry-Point Function **27-82**
Generate Multisignature MEX Function for Multiple Entry-Point Functions **27-83**

Pass an Entry-Point Function Output as an Input **27-85**
Pass an Entry-Point Function Output as an Input to Another Entry-Point Function **27-85**
Use coder.OutputType to Facilitate Code Componentization **27-86**

Generate Code for Global Data 27-88
Workflow 27-88
Declare Global Variables 27-88
Define Global Data 27-88
Synchronizing Global Data with MATLAB 27-90
Define Constant Global Data 27-92
Global Data Limitations for Generated Code 27-94

Specify Global Cell Arrays at the Command Line 27-96

Generate Code for Enumerations 27-97

Generate Code for Variable-Size Data 27-98
Disable Support for Variable-Size Data 27-98
Control Dynamic Memory Allocation 27-98
Generating Code for MATLAB Functions with Variable-Size Data 27-100
Generate Code for a MATLAB Function That Expands a Vector in a Loop 27-101

How MATLAB Coder Partitions Generated Code 27-106
Partitioning Generated Files 27-106
How to Select the File Partitioning Method 27-106
Partitioning Generated Files with One C/C++ File Per MATLAB File . . 27-106
Generated Files and Locations 27-110
File Partitioning and Inlining 27-112

Requirements for Signed Integer Representation 27-115

Build Process Customization 27-116
RTW.BuildInfo Methods 27-116
coder.updateBuildInfo Function 27-117
coder.ExternalDependency Class 27-117
Post-Code-Generation Command 27-117

Run-time Stack Overflow 27-119

Compiler and Linker Errors 27-120
Failure to Specify a Main Function 27-120
Failure to Specify External Code Files 27-120
Errors Caused by External Code 27-121

Pass Structure Arguments by Reference or by Value in Generated Code 27-122

Name the C Structure Type to Use With a Global Structure Variable 27-129

Generate Code for an LED Control Function That Uses Enumerated Types 27-131

Generate Code That Uses N-Dimensional Indexing 27-133
Improve Readability with N-Dimensional Indexing and Row-Major Layout 27-133
Column-Major Layout and N-Dimensional Indexing 27-134
Other Code Generation Considerations 27-135

Install OpenMP Library on macOS Platform **27-137**

Generate Code to Detect Edges on Images **27-138**

C Code Generation for a MATLAB Kalman Filtering Algorithm **27-142**

Generate Code to Optimize Portfolio by Using Black Litterman Approach **27-151**

Generate Code for Persistent Variables **27-159**

Generate Code for Structure Arrays **27-163**

Add Custom Toolchains to MATLAB® Coder™ Build Process **27-165**

Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type **27-174**

Build Process Support for Folder Names with Spaces or Special Characters **27-179**
Folder Names with Spaces **27-179**
Folder Names with Special Characters **27-181**
Troubleshooting Errors When Folder Names Have Spaces **27-181**

28 Verify Generated C/C++ Code

Tracing Generated C/C++ Code to MATLAB Source Code **28-2**
Generate Traceability Tags **28-2**
Format of Traceability Tags **28-2**
Location of Comments in Generated Code **28-2**
Traceability Tag Limitations **28-6**

Code Generation Reports **28-7**
Report Generation **28-7**
Report Location **28-8**
Errors and Warnings **28-8**
Files and Functions **28-8**
MATLAB Source **28-9**
MATLAB Variables **28-10**
Tracing Code **28-11**
Code Insights **28-11**
Additional Reports **28-12**
Report Limitations **28-12**

Access Code Generation Report Information Programmatically **28-13**
Create Report Information Object **28-13**
Example: Create Report Information Object for Successful Code Generation **28-13**
Example: Create Report Information Object for Successful Code Generation That Checks Out Toolbox Licenses **28-16**

Example: Create Report Information Object for Failed Code Generation 28-17
Inspect Code Manually 28-18
Transferring Code Configuration Objects to a New MATLAB Session . . 28-19

Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors **28-20**
Generated C Code vs. Generated C++ Code 28-20
Limitations 28-20
Example: Compare Generated C and C++ Code That Include Run-Time Checks 28-21

Example: Generate Standalone C Code That Detects and Reports Run-Time Errors **28-24**

Testing Code Generated from MATLAB Code **28-26**

Unit Test Generated Code with MATLAB Coder **28-27**

Unit Test External C Code with MATLAB Coder **28-33**

Calculate Number of Lines of Code by Using Report Information Object **28-43**

29 Code Replacement for MATLAB Code

What Is Code Replacement? **29-2**
Code Replacement Libraries 29-2
Code Replacement Terminology 29-4
Code Replacement Limitations 29-5

Choose a Code Replacement Library **29-6**
About Choosing a Code Replacement Library 29-6
Explore Available Code Replacement Libraries 29-6
Explore Code Replacement Library Contents 29-6

Replace Code Generated from MATLAB Code **29-8**

30 Custom Toolchain Registration

Custom Toolchain Registration **30-2**
What Is a Custom Toolchain? 30-2
What Is a Factory Toolchain? 30-2
What is a Toolchain Definition? 30-2
Key Terms 30-3
Typical Workflow 30-3

About coder.make.ToolchainInfo **30-5**

Create and Edit Toolchain Definition File **30-7**

Toolchain Definition File with Commentary **30-8**
Steps Involved in Writing a Toolchain Definition File **30-8**
Write a Function That Creates a ToolchainInfo Object **30-8**
Setup **30-9**
Macros **30-9**
C Compiler **30-9**
C++ Compiler **30-10**
Linker **30-10**
Archiver **30-11**
Builder **30-11**
Build Configurations **30-11**

Create and Validate ToolchainInfo Object **30-13**

Register the Custom Toolchain **30-14**

Use the Custom Toolchain **30-16**

Troubleshooting Custom Toolchain Validation **30-17**
Build Tool Command Path Incorrect **30-17**
Build Tool Not in System Path **30-17**
Tool Path Does Not Exist **30-18**
Path Incompatible with Builder or Build Tool **30-18**
Unsupported Platform **30-18**
Toolchain is Not installed **30-18**
Project or Configuration Is Using the Template Makefile **30-19**

Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers **30-20**

Build 32-bit DLL on 64-bit Windows® Platform Using MSVC Toolchain **30-21**

31 Deploying Generated Code

C Compiler Considerations for Signed Integer Overflows **31-2**

Use C Arrays in the Generated Function Interfaces **31-3**
Implementation of Arrays in the Generated C/C++ Code **31-3**
The emxArray Dynamic Data Structure Definition **31-4**
Utility Functions for Interacting with emxArray Data **31-5**
Examples **31-6**

Use Dynamically Allocated C++ Arrays in Generated Function Interfaces **31-15**
Examples of C++ Function Interfaces That Use Dynamically Allocated Arrays **31-15**

Using the coder::array Class Template . . . 31-16
Examples . . . 31-17

Use a Dynamic Library in a Microsoft Visual Studio Project . . . **31-20**

Incorporate Generated Code Using an Example Main Function . . . **31-23**
Workflow for Using an Example Main Function . . . 31-23
Control Example Main Generation Using the MATLAB Coder App . . . 31-23
Control Example Main Generation Using the Command-Line Interface . . . 31-24

Use an Example C Main in an Application . . . **31-25**
Prerequisites . . . 31-25
Create a Folder and Copy Relevant Files . . . 31-25
Run the Sobel Filter on the Image . . . 31-27
Generate and Test a MEX Function . . . 31-29
Generate an Example Main Function for sobel.m . . . 31-29
Copy the Example Main Files . . . 31-32
Modify the Generated Example Main Function . . . 31-32
Generate the Sobel Filter Application . . . 31-41
Run the Sobel Filter Application . . . 31-41
Display the Resulting Image . . . 31-42

Package Code for Other Development Environments . . . **31-43**
When to Package Code . . . 31-43
Package Generated Code Using the MATLAB Coder App . . . 31-43
Package Generated Code at the Command Line . . . 31-44
Specify packNGo Options . . . 31-45

Structure of Generated Example C/C++ Main Function . . . **31-47**
Contents of the File main.c or main.cpp . . . 31-47
Contents of the File main.h . . . 31-49

Troubleshoot Failures in Deployed Code . . . **31-51**

Using Dynamic Memory Allocation for an Atoms Simulation . . . **31-52**

Register New Hardware Devices . . . **31-58**
Specify Hardware Implementation for New Device . . . 31-58
Specify Hardware Implementation That Persists Over MATLAB Sessions . . . 31-58
Create Hardware Implementation by Modifying Existing Implementation . . . 31-59
Create Hardware Implementation by Reusing Existing Implementation . . . 31-59
Validate Hardware Device Data . . . 31-60
Export Hardware Device Data . . . 31-60
Create Alternative Identifier for Target Object . . . 31-61
Upgrade Data Definitions for Hardware Devices . . . 31-62

Deploy Generated C Code to External Hardware: Raspberry Pi Examples . . . **31-64**
Prerequisites . . . 31-64
Hardware Implementation Parameters . . . 31-65
Hello World Example . . . 31-66

Spring Mass Damper System Example 31-67

Deploy Generated Code 31-71
Main Function 31-71
Generated Function Interfaces 31-71
Executable Applications 31-72
Static and Dynamic Libraries 31-73
Generated File Structure 31-73
Code Verification 31-74
Custom Hardware Considerations 31-74
Other Deployment Strategies 31-74

32 Accelerating MATLAB Algorithms

Workflow for Accelerating MATLAB Algorithms 32-2
See Also 32-2

Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms 32-3
Accelerate Code That Dominates Execution Time 32-3
Include Loops Inside MEX Function 32-3
Avoid Generating MEX Functions from Unsupported Functions 32-4
Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time 32-4
Minimize MEX Function Calls 32-4

Accelerate MATLAB Algorithms 32-6

Modifying MATLAB Code for Acceleration 32-7
How to Modify Your MATLAB Code for Acceleration 32-7

Profile MEX Functions by Using MATLAB Profiler 32-8
MEX Profile Generation 32-8
Example 32-8
Effect of Folding Expressions on MEX Code Coverage 32-11

Control Run-Time Checks 32-12
Types of Run-Time Checks 32-12
When to Disable Run-Time Checks 32-12
How to Disable Run-Time Checks 32-13

Algorithm Acceleration Using Parallel for-Loops (parfor) 32-14
Parallel for-Loops (parfor) in Generated Code 32-14
How parfor-Loops Improve Execution Speed 32-14
When to Use parfor-Loops 32-15
When Not to Use parfor-Loops 32-15
parfor-Loop Syntax 32-15
parfor Restrictions 32-16

Control Compilation of parfor-Loops 32-18
When to Disable parfor 32-18

Reduction Assignments in parfor-Loops **32-19**
What are Reduction Assignments? **32-19**
Multiple Reductions in a parfor-Loop **32-19**

Classification of Variables in parfor-Loops **32-20**
Overview **32-20**
Sliced Variables **32-20**
Broadcast Variables **32-22**
Reduction Variables **32-22**
Temporary Variables **32-25**

Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor) **32-27**

Specify Maximum Number of Threads in parfor-Loops **32-28**

Troubleshooting parfor-Loops **32-29**
Global or Persistent Declarations in parfor-Loop **32-29**
Compiler Does Not Support OpenMP **32-29**

Generate MEX Code to Accelerate Simulation of Bouncing Balls **32-30**

Generate MEX Code to Calculate Geodesics in Curved Space-Time . . . **32-34**

Generate Accelerated MEX Code for Reverberation Using MATLAB® Classes **32-38**

Using PARFOR to Speed Up an Image Contrast Enhancement Algorithm **32-40**

Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler **32-49**

33 External Code Integration

Call C/C++ Code from MATLAB Code **33-2**
Call C Code **33-2**
Return Multiple Values from a C Function **33-3**
Pass Data by Reference **33-4**
Integrate External Code that Uses Custom Data Types **33-5**
Integrate External Code that Uses Pointers, Structures, and Arrays **33-6**

Configure Build for External C/C++ Code **33-9**
Provide External Files for Code Generation **33-9**
Configure Build from Within a Function **33-9**
Configure Build by Using the Configuration Object **33-10**
Configure Build by Using the MATLAB Coder App **33-11**

Develop Interface for External C/C++ Code **33-12**
Create a class from coder.ExternalDependency **33-12**
Best Practices for Using coder.ExternalDependency **33-13**

Mapping MATLAB Types to Types in Generated Code **33-15**
Complex Types **33-16**
Structure Types **33-16**
Fixed-Point Types **33-16**
Character Vectors **33-17**
Multiword Types **33-17**

Generate Code to Read a Text File **33-19**

Generate C/C++ Strings from MATLAB Strings and Character Row Vectors **33-27**
Add New Line to Strings in Generated Code **33-27**
Limitations **33-28**

34 Generate Efficient and Reusable Code

Optimization Strategies **34-3**

Modularize MATLAB Code **34-5**

Avoid Data Copies of Function Inputs in Generated Code **34-6**

Inline Code **34-8**

Control Inlining to Fine-Tune Performance and Readability of Generated Code **34-9**
Control Inlining of a Specific MATLAB Function **34-9**
Control Inlining by Using Code Generation Settings **34-9**
Interaction Between Different Inlining Controls **34-11**
Example: Control Inlining at the Boundary Between Your Functions and MathWorks® Functions **34-11**

Fold Function Calls into Constants **34-14**

Control Stack Space Usage **34-15**

Stack Allocation and Performance **34-18**
Allocate Heap Space from Command Line **34-18**
Allocate Heap Space Using the MATLAB Coder App **34-18**

Dynamic Memory Allocation and Performance **34-19**
When Dynamic Memory Allocation Occurs **34-19**

Minimize Dynamic Memory Allocation **34-20**

Provide Maximum Size for Variable-Size Arrays **34-21**

Disable Dynamic Memory Allocation During Code Generation **34-25**

Set Dynamic Memory Allocation Threshold **34-26**
Set Dynamic Memory Allocation Threshold Using the MATLAB Coder App **34-26**
Set Dynamic Memory Allocation Threshold at the Command Line **34-26**

Excluding Unused Paths from Generated Code **34-28**

Prevent Code Generation for Unused Execution Paths **34-29**
Prevent Code Generation When Local Variable Controls Flow **34-29**
Prevent Code Generation When Input Variable Controls Flow **34-29**

Generate Code with Parallel for-Loops (parfor) **34-31**

Minimize Redundant Operations in Loops **34-32**

Unroll for-Loops **34-33**
Force Loop Unrolling by Using coder.unroll **34-33**
Set Loop Unrolling Threshold for All for-Loops in the MATLAB Code . . . **34-34**

Disable Support for Integer Overflow or Nonfinites **34-37**
Disable Support for Integer Overflow **34-37**
Disable Support for Nonfinite Numbers **34-37**

Integrate External/Custom Code **34-39**

MATLAB Coder Optimizations in Generated Code **34-43**
Constant Folding **34-43**
Loop Fusion **34-44**
Successive Matrix Operations Combined **34-44**
Unreachable Code Elimination **34-44**
memcpy Calls **34-45**
memset Calls **34-45**

Use coder.const with Extrinsic Function Calls **34-46**
Reduce Code Generation Time by Using coder.const with feval **34-46**
Force Constant-Folding by Using coder.const with feval **34-46**

memcpy Optimization **34-48**

memset Optimization **34-49**

Reuse Large Arrays and Structures **34-50**

LAPACK Calls in Generated Code **34-51**

Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls **34-52**
Specify LAPACK Library **34-52**
Write LAPACK Callback Class **34-52**
Generate LAPACK Calls by Specifying a LAPACK Callback Class **34-53**
Locate LAPACK Library in Execution Environment **34-54**

BLAS Calls in Generated Code **34-55**

Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls 34-56
Specify BLAS Library 34-56
Write BLAS Callback Class 34-56
Generate BLAS Calls by Specifying a BLAS Callback Class 34-58
Locate BLAS Library in Execution Environment 34-58
Usage Notes and Limitations for OpenBLAS Library 34-58

Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls 34-60
Install FFTW Library 34-60
Write an FFT Callback Class 34-60
Generate FFTW Library Calls by Specifying an FFT Library Callback Class 34-61

Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code 34-63
Prerequisites 34-63
Create a MATLAB Function 34-63
Write Supporting C Code 34-64
Write an FFT Library Callback Class 34-64
Generate a Dynamically Linked Library 34-65
Specify Configuration Parameters in the MATLAB Coder App 34-66

Speed Up MEX Generation by Using JIT Compilation 34-67
Specify Use of JIT Compilation in the MATLAB Coder App 34-67
Specify Use of JIT Compilation at the Command Line 34-67
JIT Compilation Incompatibilities 34-67

Automatically Parallelize for Loops in Generated Code 34-69
Parallelize for Loops by Using MATLAB Coder App 34-69
Parallelize for Loops at Command Line 34-69
Inspect Generated Code and Code Insights 34-70
Disable Automatic Parallelization of a for Loop 34-71
Parallelize Implicit for Loops 34-71
Usage Notes and Limitations 34-72

Resolve Issue: Array or Variable Access Pattern Not Suitable for Parallel Execution 34-73
Issue 34-73
Possible Solutions 34-73

35 Generating Reentrant C Code from MATLAB Code

Generate Reentrant C Code from MATLAB Code 35-2
About This Tutorial 35-2
Copying Files Locally 35-3
About the Example 35-3
Providing a C main Function 35-4
Configuring Build Parameters 35-6
Generating the C Code 35-6

Viewing the Generated C Code . . . 35-6
Running the Code . . . 35-7
Key Points to Remember . . . 35-7
Learn More . . . 35-8

Reentrant Code . . . **35-9**

Specify Generation of Reentrant Code . . . **35-11**
Specify Generation of Reentrant Code Using the MATLAB Coder App . . . 35-11
Specify Generation of Reentrant Code Using the Command-Line Interface . . . 35-11

API for Generated Reusable Code . . . **35-12**

Call Reentrant Code in a Single-Threaded Environment . . . **35-13**

Call Reentrant Code in a Multithreaded Environment . . . **35-14**
Multithreaded Examples . . . 35-14

Call Reentrant Code with No Persistent or Global Data (UNIX Only) . . . **35-15**
Provide a Main Function . . . 35-15
Generate Reentrant C Code . . . 35-17
Examine the Generated Code . . . 35-17
Run the Code . . . 35-18

Call Reentrant Code — Multithreaded with Persistent Data (Windows Only) . . . **35-19**
MATLAB Code for This Example . . . 35-19
Provide a Main Function . . . 35-19
Generate Reentrant C Code . . . 35-22
Examine the Generated Code . . . 35-22
Run the Code . . . 35-23

Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only) . . . **35-24**
MATLAB Code for This Example . . . 35-24
Provide a Main Function . . . 35-24
Generate Reentrant C Code . . . 35-27
Examine the Generated Code . . . 35-28
Run the Code . . . 35-28

36 Troubleshooting Code Generation Problems

JIT MEX Incompatibility Warning . . . **36-2**
Issue . . . 36-2
Cause . . . 36-2
Solution . . . 36-2

JIT Compilation Does Not Support OpenMP . . . **36-3**
Issue . . . 36-3
Cause . . . 36-3

Solution 36-3

Output Variable Must Be Assigned Before Run-Time Recursive Call 36-4
Issue 36-4
Cause 36-4
Solution 36-4

Compile-Time Recursion Limit Reached 36-7
Issue 36-7
Cause 36-7
Solutions 36-7
Force Run-Time Recursion 36-7
Increase the Compile-Time Recursion Limit 36-9

Unable to Determine That Every Element of Cell Array Is Assigned 36-10
Issue 36-10
Cause 36-10
Solution 36-11

Nonconstant Index into varargin or varargout in a for-Loop 36-14
Issue 36-14
Cause 36-14
Solution 36-14

Unknown Output Type for coder.ceval 36-16
Issue 36-16
Cause 36-16
Solution 36-16

MEX Generated on macOS Platform Stays Loaded in Memory 36-18
Issue 36-18
Cause 36-18
Solution 36-18

Resolve Error: Code Generator Failed to Produce C++ Destructor for MATLAB Class 36-19
Issue 36-19
Possible Solutions 36-19

37 Row-Major Array Layout

Row-Major and Column-Major Array Layouts 37-2
Array Storage in Computer Memory 37-2
Conversions Between Different Array Layouts 37-2

Generate Code That Uses Row-Major Array Layout 37-4
Specify Row-Major Layout 37-4
Array Layout and Algorithmic Efficiency 37-5
Row-Major Layout for N-Dimensional Arrays 37-6
Specify Array Layout in External Function Calls 37-7

38 Deep Learning with MATLAB Coder

Prerequisites for Deep Learning with MATLAB Coder **38-2**
MathWorks Products **38-2**
Third-Party Hardware and Software **38-2**
Environment Variables **38-4**

Workflow for Deep Learning Code Generation with MATLAB Coder **38-7**

Networks and Layers Supported for Code Generation **38-8**
Supported Pretrained Networks **38-8**
Supported Layers **38-9**
Supported Classes **38-17**

Load Pretrained Networks for Code Generation **38-23**
Load a Network by Using coder.loadDeepLearningNetwork **38-23**
Specify a Network Object for Code Generation **38-23**
Specify a dlnetwork Object for Code Generation **38-24**

Generate Generic C/C++ Code for Deep Learning Networks **38-26**
Requirements **38-26**
Code Generation by Using codegen **38-26**
Code Generation by Using the MATLAB Coder App **38-27**

Code Generation for Deep Learning Networks with MKL-DNN **38-29**
Requirements **38-29**
Code Generation by Using codegen **38-29**
Code Generation by Using the MATLAB Coder App **38-30**

Code Generation for Deep Learning Networks with ARM Compute Library
. . . . **38-32**
Requirements **38-32**
Code Generation by Using codegen **38-32**
Code Generation by Using the MATLAB Coder App **38-35**

Cross-Compile Deep Learning Code That Uses ARM Compute Library
. . . . **38-37**
Prerequisites **38-37**
Generate and Deploy Deep Learning Code **38-38**

Code Generation for Quantized Deep Learning Networks **38-40**
Supported Layers and Classes **38-40**
Generating Code **38-40**

Deep Learning Code Generation on Intel Targets for Different Batch Sizes
. . . . **38-42**

Deep Learning Prediction with ARM Compute Using codegen **38-51**

Code Generation for Deep Learning on ARM Targets **38-56**

Generate C++ Code for Object Detection Using YOLO v2 and Intel MKL-DNN **38-61**

Code Generation and Deployment of MobileNet-v2 Network to Raspberry Pi **38-64**

Code Generation for Semantic Segmentation Application on Intel CPUs That Uses U-Net **38-68**

Code Generation for Semantic Segmentation Application on ARM® Neon targets That Uses U-Net **38-77**

Code Generation for LSTM Network on Raspberry Pi **38-86**

Code Generation for LSTM Network That Uses Intel MKL-DNN **38-93**

Code Generation for Convolutional LSTM Network That Uses Intel MKL-DNN **38-97**

Cross Compile Deep Learning Code for ARM Neon Targets **38-101**

Code Generation for Quantized Deep Learning Network on Raspberry Pi **38-107**

Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning **38-115**

Generate Digit Images Using Variational Autoencoder on Intel CPUs **38-124**

39 Generating Code for C++

C++ Code Generation **39-2**
- Generate C++ Code **39-2**
- C++ Language Features Supported in Generated Code **39-2**
- Additional Differences Between Generated C Code and C++ Code **39-3**

Generate C++ Code with Class Interface **39-4**
- Generate C++ Code with a Class Interface **39-4**
- Globals and Persistents in a Generated C++ Class **39-6**
- Put Multiple Entry-Point Functions in the Same Class **39-7**

Organize Generated C++ Code into Namespaces **39-9**
- Settings That Control Namespace Structure **39-9**
- Example: Generate C++ Code with Namespaces **39-10**

Integrate Multiple Generated C++ Code Projects **39-14**

Generate C++ Classes for MATLAB® Classes That Model Simple and Damped Oscillators **39-18**

View Data in the Simulation Data Inspector **40-2**
View Logged Data **40-2**
Import Data from the Workspace or a File **40-3**
View Complex Data **40-5**
View String Data **40-6**
View Frame-Based Data **40-9**
View Event-Based Data **40-9**

Import Data from a CSV File into the Simulation Data Inspector **40-11**
Basic File Format **40-11**
Multiple Time Vectors **40-11**
Signal Metadata **40-12**
Import Data from a CSV File **40-13**

Microsoft Excel Import, Export, and Logging Format **40-16**
Basic File Format **40-16**
Multiple Time Vectors **40-16**
Signal Metadata **40-17**
User-Defined Data Types **40-19**
Complex, Multidimensional, and Bus Signals **40-21**
Function-Call Signals **40-21**
Simulation Parameters **40-22**
Multiple Runs **40-22**

Configure the Simulation Data Inspector **40-24**
Logged Data Size and Location **40-24**
Archive Behavior and Run Limit **40-25**
Incoming Run Names and Location **40-26**
Signal Metadata to Display **40-27**
Signal Selection on the Inspect Pane **40-27**
How Signals Are Aligned for Comparison **40-28**
Colors Used to Display Comparison Results **40-28**
Signal Grouping **40-29**
Data to Stream from Parallel Simulations **40-29**
Options for Saving and Loading Session Files **40-30**
Signal Display Units **40-30**

How the Simulation Data Inspector Compares Data **40-32**
Signal Alignment **40-32**
Synchronization **40-33**
Interpolation **40-34**
Tolerance Specification **40-34**
Limitations **40-36**

Save and Share Simulation Data Inspector Data and Views **40-37**
Save and Load Simulation Data Inspector Sessions **40-37**
Share Simulation Data Inspector Views **40-38**
Share Simulation Data Inspector Plots **40-38**
Create a Simulation Data Inspector Report **40-39**
Export Data from the Simulation Data Inspector **40-40**

Inspect and Compare Data Programmatically **40-42**
Create a Run and View the Data **40-42**
Compare Two Signals in the Same Run **40-43**
Compare Runs with Global Tolerance **40-44**
Analyze Simulation Data Using Signal Tolerances **40-45**

Limit the Size of Logged Data **40-47**
Limit the Number of Runs Retained in the Simulation Data Inspector Archive **40-47**
Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data **40-47**
View Data Only During Simulation **40-48**
Reduce the Number of Data Points Logged from Simulation **40-48**

About MATLAB Coder

- “MATLAB Coder Product Description” on page 1-2
- “Product Overview” on page 1-3

MATLAB Coder Product Description

Generate C and C++ code from MATLAB code

MATLAB Coder generates C and C++ code from MATLAB code for a variety of hardware platforms, from desktop systems to embedded hardware. 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. The generated code is readable and portable. You can combine it with key parts of your existing C and C++ code and libraries. You can also package the generated code as a MEX-function for use in MATLAB.

When used with Embedded Coder®, MATLAB Coder provides code customizations, target-specific optimizations, code traceability, and software-in-the-loop (SIL) and processor-in-the-loop (PIL) verification.

To deploy MATLAB programs as standalone applications, use MATLAB Compiler™. To generate software components for integration with other programming languages, use MATLAB Compiler SDK™.

Product Overview

In this section...

"When to Use MATLAB Coder" on page 1-3
"Code Generation for Embedded Software Applications" on page 1-3
"Code Generation for Fixed-Point Algorithms" on page 1-3

When to Use MATLAB Coder

Use MATLAB Coder to:

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

Code Generation for Embedded Software Applications

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

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

Code Generation for Fixed-Point Algorithms

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

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

Design Considerations for C/C++ Code Generation

- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Which Code Generation Feature to Use” on page 2-3
- “Prerequisites for C/C++ Code Generation from MATLAB” on page 2-4
- “MATLAB Code Design Considerations for Code Generation” on page 2-5
- “Differences Between Generated Code and MATLAB Code” on page 2-6
- “Potential Differences Reporting” on page 2-18
- “Potential Differences Messages” on page 2-20
- “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-24

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
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 “Accelerate MATLAB Algorithm by Generating MEX Function”.
Produce readable, efficient, and compact code from MATLAB algorithms for deployment to desktop and embedded systems.	MATLAB Coder app	MATLAB Coder	Try this in “Generate C Code by Using the MATLAB Coder App”.
	`codegen` function	MATLAB Coder	Try this in “Generate C Code at the Command Line”.
Generate MEX functions to accelerate MATLAB algorithms	MATLAB Coder app	MATLAB Coder	See “Accelerate MATLAB Algorithms” on page 32-6.
	`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 “Call C/C++ Code from MATLAB Code” on page 33-2.
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 32-12

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:

- “Functions that have Multiple Possible Outputs” on page 2-6
- “Writing to ans Variable” on page 2-7
- “Logical Short-Circuiting” on page 2-7
- “Loop Index Overflow” on page 2-8
- “Indexing for Loops by Using Single Precision Operands” on page 2-9
- “Index of an Unentered for Loop” on page 2-10
- “Character Size” on page 2-10
- “Order of Evaluation in Expressions” on page 2-10
- “Name Resolution While Constructing Function Handles” on page 2-11
- “Termination Behavior” on page 2-13
- “Size of Variable-Size N-D Arrays” on page 2-13
- “Size of Empty Arrays” on page 2-13
- “Size of Empty Array That Results from Deleting Elements of an Array” on page 2-13
- “Binary Element-Wise Operations with Single and Double Operands” on page 2-14
- “Floating-Point Numerical Results” on page 2-15
- “NaN and Infinity” on page 2-15
- “Negative Zero” on page 2-15
- “Code Generation Target” on page 2-16
- “MATLAB Class Property Initialization” on page 2-16
- “MATLAB Classes in Nested Property Assignments That Have Set Methods” on page 2-16
- “MATLAB Handle Class Destructors” on page 2-16
- “Variable-Size Data” on page 2-17
- “Complex Numbers” on page 2-17
- “Converting Strings with Consecutive Unary Operators to double” on page 2-17

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 on page 2-18.

Functions that have Multiple Possible Outputs

Certain mathematical operations, such as singular value decomposition and eigenvalue decomposition of a matrix, can have multiple answers. Two different algorithms implementing such an operation can return different outputs for identical input values. Two different implementations of the same algorithm can also exhibit the same behavior.

For such mathematical operations, the corresponding functions in the generated code and MATLAB might return different outputs for identical input values. To see if a function has this behavior, in the corresponding function reference page, see the **C/C++ Code Generation** section under **Extended Capabilities**. Examples of such functions include `svd` and `eig`.

Writing to ans Variable

When you run MATLAB code that returns an output without specifying an output argument, MATLAB implicitly writes the output to the `ans` variable. If the variable `ans` already exists in the workspace, MATLAB updates its value to the output returned.

The code generated from such MATLAB code does not implicitly write the output to an `ans` variable.

For example, define the MATLAB function `foo` that explicitly creates an `ans` variable in the first line. The function then implicitly updates the value of `ans` when the second line executes.

```
function foo %#codegen
ans = 1;
2;
disp(ans);
end
```

Run `foo` at the command line. The final value of `ans`, which is `2`, is displayed at the command line.

```
foo
```

```
2
```

Generate a MEX function from `foo`.

```
codegen foo
```

Run the generated MEX function `foo_mex`. This function explicitly creates the `ans` variable and assigns the value `1` to it. But `foo_mex` does not implicitly update the value of `ans` to `2`.

```
foo_mex
```

```
1
```

Logical Short-Circuiting

Suppose that your MATLAB code has the logical operators `&` and `|` placed inside square brackets (`[` and `]`). For such code patterns, the generated code does not employ short-circuiting behavior for these logical operators, but MATLAB execution might employ short-circuiting behavior. See “Logical Short-Circuiting”.

For example, define the MATLAB function `foo` that uses the `&` operator inside square brackets in the conditional expression of an `if...end` block.

```
function foo
if [returnsFalse() & hasSideEffects()]
end
end

function out = returnsFalse
out = false;
```

```
end

function out = hasSideEffects
out = true;
disp('This is my string');
end
```

The first argument of the & operator is always `false` and determines the value of the conditional expression. So, in MATLAB execution, short-circuiting is employed and the second argument is not evaluated. So, `foo` does not call the `hasSideEffects` function during execution and does not display anything at the command line.

Generate a MEX function for `foo`. Call the generated MEX function `foo_mex`.

```
foo_mex
```

```
This is my string
```

In the generated code, short-circuiting is not employed. So, the `hasSideEffects` function is called and the string is displayed at the command line.

Loop Index Overflow

Suppose that a `for`-loop end value is equal to or close to the maximum or minimum value for the loop index data type. In the generated code, the last increment or decrement of the loop index might cause the index variable to overflow. The index overflow might result in an infinite loop.

When memory integrity checks are enabled, if the code generator detects that the loop index might overflow, it reports an error. The software error checking is conservative. It might incorrectly report a loop index overflow. By default, memory-integrity checks are enabled for MEX code and disabled for standalone C/C++ code. See “Why Test MEX Functions in MATLAB?” on page 26-2 and “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20.

To avoid a loop index overflow, use the workarounds in this table.

Loop Conditions Causing the Potential Overflow	Workaround

• The loop index increments by 1.
• The end value equals the maximum value of the integer type.
	If the loop does not have to cover 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`

<table>
<tr><th>Loop Conditions Causing the Potential Overflow</th><th>Workaround</th></tr>
<tr><td>The loop index decrements by 1.
The end value equals the minimum value of the integer type.</td><td>If the loop does not have to cover 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:
<pre>N=intmin('int32')
for k=N+10:-1:N</pre>with:
<pre>for k=10:-1:1</pre></td></tr>
<tr><td>The loop index 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.</td><td>If the loop must cover the full range of the integer type, cast the type of the loop start, step, and end values to a bigger integer or to double. For example, rewrite:
<pre>M= intmin('int16');
N= intmax('int16');
for k=M:N
    % Loop body
end</pre>as:
<pre>M= intmin('int16');
N= intmax('int16');
for k=int32(M):int32(N)
    % Loop body
end</pre></td></tr>
<tr><td>The loop index increments or decrements by a value not equal to 1.
On the last loop iteration, the loop index is not equal to the end value.</td><td>Rewrite the loop so that the loop index in the last loop iteration is equal to the end value.</td></tr>
</table>

Indexing for Loops by Using Single Precision Operands

Suppose in your MATLABcode, you are indexing a `for` loop that has a colon operator, where at least one of the colon operands is a single type operand and the number of iterations is greater than `flintmax('single') = 16777216`. When all these conditions are true, code generation might generate run-time or compile-time errors because the generated code calculates different values for the loop index variable than the values that MATLAB calculates.

For example, consider this MATLAB code:

```
function j = singlePIndex
n = flintmax('single') + 2;
j = single(0);
for i = single(1):single(n)
    j = i;
end
end
```

This code snippet executes in MATLAB, but it causes a compile-time or run-time error because the value of the loop index variable, `i`, is calculated differently in the generated code. The code generator displays a compile-time or run-time error and stops code generation or execution to prevent this discrepancy.

To avoid this discrepancy, replace the single type operands with double type or integer type operands.

For more information on run-time errors, see "Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors" on page 28-20.

Index of an Unentered for Loop

In your MATLAB code and generated code, after a `for` loop execution is complete, the value of the index variable is equal to its value during the final iteration of the `for` loop.

In MATLAB, if the loop does not execute, the value of the index variable is stored as [ ] (empty matrix). In generated code, if the loop does not execute, the value of the index variable is different than the MATLAB index variable.

- If you provide the `for` loop start and end variables at run time, the value of the index variable is equal to the start of the range. For example, consider this MATLAB code:

  ```
  function out = indexTest(a,b)
  for i = a:b
  end
  out = i;
  end
  ```

  Suppose that `a` and `b` are passed as `1` and `-1`. The `for` loop does not execute. In MATLAB, `out` is assigned [ ]. In the generated code, `out` is assigned the value of `a`, which is `1`.
- If you provide the `for` loop start and end values before compile time, the value of the index variable is equal to `0`. Consider this MATLAB code:

  ```
  function out = indexTest
  for i = 1:-1
  end
  out = i;
  end
  ```

  Suppose that you call this function. In MATLAB, `out` is assigned [ ]. In the generated code, `out` is assigned the value `0`.

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 "Encoding of Characters in Code Generation" on page 5-7.

Order of Evaluation in Expressions

Generated code does not enforce the order of evaluation in expressions. For most expressions, the order of evaluation is not significant. For expressions that have side effects, the generated code might produce the side effects in a 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;
  ```

Name Resolution While Constructing Function Handles

MATLAB and code generation follow different precedence rules for resolving names that follow the symbol `@`. These rules do not apply to anonymous functions. The precedence rules are summarized in this table.

Expression	Precedence Order in MATLAB	Precedence Order in Code Generation
An expression that does not contain periods, for example `@x`	Nested function, local function, private function, path function	Local variable, nested function, local function, private function, path function

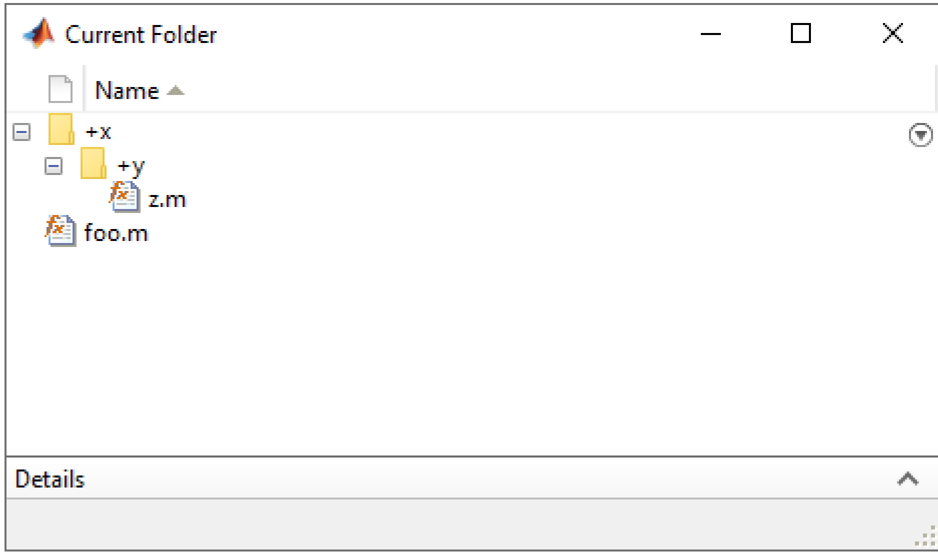
Expression	Precedence Order in MATLAB	Precedence Order in Code Generation
An expression that contains exactly one period, for example `@x.y`	Local variable, path function	Local variable, path function (Same as MATLAB)
An expression that contains more than one period, for example `@x.y.z`	Path function	Local variable, path function

If `x` is a local variable that is itself a function handle, generated code and MATLAB interpret the expression `@x` differently:

- MATLAB produces an error.
- Generated code interprets `@x` as the function handle of `x` itself.

Here is an example that shows this difference in behavior for an expression that contains two periods.

Suppose that your current working folder contains a package `x`, which contains another package `y`, which contains the function `z`. The current working folder also contains the entry-point function `foo` for which you want to generate code.



This is the definition for the file `foo`:

```
function out = foo
    x.y.z = @()'x.y.z is an anonymous function';
    out = g(x);
end

function out = g(x)
    f = @x.y.z;
    out = f();
end
```

This is the definition for function `z`:

```
function out = z
    out = 'x.y.z is a package function';
end
```

Generate a MEX function for `foo`. Separately call both the generated MEX function `foo_mex` and the MATLAB function `foo`.

```
codegen foo
foo_mex
foo
```

```
ans =

    'x.y.z is an anonymous function'


ans =

    'x.y.z is a package function'
```

The generated code produces the first output. MATLAB produces the second output. Code generation resolves `@x.y.z` to the local variable `x` that is defined in `foo`. MATLAB resolves `@x.y.z` to `z`, which is within the package `x.y`.

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 6-16.

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 6-17.

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 using the colon operator (:).	`coder.varsize('X',[1,4],[0,1]);`
	`X = zeros(1,4);`
	`X(:) = [];`
		0-by-0	1-by-0

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

Binary Element-Wise Operations with Single and Double Operands

If your MATLAB code contains a binary element-wise operation that involves a single type operand and a double type operand, the generated code might not produce the same result as MATLAB.

For such an operation, MATLAB casts both operands to double type and performs the operation with the double types. MATLAB then casts the result to single type and returns it.

The generated code casts the double type operand to single type. It then performs the operation with the two single types and returns the result.

For example, define a MATLAB function `foo` that calls the binary element-wise operation `plus`.

```
function out = foo(a,b)
out = a + b;
end
```

Define a variable `s1` of single type and a variable `v1` of double type. Generate a MEX function for `foo` that accepts a single type input and a double type input.

```
s1 = single(1.4e32);
d1 = -5.305e+32;
codegen foo -args {s1, d1}
```

Call both `foo` and `foo_mex` with inputs `s1` and `d1`. Compare the two results.

```
ml = foo(s1,d1);
mlc = foo_mex(s1,d1);
ml == mlc
```

```
ans =

  logical

   0
```

The output of the comparison is a logical `0`, which indicates that the generated code and MATLAB produces different results for these inputs.

Floating-Point Numerical Results

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

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

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.

The bit pattern for `NaN` can differ between MATLAB code output and generated code output because the C99 standard math library that is used to generate code does not specify a unique bit pattern for `NaN` across all implementations. Avoid comparing bit patterns across different implementations, for example, between MATLAB output and SIL or PIL output.

Negative Zero

In a floating-point type, the value `0` has either a positive sign or a negative sign. Arithmetically, `0` is equal to `-0`, but some operations are sensitive to the sign of a 0 input. Examples include `rdivide`, `atan2`, `atan2d`, and `angle`. Division by `0` produces `Inf`, but division by `-0` produces `-Inf`. Similarly, `atan2d(0,-1)` produces `180`, but `atan2d (-0,-1)` produces `-180`.

If the code generator detects that a floating-point variable takes only integer values of a suitable range, then the code generator can use an integer type for the variable in the generated code. If the code generator uses an integer type for the variable, then the variable stores `-0` as `+0` because an integer type does not store a sign for the value `0`. If the generated code casts the variable back to a

floating-point type, the sign of 0 is positive. Division by 0 produces Inf, not -Inf. Similarly, atan2d(0,-1) produces 180, not -180.

There are other contexts in which the generated code might treat -0 differently than MATLAB. For example, suppose that your MATLAB code computes the minimum of two scalar doubles x and y by using z = min(x,y). The corresponding line in the generated C code might be z = fmin(x,y). The function fmin is defined in the runtime math library of the C compiler. Because the comparison operation 0.0 == -0.0 returns true in C/C++, the compiler's implementation of fmin might return either 0.0 or -0.0 for fmin(0.0,-0.0).

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 Property Initialization

Before code generation, at class loading time, MATLAB computes class default values. The code generator uses the values that MATLAB computes. It does not recompute default values. If the property definition uses a function call to compute the initial value, the code generator does not execute this function. If the function has side effects such as modifying a global variable or a persistent variable, then it is possible that the generated code might produce different results than MATLAB. For more information, see “Defining Class Properties for Code Generation” on page 15-3.

MATLAB Classes in Nested Property Assignments That Have Set Methods

When you assign a value to a handle object property, which is itself a property of another object, and so on, then the generated code can call set methods for handle classes that MATLAB does not call.

For example, suppose that you define a set of variables such that x is a handle object, pa is an object, pb is a handle object, and pc is a property of pb. Then you make a nested property assignment, such as:

```
x.pa.pb.pc = 0;
```

In this case, the generated code calls the set method for the object pb and the set method for x. MATLAB calls only the set method for pb.

MATLAB Handle Class Destructors

The behavior of handle class destructors in the generated code can be different from the behavior in MATLAB in these situations:

- The order of destruction of several independent objects might be different in MATLAB than in the generated code.
- The lifetime of objects in the generated code can be different from their lifetime in MATLAB.
- The generated code does not destroy partially constructed objects. If a handle object is not fully constructed at run time, the generated code produces an error message but does not call the

`delete` method for that object. For a System object™, if there is a run-time error in `setupImpl`, the generated code does not call `releaseImpl` for that object.

MATLAB does call the `delete` method to destroy a partially constructed object.

For more information, see "Code Generation for Handle Class Destructors" on page 15-15.

Variable-Size Data

See "Incompatibilities with MATLAB in Variable-Size Support for Code Generation" on page 6-15.

Complex Numbers

See "Code Generation for Complex Data" on page 5-3.

Converting Strings with Consecutive Unary Operators to double

Converting a string that contains multiple, consecutive unary operators to `double` can produce different results between MATLAB and the generated code. Consider this function:

```
function out = foo(op)
out = double(op + 1);
end
```

For an input value `"--"`, the function converts the string `"--1"` to `double`. In MATLAB, the answer is `NaN`. In the generated code, the answer is `1`.

See Also

More About

- "Potential Differences Reporting" on page 2-18
- "Potential Differences Messages" on page 2-20
- "Why Test MEX Functions in MATLAB?" on page 26-2
- "Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors" on page 28-20

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 on page 2-6. 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 2-20. 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.

See Also

More About

- “Potential Differences Messages” on page 2-20
- “Differences Between Generated Code and MATLAB Code” on page 2-6
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 6-15
- “Why Test MEX Functions in MATLAB?” on page 26-2
- “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20

Potential Differences Messages

When you enable potential differences on page 2-18 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 2-20
- "mtimes No Dynamic Scalar Expansion" on page 2-20
- "Matrix-Matrix Indexing" on page 2-21
- "Vector-Vector Indexing" on page 2-21
- "Loop Index Overflow" on page 2-22

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)`.

Loop Index Overflow

```
The generated code assumes the loop index does not overflow on
the last iteration of the loop. If the loop index overflows,
an infinite loop can occur.
```

Suppose that a `for`-loop end value is equal to or close to the maximum or minimum value for the loop index data type. In the generated code, the last increment or decrement of the loop index might cause the index variable to overflow. The index overflow might result in an infinite loop.

When memory integrity checks are enabled, if the code generator detects that the loop index might overflow, it reports an error. The software error checking is conservative. It might incorrectly report a loop index overflow. By default, memory-integrity checks are enabled for MEX code and disabled for standalone C/C++ code. See “Why Test MEX Functions in MATLAB?” on page 26-2 and “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20.

To avoid a loop index overflow, use the workarounds in this table.

Loop Conditions Causing the Potential Overflow	Workaround

• The loop index increments by 1.
• The end value equals the maximum value of the integer type.
	If the loop does not have to cover 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 index decrements by 1.
• The end value equals the minimum value of the integer type.
	If the loop does not have to cover 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`

Loop Conditions Causing the Potential Overflow	Workaround

• The loop index 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.
	If the loop must cover the full range of the integer type, cast the type of the loop 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`
	as:
	`M= intmin('int16');`
	`N= intmax('int16');`
	`for k=int32(M):int32(N)`
	`    % Loop body`
	`end`

• The loop index increments or decrements by a value not equal to 1. • On the last loop iteration, the loop index is not equal to the end value.	Rewrite the loop so that the loop index in the last loop iteration is equal to the end value.

See Also

More About

- “Potential Differences Reporting” on page 2-18
- “Differences Between Generated Code and MATLAB Code” on page 2-6
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 6-15
- “Why Test MEX Functions in MATLAB?” on page 26-2
- “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20

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

MATLAB Features That Code Generation Supports

Code generation from MATLAB code supports many major language features including:

- n-dimensional arrays (see “Array Size Restrictions for Code Generation” on page 5-8)
- matrix operations, including deletion of rows and columns
- variable-size 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 6-19)
- complex numbers (see “Code Generation for Complex Data” on page 5-3)
- numeric classes (see “Supported Variable Types” on page 4-11)
- double-precision, single-precision, and integer math
- enumerations (see “Code Generation for Enumerations” on page 14-2)
- 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 24-26)
- structures (see “Structure Definition for Code Generation” on page 7-2)
- cell arrays (see “Cell Arrays”)
- tables (see “Code Generation for Tables” on page 12-2)
- timetables (see “Code Generation for Timetables” on page 13-2)
- characters (see “Encoding of Characters in Code Generation” on page 5-7)
- string scalars (see “Code Generation for Strings” on page 5-11)
- `categorical` arrays (see “Code Generation for Categorical Arrays” on page 8-2)
- `datetime` arrays (see “Code Generation for Datetime Arrays” on page 10-2)
- `duration` arrays (see “Code Generation for Duration Arrays” on page 11-2)
- sparse matrices (see “Code Generation for Sparse Matrices” on page 5-14)
- function handles (see “Function Handle Limitations for Code Generation” on page 17-2)
- anonymous functions (see “Code Generation for Anonymous Functions” on page 19-6)
- recursive functions (see “Code Generation for Recursive Functions” on page 20-14)
- nested functions (see “Code Generation for Nested Functions” on page 19-7)
- variable length input and output argument lists (see “Code Generation for Variable Length Argument Lists” on page 19-2)
- subset of MATLAB toolbox functions (see “Functions and Objects Supported for C/C++ Code Generation” on page 3-2)

- subset of functions and System objects in several toolboxes (see “Functions and Objects Supported for C/C++ Code Generation” on page 3-2)
- MATLAB classes (see “MATLAB Classes Definition for Code Generation” on page 15-2)
- function calls (see “Resolution of Function Calls for Code Generation” on page 20-2)

MATLAB Language Features That Code Generation Does Not Support

Code generation from MATLAB does not support the following frequently used MATLAB features (this list is not exhaustive):

- scripts
- 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”. For code generation, to achieve implicit expansion, use `bsxfun`.
- GPU arrays

  MATLAB Coder does not support GPU arrays. However, if you have GPU Coder™, you can generate CUDA® MEX code that takes GPU array inputs.
- `calendarDuration` arrays
- Java
- Map containers
- time series objects
- tall arrays
- `try`/`catch` statements
- `import` statements
- Function argument validation

Functions, Classes, and System Objects Supported for Code Generation

Functions and Objects Supported for C/C++ Code Generation

You can generate efficient C/C++ code for a subset of MATLAB built-in functions and toolbox functions and System objects that you call from MATLAB code.

These functions and System objects are listed in the following tables. In these tables, a ⚠ icon before the name of a function or a System object indicates that there are specific usage notes and limitations related to C/C++ code generation for that function or System object. To view these usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom and expand the **C/C++ Code Generation** section.

- Functions and Objects Supported for C/C++ Code Generation (Category List)
- Functions and Objects Supported for C/C++ Code Generation (Alphabetical List)

See Also

More About

- “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-24

Defining MATLAB Variables for C/C++ Code Generation

- “Variables Definition for Code Generation” on page 4-2
- “Best Practices for Defining Variables for C/C++ Code Generation” on page 4-3
- “Eliminate Redundant Copies of Variables in Generated Code” on page 4-6
- “Reassignment of Variable Properties” on page 4-8
- “Reuse the Same Variable with Different Properties” on page 4-9
- “Supported Variable Types” on page 4-11
- “Edit and Represent Coder Type Objects and Properties” on page 4-12

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" on page 4-3
"Use Caution When Reassigning Variables" on page 4-5
"Use Type Cast Operators in Variable Definitions" on page 4-5
"Define Matrices Before Assigning Indexed Variables" on page 4-5

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.

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.

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 4-6.

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

Example 4.1. 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
...
```

Example 4.2. 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 4-8.

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 6-19.

See Also

`coder.nullcopy` | `persistent`

More About

- “Eliminate Redundant Copies of Variables in Generated Code” on page 4-6
- “Structure Definition for Code Generation” on page 7-2
- “Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation” on page 6-19
- “Avoid Data Copies of Function Inputs in Generated Code” on page 34-6

Eliminate Redundant Copies of Variables in Generated Code

In this section...

"When Redundant Copies Occur" on page 4-6

"How to Eliminate Redundant Copies by Defining Uninitialized Variables" on page 4-6

"Defining Uninitialized Variables" on page 4-6

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 4-6.

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 = withoutNullcopy %#codegen

N = 5;
X = zeros(1,N);
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    elseif mod(i,2) == 1
        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 = withNullcopy %#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
```

See Also

`coder.nullcopy`

More About

- "Avoid Data Copies of Function Inputs in Generated Code" on page 34-6

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 4-9.

Reuse the Same Variable with Different Properties

In this section...

"When You Can Reuse the Same Variable with Different Properties" on page 4-9
"When You Cannot Reuse Variables" on page 4-9
"Limitations of Variable Reuse" on page 4-10

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.

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)));
```

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
```

Example 4.3. 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 Generate a MEX function for `example1` and produce a code generation report.

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

3 Open the code generation report.

On the **Variables** tab, you see two uniquely named local variables `t>1` and `t>2`.

SUMMARY | ALL MESSAGES (0) | BUILD LOGS | COD

Name	Type	Size	Class
y	Output	1 × 1	double
u	Input	5 × 5	double
t > 1	Local	1 × 1	double
t > 2	Local	:25 × 1	double

4 In the list of variables, click `t>1`. The report highlights the instances of the variable `t` that are inside of the `if` statement. These instances of `t` are scalar double.

5 Click `t>2`. The code generation report highlights the instances of `t` that are outside of the `if` statement. These instances of `t` are variable-size column vectors with an upper bound of `25`.

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.

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	Fixed-point data types

Edit and Represent Coder Type Objects and Properties

Passing an object to `coder.typeof` or passing a class name as a string scalar to `coder.newtype` creates an object that represents the type of object for code generation.

The coder type object displays a succinct description of the object properties while excluding internal state values. Nonconstant properties display their type and size, while constant properties display only their values.

To create a coder type object, pass a compatible object to `coder.typeof`. For example:

```
t = categorical({'r','g','b'});
tType = coder.typeof(t)
```

The representation of variable `t` is stored in coder type object `tType`.

```
tType = 

   matlab.coder.type.CategoricalType
     1x3 categorical
    Categories : 3x1 homogeneous cell
       Ordinal : 1x1 logical
     Protected : 1x1 logical
```

Object Properties

You can edit the properties of coder type objects. You can assign scalar values to the object properties. Values are implicitly converted to the corresponding coder type values when they are assigned to coder type object properties. The code generator implicitly converts constants assigned to coder type object properties to `coder.Constant` values. You can resize objects themselves

Resize Objects by Using coder.resize

You can resize most objects by using `coder.resize`. You can resize objects, its properties and create arrays within the properties.

For example, for a `timetable` coder object, you can resize the object:

```
t = timetable((1:5)',(11:15)','SampleRate',1);
tType = coder.typeof(t);
tType = coder.resize(tType, [10 2],[1 0])
```

This code resizes the `timetable` to a `:10x2` object.

```
tType = 

   matlab.coder.type.RegularTimetableType
     :10x2 timetable
                    Data : 1x2 homogeneous cell
             Description : 1x0 char
                UserData : 0x0 double
          DimensionNames : {'Time'}    {'Variables'}
           VariableNames : {'Var1'}    {'Var2'}
    VariableDescriptions : 1x2 homogeneous cell
           VariableUnits : 1x2 homogeneous cell
      VariableContinuity : 1x2 matlab.internal.coder.tabular.Continuity
```

```
   StartTime : 1x1 matlab.coder.type.DurationType
  SampleRate : 1x1 double
    TimeStep : 1x1 matlab.coder.type.DurationType
```

The constant properties of `tType` display their values. The nonconstant properties display only their type and size.

Note Not all types representing MATLAB classes are compatible with `coder.resize`.

Resize Objects by Editing Object Properties

You can resize the objects by editing the properties themselves. For a `duration` coder type object `x`, edit the `Size` property to change the size as needed.

```
x = coder.typeof(duration((1:3),0,0));
x.Size = [10 10]
```

This code changes the size of the coder type object.

```
x =

   matlab.coder.type.DurationType
     10x10 duration
    Format : 1x8 char
```

You can also make the coder type object variable-size by setting the `VarDims` flag:

```
x.VarDims(2) = true
```

The second dimension of the coder type object is upper-bounded at `10`.

```
x =

   matlab.coder.type.DurationType
     10x:10 duration
    Format : 1x8 char
```

Legacy Representation of Coder Type Objects

In R2021a, calling `coder.typeof` no longer returns a `coder.ClassType` object. If your workflow requires the legacy representation of coder type objects, use the `getCoderType` function on the variable that has the new representation of your class or object. For example, to get the legacy representation of a `datetime` variable, use the variable that has the new representation `tt` to call the `getCoderType` function:

```
t = datetime;
tt = coder.typeof(t);
ttLegacy = tt.getCoderType()
```

In the Coder Type Editor, the code generator includes the function `getCoderType` for coder type objects. Use this function to return the legacy representation of coder types. See, “Create and Edit Input Types by Using the Coder Type Editor” on page 27-69

Certain MATLAB data types provide customized type representations for MATLAB code generation. In other cases, the type is represented using `coder.ClassType`.

See Also

"Code Generation for Variable-Size Arrays" on page 6-2 | `coder.newtype` | `coder.resize` | `coder.typeof`

Defining Data for Code Generation

- “Data Definition for Code Generation” on page 5-2
- “Code Generation for Complex Data” on page 5-3
- “Encoding of Characters in Code Generation” on page 5-7
- “Array Size Restrictions for Code Generation” on page 5-8
- “Code Generation for Constants in Structures and Arrays” on page 5-9
- “Code Generation for Strings” on page 5-11
- “Define String Scalar Inputs” on page 5-12
- “Code Generation for Sparse Matrices” on page 5-14
- “Specify Array Layout in Functions and Classes” on page 5-17
- “Code Design for Row-Major Array Layout” on page 5-21

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 Is Different	More Information
Arrays	Maximum number of elements is restricted	“Array Size Restrictions for Code Generation” on page 5-8
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 5-3
Characters	Restricted to 8 bits of precision	“Encoding of Characters in Code Generation” on page 5-7
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 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	“Function Handles”

Code Generation for Complex Data

In this section...

"Restrictions When Defining Complex Variables" on page 5-3
"Code Generation for Complex Data with Zero-Valued Imaginary Parts" on page 5-3
"Results of Expressions That Have Complex Operands" on page 5-5
"Results of Complex Multiplication with Nonfinite Values" on page 5-6

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 5-3.
- 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 5-5.

- **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
  ```

Name	Type	Size	Class
y	Output	1 × 1	complex double

  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 5-5.
- 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.
```

Results of Complex Multiplication with Nonfinite Values

When an operand of a complex multiplication contains a nonfinite value, the generated code might produce a different result than the result that MATLAB produces. The difference is due to the way that code generation defines complex multiplication. For code generation:

- Multiplication of a complex value by a complex value $(a + b\mathrm{i})(c + d\mathrm{i})$ is defined as $(ac - bd) + (ad + bc)\mathrm{i}$. The complete calculation is performed, even when a real or an imaginary part is zero.
- Multiplication of a real value by a complex value $c(a + b\mathrm{i})$ is defined as $ca + cb\mathrm{i}$.

Encoding of Characters in Code Generation

MATLAB represents characters in 16-bit Unicode. The code generator represents characters in an 8-bit codeset that the locale setting determines. Differences in character encoding between MATLAB and code generation have these consequences:

- Code generation of characters with numeric values greater than 255 produces an error.
- For some characters in the range 128–255, it might not be possible to represent the character in the codeset of the locale setting or to convert the character to an equivalent 16-bit Unicode character. Passing characters in this range between MATLAB and generated code can result in errors or different answers.
- For code generation, some toolbox functions accept only 7-bit ASCII characters.
- Casting a character that is not in the 7-bit ASCII codeset to a numeric type, such as double, can produce a different result in the generated code than in MATLAB. As a best practice, for code generation, avoid performing arithmetic with characters.

See Also

More About

- “Locale Setting Concepts for Internationalization”
- “Differences Between Generated Code and MATLAB Code” on page 2-6

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, at run time, if the number of elements exceeds the maximum and run-time error checks are enabled, the generated code reports an error. By default, run-time error checks are enabled for MEX code and disabled for standalone C/C++ code.

See Also

`coder.HardwareImplementation`

More About

- "Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors" on page 28-20
- "Control Run-Time Checks" on page 32-12
- "Potential Differences Reporting" on page 2-18

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.

Code Generation for Strings

Code generation supports 1-by-1 MATLAB string arrays. Code generation does not support string arrays that have more than one element.

A 1-by-1 string array, called a string scalar, contains one piece of text, represented as a 1-by-n character vector. An example of a string scalar is `"Hello, world"`. For more information about strings, see “Text in String and Character Arrays”.

Limitations

For string scalars, code generation does not support:

- Global variables
- Indexing with curly braces `{}`
- Missing values
- Defining input types programmatically (by using preconditioning with `assert` statements)
- Their use with `coder.varsize`

For code generation, limitations that apply to classes apply to strings. See “MATLAB Classes Definition for Code Generation” on page 15-2.

Differences Between Generated Code and MATLAB Code

- Converting a string that contains multiple unary operators to `double` can produce different results between MATLAB and the generated code. Consider this function:

  ```
  function out = foo(op)
  out = double(op + 1);
  end
  ```

  For an input value `"--"`, the function converts the string `"--1"` to `double`. In MATLAB, the answer is `NaN`. In the generated code, the answer is `1`.
- Double conversion for a string with misplaced commas (commas that are not used as thousands separators) can produce different results from MATLAB.

See Also

More About

- “Define String Scalar Inputs” on page 5-12

Define String Scalar Inputs

You can define string scalar inputs at the command line or in the MATLAB Coder app. Programmatic specification of string scalar input types by using preconditioning (`assert` statements) is not supported.

Define String Scalar Types at the Command Line

To define string scalar inputs at the command line, use one of these procedures:

- "Provide an Example String Scalar Input" on page 5-12
- "Provide a String Scalar Type" on page 5-12
- "Provide a Constant String Scalar Input" on page 5-12
- "Provide a Variable-Size String Scalar Input" on page 5-12

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example String Scalar Input

To provide an example string scalar to `codegen`, use the `-args` option:

```
codegen myFunction -args {"Hello, world"}
```

Provide a String Scalar Type

To provide a type for a string scalar to `codegen`:

1. Define a string scalar. For example:

   ```
   s = "mystring";
   ```
2. Create a type from `s`.

   ```
   t = coder.typeof(s);
   ```
3. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Provide a Constant String Scalar Input

To specify that a string scalar input is constant, use `coder.Constant` with the `-args` option:

```
codegen myFunction -args {coder.Constant("Hello, world")}
```

Provide a Variable-Size String Scalar Input

To specify that a string scalar input has a variable-size:

1. Define a string scalar. For example:

   ```
   s = "mystring";
   ```
2. Create a type from `s`.

   ```
   t = coder.typeof(s);
   ```

3. Assign the `Value` property of the type to a type for a variable-size character vector that has the upper bound that you want. For example, specify that type `t` is variable-size with an upper bound of 10.

   ```
   t.Properties.Value = coder.typeof('a',[1 10], [0 1]);
   ```

   To specify that `t` is variable-size with no upper bound:

   ```
   t.Properties.Value = coder.typeof('a',[1 inf]);
   ```

4. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Define String Scalar Inputs in the MATLAB Coder App

To define string scalar inputs in the app, use one of these procedures:

- "Automatically Define Input Types by Using the App" on page 24-4
- "Define Input Parameter by Example by Using the App" on page 24-6
- "Define or Edit Input Parameter Type by Using the App" on page 24-14

See Also

`coder.Constant` | `coder.getArgTypes` | `coder.typeof`

More About

- "Code Generation for Strings" on page 5-11
- "Specify Properties of Entry-Point Function Inputs" on page 27-43

Code Generation for Sparse Matrices

Sparse matrices provide efficient storage in memory for arrays with many zero elements. Sparse matrices can provide improved performance and reduced memory usage for generated code. Computation time on sparse matrices scales only with the number of operations on nonzero elements.

Functions for creating and manipulating sparse matrices are listed in “Sparse Matrices”. To check if a function is supported for code generation, see the function reference page. Code generation does not support sparse matrix inputs for all functions.

Sparse Data Types in Generated Code

If your target language is C, the code generator creates a type definition for sparse matrices called `sparse`. This definition stores the arrays of row indices, column indices, and corresponding element values for the sparse matrix. The `sparse` type definition is generated in the file `myFunction_types.h`, where `myFunction` refers to the name of your top-level function.

If your target language is C++, the code generator creates a class `sparse` in the file `sparse.h`.

The number of nonzero elements in a sparse matrix can change during computation. For this reason, sparse matrices in the generated code use variable-size arrays and dynamic memory allocation. If your target language is C, the generated code implements dynamically allocated variables by using the `emxArray` type. If your target language is C++, the generated code implements dynamically allocated variables by using the `coder::array` class template.

For example, consider the function `myDiag`:

```
function out = myDiag(n,k)
% create diagonal sparse matrix
%#codegen
A = speye(n);
out = A.*k;
end
```

Generate code for the function by using the `codegen` command:

```
codegen -config:lib myDiag -args {3, 5} -launchreport
```

The sparse type can be found in the file `myDiag_types.h`.

Input Definition

Suppose that you have a function `foo` that accepts a sparse matrix as an input. This function multiplies the sparse matrix by an identity matrix and outputs the product:

```
function C = foo(ASparseInput)
%#codegen
B = speye(size(ASparseInput'));
C = ASparseInput*B;
```

Suppose that you want to generate standalone `lib`, `dll`, or `exe` code to use outside of the MATLAB environment. To generate `lib` code, enter:

```
codegen -config:lib foo -args {sparse(5,5)} -launchreport
```

You can simplify your standalone code by constructing the sparse matrix inside your entry-point function rather than passing a sparse matrix as an input. When you follow this guideline, construction of the sparse matrix can be deferred to the code generator. Other code that uses your generated code can pass input types such as arrays rather than specialized sparse types.

For example, instead of generating code directly from `foo`, create a new entry-point function `fooMain` to generate code from. Replace the sparse input with the triplet form of the sparse data.

```
function [ii,jj,out] = fooMain(i,j,v,m,n)
%#codegen
S = sparse(i,j,v,m,n);
[ii,jj,out] = find(foo(S));
```

Suppose that you want to generate code for a 5-by-5 sparse matrix `S` with a variable-size number of nonzero elements. To generate code, enter:

```
S = sparse(5,5);
[m,n] = size(S);
[i,j,v] = find(S);
i = coder.typeof(i,[inf 1]);
codegen -config:lib fooMain -args {i,i,i,m,n} -launchreport
```

You can specify the input for `fooMain` with integer and variable-size array types. If you generate code directly from `foo`, you must construct the input as a `sparse` type.

If you do choose to pass a sparse matrix as an entry-point function input, you can use `coder.typeof` to initialize the input. For example, for the function `foo`, you can enter:

```
t = coder.typeof(sparse(5,5));
codegen -config:lib foo -args {t} -launchreport
```

For sparse matrices, the code generator does not track upper bounds for variable-size dimensions. All variable-size dimensions are treated as unbounded.

If you generate a MEX function for `foo`, the input and output data must be converted to `sparse` type. This conversion can slow performance for repeated MEX function calls or large inputs and outputs.

You cannot define sparse input types programmatically by using `assert` statements.

Code Generation Guidelines

Initialize matrices by using sparse constructors to maximize your code efficiency. For example, to construct a 3-by-3 identity matrix, use `speye(3,3)` rather than `sparse(eye(3,3))`.

Indexed assignment into sparse matrices incurs an overhead compared to indexed assignment into full matrices. For example:

```
S = speye(10);
S(7,7) = 42;
```

As in MATLAB, sparse matrices are stored in compressed sparse column format. When you insert a new nonzero element into a sparse matrix, all subsequent nonzero elements must be shifted downward, column by column. These extra manipulations can slow performance.

Code Generation Limitations

To generate code that uses sparse matrices, dynamic memory allocation must be enabled. To store the changing number of nonzero elements, and their values, sparse matrices use variable-size arrays in the generated code. To change dynamic memory allocation settings, see “Control Memory Allocation for Variable-Size Arrays” on page 6-4. Because sparse matrices use variable-size arrays for dynamic memory allocation, limitations on “Variable-Size Data” also apply to sparse matrices.

You cannot assign sparse data to data that is not sparse. The generated code uses distinct data type representations for sparse and full matrices. To convert to and from sparse data, use the explicit `sparse` and `full` conversion functions.

You cannot define a sparse matrix with competing size specifications. The code generator fixes the size of the sparse matrix when it produces the corresponding data type definition in C/C++. As an example, the function `foo` causes an error in code generation:

```
function y = foo(n)
%#codegen
if n > 0
    y = sparse(3,2);
else
    y = sparse(4,3);
end
```

Logical indexing into sparse matrices is not supported for code generation. For example, this syntax causes an error:

```
S = magic(3);
S(S > 7) = 42;
```

For sparse matrices, you cannot delete array elements by assigning empty arrays:

```
S(:,2) = [];
```

See Also
`codegen` | `coder.typeof` | `full` | `magic` | `sparse` | `speye`

More About
- “Sparse Matrices”
- “Code Generation for Variable-Size Arrays” on page 6-2
- “Use C Arrays in the Generated Function Interfaces” on page 31-3
- “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces” on page 31-15

Specify Array Layout in Functions and Classes

You can specialize individual MATLAB functions for row-major layout or column-major layout by inserting `coder.rowMajor` or `coder.columnMajor` calls into the function body. Using these function specializations, you can combine row-major data and column-major data in your generated code. You can also specialize classes for one specific array layout. Function and class specializations allow you to:

- Incrementally modify your code for row-major layout or column-major layout.
- Define array layout boundaries for applications that require different layouts in different components.
- Structure the inheritance of array layout between many different functions and classes.

For MATLAB Coder entry-point (top-level) functions, all inputs and outputs must use the same array layout. In the generated C/C++ code, the entry-point function interface accepts and returns data with the same array layout as the function array layout specification.

Note By default, code generation uses column-major array layout.

Specify Array Layout in a Function

For an example of a specialized function, consider `addMatrixRM`:

```
function [S] = addMatrixRM(A,B)
%#codegen
S = zeros(size(A));
coder.rowMajor; % specify row-major code
for row = 1:size(A,1)
   for col = 1:size(A,2)
       S(row,col) = A(row,col) + B(row,col);
   end
end
```

For MATLAB Coder, you can generate code for `addMatrixRM` by using the `codegen` command.

```
codegen addMatrixRM -args {ones(20,10),ones(20,10)} -config:lib -launchreport
```

Because of the `coder.rowMajor` call, the code generator produces code that uses data stored in row-major layout.

Other functions called from a row-major function or column-major function inherit the same array layout. If a called function has its own distinct `coder.rowMajor` or `coder.columnMajor` call, the local call takes precedence.

You can mix column-major and row-major functions in the same code. The code generator inserts transpose or conversion operations when passing data between row-major and column-major functions. These conversion operations ensure that array elements are stored as required by functions with different array layout specifications. For example, the inputs to a column-major function, called from a row-major function, are converted to column-major layout before being passed to the column-major function.

Query Array Layout of a Function

To query the array layout of a function at compile time, use `coder.isRowMajor` or `coder.isColumnMajor`. This query can be useful for specializing your generated code when it involves row-major and column-major functions. For example, consider this function:

```
function [S] = addMatrixRouted(A,B)
 if coder.isRowMajor
     %execute this code if row-major
     S = addMatrixRM(A,B);
 elseif coder.isColumnMajor
     %execute this code if column-major
     S = addMatrix_OptimizedForColumnMajor(A,B);
 end
```

This function behaves differently depending on whether it is row-major or column-major. When `addMatrixRouted` is row-major, it calls the `addMatrixRM` function, which has efficient memory access for row-major data. When the function is column-major, it calls a version of the `addMatrixRM` function optimized for column-major data.

For example, consider this function definition. The algorithm iterates through the columns in the outer loop and the rows in the inner loop, in contrast to the `addMatrixRM` function.

```
function [S] = addMatrix_OptimizedForColumnMajor(A,B)
%#codegen
S = zeros(size(A));
for col = 1:size(A,2)
   for row = 1:size(A,1)
       S(row,col) = A(row,col) + B(row,col);
   end
end
```

Code generation for this function yields:

```
...
/* column-major layout */
for (col = 0; col < 10; col++) {
  for (row = 0; row < 20; row++) {
     S[row + 20 * col] = A[row + 20 * col] + B[row + 20 * col];
  }
}
...
```

The generated code has a stride length of only one element. Due to the specializing queries, the generated code for `addMatrixRouted` provides efficient memory access for either choice of array layout.

Specify Array Layout in a Class

You can specify array layout for a class so that object property variables are stored with a specific array layout. To specify the array layout, place a `coder.rowMajor` or `coder.columnMajor` call in the class constructor. If you assign an object with a specified array layout to the property of another object, the array layout of the assigned object takes precedence.

Consider the row-major class `rowMats` as an example. This class contains matrix properties and a method that consists of an element-wise addition algorithm. The algorithm in the method performs

more efficiently for data stored in row-major layout. By specifying `coder.rowMajor` in the class constructor, the generated code uses row-major layout for the property data.

```
classdef rowMats
    properties (Access = public)
        A;
        B;
        C;
    end
    methods
        function obj = rowMats(A,B)
            coder.rowMajor;
            if nargin == 0
                obj.A = 0;
                obj.B = 0;
                obj.C = 0;
            else
                obj.A = A;
                obj.B = B;
                obj.C = zeros(size(A));
            end
        end
        function obj = add(obj)
            for row = 1:size(obj.A,1)
                for col = 1:size(obj.A,2)
                    obj.C(row,col) = obj.A(row,col) + obj.B(row,col);
                end
            end
        end
    end
end
```

Use the class in a simple function `doMath`. The inputs and outputs of the entry-point function must all use the same array layout.

```
function [out] = doMath(in1,in2)
%#codegen
out = zeros(size(in1));
myMats = rowMats(in1,in2);
myMats = myMats.add;
out = myMats.C;
end
```

For MATLAB Coder, you can generate code by entering:

```
A = rand(20,10);
B = rand(20,10);
cfg = coder.config('lib');
codegen -config cfg doMath -args {A,B} -launchreport
```

With default settings, the code generator assumes that the entry-point function inputs and outputs use column-major layout, because you do not specify row-major layout for the function `doMath`. Therefore, before calling the class constructor, the generated code converts `in1` and `in2` to row-major layout. Similarly, it converts the `doMath` function output back to column-major layout.

When designing a class for a specific array layout, consider:

- If you do not specify the array layout in a class constructor, objects inherit their array layout from the function that calls the class constructor, or from code generation configuration settings.
- You cannot specify the array layout in a nonstatic method by using `coder.rowMajor` or `coder.columnMajor`. Methods use the same array layout as the receiving object. Methods do not inherit the array layout of the function that calls them. For static methods, which are used similarly to ordinary functions, you can specify the array layout in the method.
- If you specify the array layout of a superclass, the subclass inherits this array layout specification. You cannot specify conflicting array layouts between superclasses and subclasses.

See Also

`codegen` | `coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

More About

- "Generate Code That Uses Row-Major Array Layout" on page 37-4
- "Code Design for Row-Major Array Layout" on page 5-21
- "Generate Code That Uses N-Dimensional Indexing" on page 27-133

Code Design for Row-Major Array Layout

Outside of code generation, MATLAB uses column-major layout by default. Array layout specifications do not affect self-contained MATLAB code. To test the efficiency of your generated code or your MATLAB Function block, create separate versions with row-major layout and column-major layout. Then, compare their performance.

You can design your MATLAB code to avoid potential inefficiencies related to array layout. Inefficiencies can be caused by:

- Conversions between row-major layout and column-major layout.
- One-dimensional or linear indexing of row-major data.
- Reshaping or rearrangement of row-major data.

Array layout conversions are necessary when you mix row-major and column-major specifications in the same code or model, or when you use linear indexing on data that is stored in row-major. When you simulate a model or generate code for a model that uses column-major, and that contains a MATLAB Function block that uses row-major, then the software converts input data to row-major and output data back to column-major as needed, and vice versa.

Inefficiencies can be caused by functions or algorithms that are less optimized for a given choice of array layout. If a function or algorithm is more efficient for a different layout, you can enforce that layout by embedding it in another function with a `coder.rowMajor` or `coder.columnMajor` call.

Understand Potential Inefficiencies Caused by Array Layout

Consider the code for `myMixedFn2`, which uses `coder.ceval` to pass data with row-major and column-major layout:

```
function [B, C] = myMixedFn2(x,y)
%#codegen
% specify type of return arguments for ceval calls
A = zeros(size(x));
B = zeros(size(x));
C = zeros(size(x));

% include external C functions that use row-major & column-major
coder.cinclude('addMatrixRM.h');
coder.updateBuildInfo('addSourceFiles', 'addMatrixRM.c');
coder.cinclude('addMatrixCM.h');
coder.updateBuildInfo('addSourceFiles', 'addMatrixCM.c');

% call C function that uses column-major order
coder.ceval('-layout:columnMajor','addMatrixCM', ...
    coder.rref(x),coder.rref(y),coder.wref(A));

% compute B
for i = 1:numel(A)
    B(i) = A(i) + 7;
end

% call C function that uses row-major order
coder.ceval('-layout:rowMajor','addMatrixRM', ...
    coder.rref(y),coder.rref(B),coder.wref(C));
end
```

The external files are:

addMatrixRM.h

```
extern void addMatrixRM(const double x[200], const double y[200], double z[200]);
```

addMatrixRM.c

```
#include "addMatrixRM.h"

void addMatrixRM(const double x[200], const double y[200], double z[200])
{
  int row;
  int col;

  /* add two matrices */
  for (row = 0; row < 20; row++) {
    /* row by row */
    for (col = 0; col < 10; col++) {
      /* each element in current row */
      z[col + 10 * row] = x[col + 10 * row] + y[col + 10 * row];
    }
  }
}
```

addMatrixCM.h

```
extern void addMatrixCM(const double x[200], const double y[200], double z[200]);
```

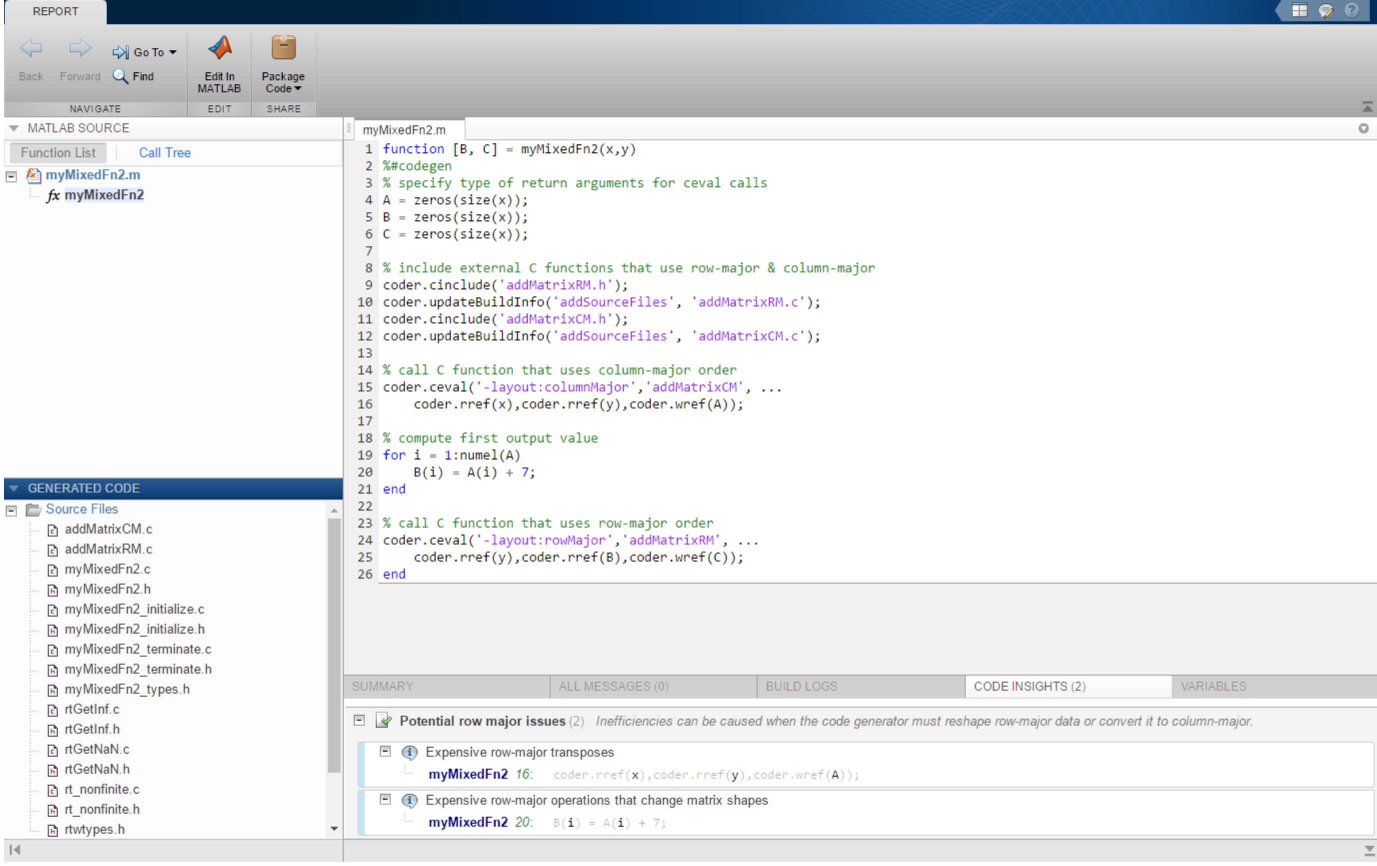
addMatrixCM.c

```
#include "addMatrixCM.h"

void addMatrixCM(const double x[200], const double y[200], double z[200])
{
  int row;
  int col;

  /* add two matrices */
  for (row = 0; row < 20; row++) {
    /* row by row */
    for (col = 0; col < 10; col++) {
      /* each element in current row */
      z[row + 20 * col] = x[row + 20 * col] + y[row + 20 * col];
    }
  }
}
```

Declare the configuration object, `cfg`. Generate code that uses row-major layout by using the `-rowmajor` option.

```
cfg = coder.config('lib');
cfg.HighlightPotentialRowMajorIssues = true;
codegen myMixedFn2 -args {ones(20,10),ones(20,10)} -config cfg -launchreport -rowmajor
```

Highlighted issues are displayed in the code generation report, on the **Code Insights** tab, under the **Potential row major issues** section.



Array layout inefficiencies occur here because:

- The code generator must convert the input variables `x` and `y` to column-major layout before passing them to `addMatrixCM`. Transposes must be inserted into the generated code.
- The code generator must transpose the output variable `A` back into row-major layout, because `myMixedFn2` uses row-major layout.
- The for-loop uses linear indexing, which requires column-major data. The code generator must recalculate the linear indexing because variables `A` and `B` are stored in row-major.

Linear Indexing Uses Column-Major Array Layout

The code generator follows MATLAB column-major semantics for linear indexing. For more information on linear indexing in MATLAB, see "Array Indexing".

To use linear indexing on row-major data, the code generator must first recalculate the data representation in column-major layout. This additional processing can slow performance. To improve code efficiency, avoid using linear indexing on row-major data, or use column-major layout for code that uses linear indexing.

For example, consider the function `sumShiftedProducts`, which accepts a matrix as an input and outputs a scalar value. The function uses linear indexing on the input matrix to sum up the product of each matrix element with an adjacent element. The output value of this operation depends on the order in which the input elements are stored.

```
function mySum = sumShiftedProducts(A)
%#codegen
mySum = 0;
% create linear vector of A elements
B = A(:);
% multiply B by B with elements shifted by one, and take sum
mySum = sum( B.*circshift(B,1) );
end
```

For MATLAB Coder, to generate code that uses row-major layout, enter:

```
codegen -config:mex sumShiftedProducts -args {ones(2,3)} -launchreport -rowmajor
```

For an example input, consider the matrix:

```
D = reshape(1:6,3,2)'
```

which yields:

```
D =
     1     2     3
     4     5     6
```

If you pass this matrix as input to the generated code, the elements of `A` are stored in the order:

```
     1     2     3     4     5     6
```

In contrast, because the vector `B` is obtained by linear indexing, it is stored in the order:

```
     1     4     2     5     3     6
```

The code generator must insert a reshaping operation to rearrange the data from row-major layout for `A` to column-major layout for `B`. This additional operation reduces the efficiency of the function for row-major layout. The inefficiency increases with the size of the array. Because linear indexing always uses column-major layout, the generated code for `sumShiftedProducts` produces the same output result whether generated with row-major layout or column-major layout.

In general, functions that compute indices or subscripts also use linear indexing, and produce results corresponding to data stored in column-major layout. These functions include:

- `ind2sub`
- `sub2ind`
- `colon`

See Also

`coder.ceval` | `coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

More About

- “Generate Code That Uses Row-Major Array Layout” on page 37-4
- “Specify Array Layout in Functions and Classes” on page 5-17
- “Generate Code That Uses N-Dimensional Indexing” on page 27-133
- “Code Generation Reports” on page 28-7

Code Generation for Variable-Size Data

- “Code Generation for Variable-Size Arrays” on page 6-2
- “Control Memory Allocation for Variable-Size Arrays” on page 6-4
- “Specify Upper Bounds for Variable-Size Arrays” on page 6-6
- “Define Variable-Size Data for Code Generation” on page 6-8
- “Diagnose and Fix Variable-Size Data Errors” on page 6-12
- “Incompatibilities with MATLAB in Variable-Size Support for Code Generation” on page 6-15
- “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” on page 6-22

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 6-8.

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 6-4.

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.

Name	Type	Size	Class
y	Output	1 × 1	double
A	Input	1 × :16	char
n	Input	1 × 1	double
X	Local	1 × :?	double

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. Italics indicates that the code generator produced a variable-size array, but the size of the array does not change during execution.

Name	Type	Size	Class
y	Output	1 × 2	double
n	Input	1 × 1	double
Z	Local	*1 × 4*	double

See Also

More About

- “Control Memory Allocation for Variable-Size Arrays” on page 6-4
- “Specify Upper Bounds for Variable-Size Arrays” on page 6-6
- “Define Variable-Size Data for Code Generation” on page 6-8

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, you can:

- Provide upper bounds for variable-size arrays on page 6-4.
- Disable dynamic memory allocation on page 6-4.
- Configure the code generator to use dynamic memory allocation for arrays bigger than a threshold on page 6-4.

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 6-6.
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 6-4.

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`.

See Also

More About

- "Code Generation for Variable-Size Arrays" on page 6-2
- "Configure Build Settings" on page 27-13

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 24-3 and “Make Dimensions Variable-Size When They Meet Size Threshold” on page 24-5.

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 6-8

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 6-8
Assign multiple, constant sizes to the same variable before using (reading) the variable.	"Assign Multiple Sizes to the Same Variable" on page 6-8
Define all instances of a variable to be variable-size.	"Define Variable-Size Data Explicitly by Using coder.varsize" on page 6-9

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 6-10.

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 6-6

Diagnose and Fix Variable-Size Data Errors

In this section...

"Diagnosing and Fixing Size Mismatch Errors" on page 6-12
"Diagnosing and Fixing Errors in Detecting Upper Bounds" on page 6-13

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.

Diagnosing and Fixing Errors in Detecting Upper Bounds

Check your code for these issues:

Using Nonconstant Dimensions in a Matrix Constructor

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

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

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

There are several ways to fix the problem:

- Enable dynamic memory allocation and recompile. During code generation, MATLAB does not check for upper bounds when it uses dynamic memory allocation for variable-size data.

- If you do not want to use dynamic memory allocation, add an `assert` statement before the first use of `u`:

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

Incompatibilities with MATLAB in Variable-Size Support for Code Generation

In this section...

"Incompatibility with MATLAB for Scalar Expansion" on page 6-15
"Incompatibility with MATLAB in Determining Size of Variable-Size N-D Arrays" on page 6-16
"Incompatibility with MATLAB in Determining Size of Empty Arrays" on page 6-17
"Incompatibility with MATLAB in Determining Class of Empty Arrays" on page 6-18
"Incompatibility with MATLAB in Matrix-Matrix Indexing" on page 6-18
"Incompatibility with MATLAB in Vector-Vector Indexing" on page 6-19
"Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation" on page 6-19
"Incompatibility with MATLAB in Concatenating Variable-Size Matrices" on page 6-20
"Differences When Curly-Brace Indexing of Variable-Size Cell Array Inside Concatenation Returns No Elements" on page 6-20

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`.

SUMMARY	ALL MESSAGES (0)	BUILD LOGS	CODE INSIGHTS (1)	VARIABLES

Name	Type	Size	Class
y	Output	3 × 3	double
u	Input	1 × 1	double
z	Local	:3 × :3	double

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)
Subscripted assignment dimension 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`.

This behavior persists while determining the size of empty character vectors which are denoted as `''`. For example, consider the following code:

```
function out = string_size
out = size('');
end
```

Here, the value of `out` might be `1x0` or `0x1` in generated code, but `0x0` in MATLAB.

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 2-13.

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, 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.

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”. 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” on page 6-22
“Toolbox Functions with Restrictions for Variable-Size Data” on page 6-23

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 6-23.

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” on page 3-2.

Function	Restrictions for Variable-Size Data
`all`	• See “Automatic dimension restriction” on page 6-22. • 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 6-22. • An error occurs if you pass the first argument a variable-size matrix that is 0-by-0 at run time.
`cat`	• Dimension argument must be a constant.
`conv`	• See “Variable-length vector restriction” on page 6-22. • 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 6-22.
`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 6-22.
`detrend`	• For first argument for row vectors only, see “Array-to-vector restriction” on page 6-22.
`diag`	• See “Array-to-vector restriction” on page 6-22.
`diff`	• See “Automatic dimension restriction” on page 6-22. • 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 6-22.
`filter`	• For first and second arguments, see “Variable-length vector restriction” on page 6-22. • See “Automatic dimension restriction” on page 6-22.
`hist`	• For second argument, see “Variable-length vector restriction” on page 6-22. • For second input argument, see “Array-to-scalar restriction” on page 6-22.
`histc`	• See “Automatic dimension restriction” on page 6-22.
`ifft`	• See “Automatic dimension restriction” on page 6-22.
`ind2sub`	• First input (the size vector input) must be fixed size.

Function	Restrictions for Variable-Size Data
interp1	• For the xq input, see "Array-to-vector restriction" on page 6-22. • If v becomes a row vector at run time, the array to vector restriction on page 6-22 applies. If v becomes a column vector at run time, this restriction does not apply.
ipermute	• Order input must be fixed size.
issorted	• See "Automatic dimension restriction" on page 6-22.
magic	• Argument must be a constant. • Output can be fixed-size matrices only.
max	• See "Automatic dimension restriction" on page 6-22.
maxk	• See "Automatic dimension restriction" on page 6-22.
mean	• See "Automatic dimension restriction" on page 6-22. • 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 6-22. • 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 6-22.
mink	• See "Automatic dimension restriction" on page 6-22.
mode	• See "Automatic dimension restriction" on page 6-22. • 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.
poly	• See "Variable-length vector restriction" on page 6-22.
polyfit	• For first and second arguments, see "Variable-length vector restriction" on page 6-22.

Function	Restrictions for Variable-Size Data
prod	• See "Automatic dimension restriction" on page 6-22. • 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.
roots	• See "Variable-length vector restriction" on page 6-22.
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.
sort	• See "Automatic dimension restriction" on page 6-22.
std	• See "Automatic dimension restriction" on page 6-22. • 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.

Function	Restrictions for Variable-Size Data
`sum`	• See "Automatic dimension restriction" on page 6-22. • 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 6-22. • 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 6-22 on first argument.
`var`	• See "Automatic dimension restriction" on page 6-22. • An error occurs if you pass a variable-size matrix with 0-by-0 dimensions at run time.
`vecnorm`	• See "Automatic dimension restriction" on page 6-22.

Code Generation for MATLAB Structures

- “Structure Definition for Code Generation” on page 7-2
- “Structure Operations Allowed for Code Generation” on page 7-3
- “Define Scalar Structures for Code Generation” on page 7-4
- “Define Arrays of Structures for Code Generation” on page 7-6
- “Index Substructures and Fields” on page 7-8
- “Assign Values to Structures and Fields” on page 7-10

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:

- Index structure fields using dot notation
- Define primary function inputs as structures
- Pass structures to local functions

Define Scalar Structures for Code Generation

In this section...

"Restrictions When Defining Scalar Structures by Assignment" on page 7-4
"Adding Fields in Consistent Order on Each Control Flow Path" on page 7-4
"Restriction on Adding New Fields After First Use" on page 7-4

Restrictions When Defining Scalar Structures by Assignment

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

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

Adding Fields in Consistent Order on Each Control Flow Path

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

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

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

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

Restriction on Adding New Fields After First Use

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

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

For example, consider this code:

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

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

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

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

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

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

Define Arrays of Structures for Code Generation

In this section...

"Ensuring Consistency of Fields" on page 7-6
"Using repmat to Define an Array of Structures with Consistent Field Properties" on page 7-6
"Defining an Array of Structures by Using struct" on page 7-6
"Defining an Array of Structures Using Concatenation" on page 7-7

Ensuring Consistency of Fields

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

Once you have created the array of structures, you can make the structure fields variable-size by using `coder.varsize`. See "Declare Variable-Size Structure Fields".

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 “Creating, Concatenating, and Expanding Matrices”. For code generation, the structures that you concatenate must have the same size, class, and complexity.

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

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

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

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”).

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 20-11).

Do not assign handle classes or sparse arrays to global structure variables

Global structure variables cannot contain handle objects or sparse arrays.

Code Generation for Categorical Arrays

Code Generation for Categorical Arrays

In this section...
“Define Categorical Arrays for Code Generation” on page 8-2
“Allowed Operations on Categorical Arrays” on page 8-2
“MATLAB Toolbox Functions That Support Categorical Arrays” on page 8-3

Categorical arrays store data with values from a finite set of discrete categories. You can specify an order for the categories, but it is not required. A categorical array provides efficient storage and manipulation of nonnumeric data, while also maintaining meaningful names for the values.

When you use categorical arrays with code generation, adhere to these restrictions:

Define Categorical Arrays for Code Generation

For code generation, use the `categorical` function to create categorical arrays. For example, suppose the input argument to your MATLAB function is a numeric array of arbitrary size whose elements have values of either `1`, `2`, or `3`. You can convert these values to the categories `small`, `medium`, and `large` and turn the input array into a categorical array, as shown in this code.

```
function c = foo(x) %#codegen
    c = categorical(x,1:3,{'small','medium','large'});
end
```

Allowed Operations on Categorical Arrays

For code generation, you are restricted to the operations on categorical arrays listed in this table.

Operation	Example	Notes
assignment operator: `=`	`c = categorical(1:3,1:3,{'small','medium','large'});` `c(1) = 'large';`	Code generation does not support using the assignment operator `=` to: • Delete an element. • Expand the size of a categorical array. • Add a new category, even when the array is not protected.
relational operators: `< > <= >= == ~=`	`c = categorical(1:3,'Ordinal',true);` `tf = c(1) < c(2);`	Code generation supports all relational operators.
cast to numeric type	`c = categorical(1:3);` `double(c(1));`	Code generation supports casting categorical arrays to arrays of double- or single-precision floating-point numbers, or to integers.

Operation	Example	Notes

conversion to text	`c = categorical(1:3,1:3,{'small','medium','large'});`
	`c1 = cellstr(c(1)); % One element`
	`c2 = cellstr(c);    % Entire array`
		Code generation does not support using the `char` or `string` functions to convert categorical values to text.
		
		To convert one or more elements of a categorical array to text, use the `cellstr` function.

indexing operation	`c = categorical(1:3,1:3,{'small','medium','large'});`
	`idx = [1 2];`
	`c(idx);`
	`idx = logical([1 1 0]);`
	`c(idx);`
		Code generation supports indexing by position, linear indexing, and logical indexing.

concatenation	`c1 = categorical(1:3,1:3,{'small','medium','large'});`
	`c2 = categorical(4:6,[2 1 4],{'medium','small','extra-large'});`
	`c = [c1 c2];`
		Code generation supports concatenation of categorical arrays along any dimension.

MATLAB Toolbox Functions That Support Categorical Arrays

For code generation, you can use categorical arrays with these MATLAB toolbox functions:

- `addcats`
- `cat`
- `categorical`
- `categories`
- `cellstr`
- `countcats`
- `ctranspose`
- `double`
- `eq`
- `ge`
- `gt`
- `histcounts`
- `horzcat`
- `int8`
- `int16`
- `int32`

- `int64`
- `intersect`
- `iscategory`
- `iscolumn`
- `isempty`
- `isequal`
- `isequaln`
- `ismatrix`
- `ismember`
- `isordinal`
- `isprotected`
- `isrow`
- `isscalar`
- `issorted`
- `issortedrows`
- `isundefined`
- `isvector`
- `le`
- `length`
- `lt`
- `max`
- `mergecats`
- `min`
- `ndims`
- `ne`
- `numel`
- `permute`
- `removecats`
- `renamecats`
- `reordercats`
- `reshape`
- `setcats`
- `setdiff`
- `setxor`
- `single`
- `size`
- `sort`
- `sortrows`
- `transpose`

- `uint8`
- `uint16`
- `uint32`
- `uint64`
- `union`
- `unique`
- `vertcat`

See Also

More About

- "Define Categorical Array Inputs" on page 8-6
- "Categorical Array Limitations for Code Generation" on page 8-9

Define Categorical Array Inputs

You can define categorical array inputs at the command line or in the MATLAB Coder app. Programmatic specification of categorical input types by using preconditioning (`assert` statements) is not supported.

Define Categorical Array Inputs at the Command Line

Use one of these procedures:

- "Provide an Example Categorical Array Input" on page 8-6
- "Provide a Categorical Array Type" on page 8-6
- "Provide a Constant Categorical Array Input" on page 8-6

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Categorical Array Input

Use the `-args` option:

```
C = categorical({'r','g','b'});
codegen myFunction -args {C}
```

Provide a Categorical Array Type

To provide a type for a categorical array to `codegen`:

1. Define a categorical array. For example:

   ```
   C = categorical({'r','g','b'});
   ```
2. Create a type from C.

   ```
   t = coder.typeof(C);
   ```
3. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Provide a Constant Categorical Array Input

To specify that a categorical array input is constant, use `coder.Constant` with the `-args` option:

```
C = categorical({'r','g','b'});
codegen myFunction -args {coder.Constant(C)}
```

Define Categorical Array Inputs in the MATLAB Coder App

Use one of these procedures:

- "Automatically Define Input Types by Using the App" on page 24-4
- "Define Input Parameter by Example by Using the App" on page 24-6
- "Define or Edit Input Parameter Type by Using the App" on page 24-14

Representation of Categorical Arrays

A coder type object for a categorical array describes the object and its properties. Use `coder.typeof` or pass `categorical` as a string scalar to `coder.newtype`.

The coder type object displays a succinct description of the object properties while excluding internal state values. Nonconstant properties display their type and size, while constant properties display only their values. For example:

```
t = categorical({'r','g','b'});
tType = coder.typeof(t)
```

The representation of variable `t` is stored in coder type object `tType`.

```
tType =

   matlab.coder.type.CategoricalType
     1x3 categorical
    Categories : 3x1 homogeneous cell
       Ordinal : 1x1 logical
     Protected : 1x1 logical
```

If your workflow requires the legacy representation of coder type objects, use the `getCoderType` function on the variable that has the new representation of your class or object. See “Legacy Representation of Coder Type Objects” on page 4-13.

Resize Object Properties by Using coder.resize

You can resize most objects by using `coder.resize`. You can resize objects, its properties and create arrays within the properties.

For a `categorical` coder object, you can resize the object properties:

```
t = categorical({'r','g','b'});
tType = coder.typeof(t);
tType.Categories = coder.resize(tType.Categories, [3 1],[1 0])
```

This code resizes the `Categories` property to be upper-bounded at `3` for the first dimension.

```
tType =

   matlab.coder.type.CategoricalType
     1x3 categorical
    Categories : :3x1 homogeneous cell
       Ordinal : 1x1 logical
     Protected : 1x1 logical
```

You can also resize the object by using `coder.resize`. See “Edit and Represent Coder Type Objects and Properties” on page 4-12.

See Also

`categorical` | `coder.Constant` | `coder.typeof`

More About

- "Code Generation for Categorical Arrays" on page 8-2
- "Categorical Array Limitations for Code Generation" on page 8-9

Categorical Array Limitations for Code Generation

When you create categorical arrays in MATLAB code that you intend for code generation, you must specify the categories and elements of each categorical array by using the `categorical` function. See “Categorical Arrays”.

For categorical arrays, code generation does not support the following inputs and operations:

- Arrays of MATLAB objects.
- Sparse matrices.
- Duplicate category names when you specify them using the `categoryNames` input argument of the `categorical` function.
- Growth by assignment. For example, assigning a value beyond the end of an array produces an error.

```
function c = foo() %#codegen
    c = categorical(1:3,1:3,{'small','medium','large'});
    c(4) = 'medium';
end
```

- Adding a category. For example, specifying a new category by using the = operator produces an error, even when the categorical array is unprotected.

```
function c = foo() %#codegen
    c = categorical(1:3,1:3,{'small','medium','large'});
    c(1) = 'extra-large';
end
```

- Deleting an element. For example, assigning an empty array to an element produces an error.

```
function c = foo() %#codegen
    c = categorical(1:3,1:3,{'small','medium','large'});
    c(1) = [];
end
```

- Converting categorical values to text by using the `char` or `string` functions. To convert elements of a categorical array to text, use the `cellstr` function.

Limitations that apply to classes also apply to categorical arrays. For more information, see “MATLAB Classes Definition for Code Generation” on page 15-2.

See Also

`categorical` | `cellstr`

More About

- “Code Generation for Categorical Arrays” on page 8-2
- “Define Categorical Array Inputs” on page 8-6

Code Generation for Cell Arrays

- “Code Generation for Cell Arrays” on page 9-2
- “Control Whether a Cell Array Is Variable-Size” on page 9-5
- “Define Cell Array Inputs” on page 9-7
- “Cell Array Limitations for Code Generation” on page 9-8

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 9-8.

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` and `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 24-14.

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 9-5.

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 9-7. 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

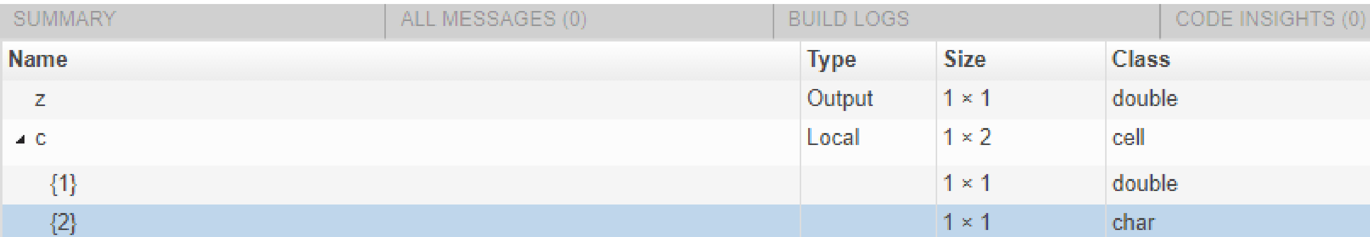
To see whether a cell array is homogeneous or heterogeneous, view the variable in the code generation report.

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.

SUMMARY | ALL MESSAGES (0) | BUILD LOGS

Name	Type	Size	Class
z	Output	1 × 1	double
◢ c	Local	1 × 3	cell
{:}		1 × 1	double

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.



See Also

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

More About

- “Control Whether a Cell Array Is Variable-Size” on page 9-5
- “Cell Array Limitations for Code Generation” on page 9-8
- “Code Generation Reports” on page 28-7

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 27-55.
- 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 9-9.
- 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 9-2
- “Cell Array Limitations for Code Generation” on page 9-8
- “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 27-52
Programmatically in the MATLAB file	"Define Input Properties Programmatically in the MATLAB File" on page 27-60
In the MATLAB Coder app	"Automatically Define Input Types by Using the App" on page 24-4 "Define Input Parameter by Example by Using the App" on page 24-6 "Define or Edit Input Parameter Type by Using the App" on page 24-14

See Also

`coder.CellType`

More About

- "Code Generation for Cell Arrays" on page 9-2

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 9-8
- “Variable-Size Cell Arrays” on page 9-9
- “Definition of Variable-Size Cell Array by Using cell” on page 9-9
- “Cell Array Indexing” on page 9-12
- “Growing a Cell Array by Using {end + 1}” on page 9-12
- “Cell Array Contents” on page 9-13
- “Passing Cell Arrays to External C/C++ Functions” on page 9-13

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
```

Variable-Size Cell Arrays

- `coder.varsize` is not supported for heterogeneous cell arrays.
- If you use the `cell` function to define a fixed-size cell array, you cannot use `coder.varsize` to specify that the cell array has a variable size. For example, this code causes a code generation error because `x = cell(1,3)` makes `x` a fixed-size,1-by-3 cell array.

  ```
  ...
  x = cell(1,3);
  coder.varsize('x',[1 5])
  ...
  ```

  You can use `coder.varsize` with a cell array that you define by using curly braces. For example:

  ```
  ...
  x = {1 2 3};
  coder.varsize('x',[1 5])
  ...
  ```
- To create a variable-size cell array by using the `cell` function, use this code pattern:

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

  See "Definition of Variable-Size Cell Array by Using cell" on page 9-9.

  To specify upper bounds for the cell array, use `coder.varsize`.

  ```
  function mycell(n)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  coder.varsize('x',[1,20]);
  end
  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 an error message. For example, modify the upper bound of the `for`-loop to `j`.

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

With this modification and with inputs `j` less than `n`, the function does not assign values to all of the cell array elements. Code generation produces the error:

```
Unable to determine that every element of 'x{:}' 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 36-10.

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 on page 9-2.

  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
  ...
  ```

Cell Array Contents

Cell arrays cannot contain `mxarrays`. In a cell array, you cannot store a value that an extrinsic function returns.

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.

See Also

More About

- "Code Generation for Cell Arrays" on page 9-2
- "Differences Between Generated Code and MATLAB Code" on page 2-6

Code Generation for Datetime Arrays

- "Code Generation for Datetime Arrays" on page 10-2
- "Define Datetime Array Inputs" on page 10-5
- "Datetime Array Limitations for Code Generation" on page 10-7

Code Generation for Datetime Arrays

In this section...
“Define Datetime Arrays for Code Generation” on page 10-2 “Allowed Operations on Datetime Arrays” on page 10-2 “MATLAB Toolbox Functions That Support Datetime Arrays” on page 10-3

The values in a `datetime` array represent points in time using the proleptic ISO calendar.

When you use `datetime` arrays with code generation, adhere to these restrictions.

Define Datetime Arrays for Code Generation

For code generation, use the `datetime` function to create `datetime` arrays. For example, suppose the input arguments to your MATLAB function are numeric arrays whose values indicate the year, month, day, hour, minute, and second components for a point in time. You can create a `datetime` array from these input arrays.

```
function d = foo(y,mo,d,h,mi,s) %#codegen
    d = datetime(y,mo,d,h,mi,s);
end
```

Allowed Operations on Datetime Arrays

For code generation, you are restricted to the operations on `datetime` arrays listed in this table.

Operation	Example	Notes

Assignment operator: =	d = datetime(2019,1:12,1,12,0,0);
	d(1) = datetime(2019,1,31);
	
	d = datetime(2019,1:12,1,12,0,0);
	d(1) = datetime(2019,1,31);
		Code generation does not support using the assignment operator = to:
		• Delete an element.
		• Expand the size of a `datetime` array.

Relational operators: < > <= >= == ~=	d = datetime(2019,1:12,1,12,0,0);
	tf = d(1) < d(2);
	
	d = datetime(2019,1:12,1,12,0,0);
	tf = d(1) < d(2);
		Code generation supports relational operators.

Indexing operation	d = datetime(2019,1:12,1,12,0,0);
	idx = [1 2];
	d(idx);
	idx = logical([1 1 0]);
	d(idx);
	
	d = datetime(2019,1:12,1,12,0,0);
	idx = [1 2];
	d(idx);
	idx = logical([1 1 0]);
	d(idx);
		Code generation supports indexing by position, linear indexing, and logical indexing.

Operation	Example	Notes

Concatenation	`d1 = datetime(2019,1:6,1,12,0,0);`
	`d2 = datetime(2019,7:12,1,12,0,0);`
	`d = [d1 d2];`
	
	`d1 = datetime(2019,1:6,1,12,0,0);`
	`d2 = datetime(2019,7:12,1,12,0,0);`
	`d = [d1 d2];`
		Code generation supports concatenation of `datetime` arrays.

MATLAB Toolbox Functions That Support Datetime Arrays

For code generation, you can use `datetime` arrays with these MATLAB toolbox functions:

- `cat`
- `colon`
- `ctranspose`
- `datetime`
- `datevec`
- `diff`
- `eq`
- `ge`
- `gt`
- `hms`
- `horzcat`
- `hour`
- `interp1`
- `intersect`
- `iscolumn`
- `isempty`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismatrix`
- `ismember`
- `isnat`
- `isreal`
- `isrow`
- `isscalar`
- `issorted`
- `issortedrows`
- `isvector`
- `le`

- `length`
- `linspace`
- `lt`
- `max`
- `mean`
- `min`
- `minus`
- `minute`
- `NaT`
- `ndims`
- `ne`
- `numel`
- `permute`
- `plus`
- `posixtime`
- `repmat`
- `reshape`
- `setdiff`
- `setxor`
- `size`
- `sort`
- `sortrows`
- `topkrows`
- `transpose`
- `union`
- `unique`
- `vertcat`
- `ymd`

See Also

More About

- “Define Datetime Array Inputs” on page 10-5
- “Datetime Array Limitations for Code Generation” on page 10-7

Define Datetime Array Inputs

You can define `datetime` array inputs at the command line or in the MATLAB Coder app. Programmatic specification of `datetime` input types by using preconditioning (`assert` statements) is not supported.

Define Datetime Array Inputs at the Command Line

Use one of these procedures:

- "Provide an Example Datetime Array Input" on page 10-5
- "Provide a Datetime Array Type" on page 10-5
- "Provide a Constant Datetime Array Input" on page 10-5

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Datetime Array Input

Use the `-args` option:

```
D = datetime(2019,1:12,1,12,0,0);
codegen myFunction -args {D}
```

Provide a Datetime Array Type

To provide a type for a `datetime` array to `codegen`:

1. Define a `datetime` array. For example:

   ```
   D = datetime(2019,1:12,1,12,0,0);
   ```
2. Create a type from `D`.

   ```
   t = coder.typeof(D);
   ```
3. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Provide a Constant Datetime Array Input

To specify that a `datetime` array input is constant, use `coder.Constant` with the `-args` option:

```
D = datetime(2019,1:12,1,12,0,0);
codegen myFunction -args {coder.Constant(C)}
```

Define Datetime Array Inputs in the MATLAB Coder App

Use one of these procedures:

- "Automatically Define Input Types by Using the App" on page 24-4
- "Define Input Parameter by Example by Using the App" on page 24-6
- "Define or Edit Input Parameter Type by Using the App" on page 24-14

Representation of Datetime Arrays

A coder type object for a datetime array describes the object and its properties. Use `coder.typeof` or pass `datetime` as a string scalar to `coder.newtype`.

The coder type object displays a succinct description of the object properties while excluding internal state values. Nonconstant properties display their type and size, while constant properties display only their values. For example:

```
t = datetime(2019,1:12,1,12,0,0);
tType = coder.typeof(t)
```

The representation of variable `t` is stored in coder type object `tType`.

```
tType = 

   matlab.coder.type.DatetimeType
     1x12 datetime
      Format : 1x0 char
    TimeZone : 1x0 char
```

If your workflow requires the legacy representation of coder type objects, use the `getCoderType` function on the variable that has the new representation of your class or object. See "Legacy Representation of Coder Type Objects" on page 4-13.

Resize Object Properties by Using coder.resize

You can resize most objects by using `coder.resize`. You can resize objects, its properties and create arrays within the properties.

For a `datetime` coder object, you can resize the object properties:

```
t = datetime(2019,1:12,1,12,0,0);
tType = coder.typeof(t)
tType.Format = coder.resize(tType.Format, [1 12])
```

This code resizes the `Format` property to be a `1x12 char` property.

```
tType = 

   matlab.coder.type.DatetimeType
     1x12 datetime
      Format : 1x12 char
    TimeZone : 1x0 char
```

You can also resize the object by using `coder.resize`. See "Edit and Represent Coder Type Objects and Properties" on page 4-12.

See Also

`NaT` | `coder.Constant` | `coder.typeof` | `datetime`

More About

- "Code Generation for Datetime Arrays" on page 10-2
- "Datetime Array Limitations for Code Generation" on page 10-7

Datetime Array Limitations for Code Generation

When you create `datetime` arrays in MATLAB code that you intend for code generation, you must specify the values by using the `datetime` function. See “Dates and Time”.

For `datetime` arrays, code generation does not support the following inputs and operations:

- Text inputs. For example, specifying a character vector as the input argument produces an error.

```
function d = foo() %#codegen
    d = datetime('2019-12-01');
end
```

- The `'Format'` name-value pair argument. You cannot specify the display format by using the `datetime` function, or by setting the `Format` property of a `datetime` array. To use a specific display format, create a `datetime` array in MATLAB, then pass it as an input argument to a function that is intended for code generation.
- The `'TimeZone'` name-value pair argument and the `TimeZone` property. When you use `datetime` arrays in code that is intended for code generation, they must be unzoned.
- Setting time component properties. For example, setting the `Hour` property in the following code produces an error:

```
d = datetime;
d.Hour = 2;
```

- Growth by assignment. For example, assigning a value beyond the end of an array produces an error.

```
function d = foo() %#codegen
    d = datetime(2019,1:12,1,12,0,0);
    d(13) = datetime(2020,1,1,12,0,0);
end
```

- Deleting an element. For example, assigning an empty array to an element produces an error.

```
function d = foo() %#codegen
    d = datetime(2019,1:12,1,12,0,0);
    d(1) = [];
end
```

- Converting `datetime` values to text by using the `char`, `cellstr`, or `string` functions.

Limitations that apply to classes also apply to `datetime` arrays. For more information, see “MATLAB Classes Definition for Code Generation” on page 15-2.

See Also

`NaT` | `datetime`

More About

- “Code Generation for Datetime Arrays” on page 10-2
- “Define Datetime Array Inputs” on page 10-5

Code Generation for Duration Arrays

- “Code Generation for Duration Arrays” on page 11-2
- “Define Duration Array Inputs” on page 11-6
- “Duration Array Limitations for Code Generation” on page 11-8

Code Generation for Duration Arrays

In this section...

"Define Duration Arrays for Code Generation" on page 11-2

"Allowed Operations on Duration Arrays" on page 11-2

"MATLAB Toolbox Functions That Support Duration Arrays" on page 11-3

The values in a duration array represent elapsed times in units of fixed length, such as hours, minutes, and seconds. You can create elapsed times in terms of fixed-length (24-hour) days and fixed-length (365.2425-day) years.

You can add, subtract, sort, compare, concatenate, and plot duration arrays.

When you use duration arrays with code generation, adhere to these restrictions.

Define Duration Arrays for Code Generation

For code generation, use the `duration` function to create duration arrays. For example, suppose the input arguments to your MATLAB function are three numeric arrays of arbitrary size whose elements specify lengths of time as hours, minutes, and seconds. You can create a duration array from these three input arrays.

```
function d = foo(h,m,s) %#codegen
    d = duration(h,m,s);
end
```

You can use the `years`, `days`, `hours`, `minutes`, `seconds`, and `milliseconds` functions to create duration arrays in units of years, days, hours, minutes, or seconds. For example, you can create an array of hours from an input numeric array.

```
function d = foo(h) %#codegen
    d = hours(h);
end
```

Allowed Operations on Duration Arrays

For code generation, you are restricted to the operations on duration arrays listed in this table.

Operation	Example	Notes

assignment operator: =	`d = duration(1:3,0,0);`
	`d(1) = hours(5);`
	
	`d = duration(1:3,0,0);`
	`d(1) = hours(5);`
		Code generation does not support using the assignment operator = to:
		• Delete an element.
		• Expand the size of a duration array.

relational operators: < > <= >= == ~=	`d = duration(1:3,0,0);`
	`tf = d(1) < d(2);`
	
	`d = duration(1:3,0,0);`
	`tf = d(1) < d(2);`
		Code generation supports relational operators.

Operation	Example	Notes

indexing operation	`d = duration(1:3,0,0);`
	`idx = [1 2];`
	`d(idx);`
	`idx = logical([1 1 0]);`
	`d(idx);`
	
	`d = duration(1:3,0,0);`
	`idx = [1 2];`
	`d(idx);`
	`idx = logical([1 1 0]);`
	`d(idx);`
		Code generation supports indexing by position, linear indexing, and logical indexing.

concatenation	`d1 = duration(1:3,0,0);`
	`d2 = duration(4,30,0);`
	`d = [d1 d2];`
	
	`d1 = duration(1:3,0,0);`
	`d2 = duration(4,30,0);`
	`d = [d1 d2];`
		Code generation supports concatenation of duration arrays.

MATLAB Toolbox Functions That Support Duration Arrays

For code generation, you can use duration arrays with these MATLAB toolbox functions:

- `abs`
- `cat`
- `ceil`
- `colon`
- `cummax`
- `cummin`
- `cumsum`
- `ctranspose`
- `datevec`
- `days`
- `diff`
- `duration`
- `eps`
- `eq`
- `floor`
- `ge`
- `gt`
- `hms`
- `horzcat`
- `hours`
- `interp1`
- `intersect`
- `iscolumn`

- `isempty`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismatrix`
- `ismember`
- `isnan`
- `isreal`
- `isrow`
- `isscalar`
- `issorted`
- `issortedrows`
- `isvector`
- `ldivide`
- `le`
- `length`
- `linspace`
- `lt`
- `max`
- `mean`
- `median`
- `milliseconds`
- `min`
- `minus`
- `minutes`
- `mldivide`
- `mode`
- `mrdivide`
- `mod`
- `mtimes`
- `ndims`
- `ne`
- `nnz`
- `numel`
- `permute`
- `plus`
- `repmat`
- `rdivide`

- `rem`
- `reshape`
- `seconds`
- `setdiff`
- `setxor`
- `sign`
- `size`
- `sort`
- `sortrows`
- `std`
- `sum`
- `times`
- `transpose`
- `uminus`
- `union`
- `unique`
- `uplus`
- `vertcat`
- `years`

See Also

More About

- "Define Duration Array Inputs" on page 11-6
- "Duration Array Limitations for Code Generation" on page 11-8

Define Duration Array Inputs

You can define duration array inputs at the command line or in the MATLAB Coder app. Programmatic specification of duration input types by using preconditioning (`assert` statements) is not supported.

Define Duration Array Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Duration Array Input” on page 11-6
- “Provide a Duration Array Type” on page 11-6
- “Provide a Constant Duration Array Input” on page 11-6

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Duration Array Input

Use the `-args` option:

```
D = duration(1:3,0,0);
codegen myFunction -args {D}
```

Provide a Duration Array Type

To provide a type for a duration array to `codegen`:

1. Define a duration array. For example:

   ```
   D = duration(1:3,0,0);
   ```
2. Create a type from `D`.

   ```
   t = coder.typeof(D);
   ```
3. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Provide a Constant Duration Array Input

To specify that a duration array input is constant, use `coder.Constant` with the `-args` option:

```
D = duration(1:3,0,0);
codegen myFunction -args {coder.Constant(C)}
```

Define Duration Array Inputs in the MATLAB Coder App

Use one of these procedures:

- “Automatically Define Input Types by Using the App” on page 24-4
- “Define Input Parameter by Example by Using the App” on page 24-6
- “Define or Edit Input Parameter Type by Using the App” on page 24-14

Representation of Duration Arrays

A coder type object for a duration array describes the object and its properties. Use `coder.typeof` or pass `duration` as a string scalar to `coder.newtype`.

The coder type object displays a succinct description of the object properties while excluding internal state values. Nonconstant properties display their type and size, while constant properties display only their values. For example:

```
tType = coder.newtype('duration')
```

A representation of an empty duration variable is stored in coder type object `tType`.

```
tType = 

   matlab.coder.type.DurationType
     1x1 duration
    Format : 1x8 char
```

If your workflow requires the legacy representation of coder type objects, use the `getCoderType` function on the variable that has the new representation of your class or object. See “Legacy Representation of Coder Type Objects” on page 4-13.

Resize duration Properties by Editing Object Properties

You can resize most objects by editing the object properties. You can resize `duration` objects, its properties and create arrays within the properties.

For a `duration` coder object, you can resize the object properties:

```
t = duration((1:3),0,0);
tType = coder.typeof(t)
tType.Format = 'DD/MM/YYYY'
```

This code resizes the `Format` property to be a `1x10 char` property.

```
tType = 

   matlab.coder.type.DurationType
     1x3 duration
    Format : 1x10 char
```

You can also resize the object by using `coder.resize`. See “Edit and Represent Coder Type Objects and Properties” on page 4-12.

See Also
`coder.Constant` | `coder.typeof` | `duration`

More About

- “Code Generation for Duration Arrays” on page 11-2
- “Duration Array Limitations for Code Generation” on page 11-8

Duration Array Limitations for Code Generation

When you create duration arrays in MATLAB code that you intend for code generation, you must specify the durations by using the `duration`, `years`, `days`, `hours`, `minutes`, `seconds`, or `milliseconds` functions. See “Dates and Time”.

For duration arrays, code generation does not support the following inputs and operations:

- Text inputs. For example, specifying a character vector as the input argument produces an error.

  ```
  function d = foo() %#codegen
      d = duration('01:30:00');
  end
  ```

- Growth by assignment. For example, assigning a value beyond the end of an array produces an error.

  ```
  function d = foo() %#codegen
      d = duration(1:3,0,0);
      d(4) = hours(4);
  end
  ```

- Deleting an element. For example, assigning an empty array to an element produces an error.

  ```
  function d = foo() %#codegen
      d = duration(1:3,0,0);
      d(1) = [];
  end
  ```

- Converting duration values to text by using the `char`, `cellstr`, or `string` functions.

Limitations that apply to classes also apply to duration arrays. For more information, see “MATLAB Classes Definition for Code Generation” on page 15-2.

See Also

`days` | `duration` | `hours` | `milliseconds` | `minutes` | `seconds` | `years`

More About

- “Code Generation for Duration Arrays” on page 11-2
- “Define Duration Array Inputs” on page 11-6

Code Generation for Tables

- "Code Generation for Tables" on page 12-2
- "Define Table Inputs" on page 12-5
- "Table Limitations for Code Generation" on page 12-8

Code Generation for Tables

In this section...
"Define Tables for Code Generation" on page 12-2
"Allowed Operations on Tables" on page 12-2
"MATLAB Toolbox Functions That Support Tables" on page 12-3

The `table` data type is a data type suitable for column-oriented or tabular data that is often stored as columns in a text file or in a spreadsheet. Tables consist of rows and column-oriented variables. Each variable in a table can have a different data type and a different size with one restriction: each variable must have the same number of rows. For more information, see "Tables".

When you use tables with code generation, adhere to these restrictions.

Define Tables for Code Generation

For code generation, use the `table` function. For example, suppose the input arguments to your MATLAB function are three arrays that have the same number of rows and a cell array that has variable names. You can create a table that contains these arrays as table variables.

```
function T = foo(A,B,C,vnames) %#codegen
    T = table(A,B,C,'VariableNames',vnames);
end
```

You can use the `array2table`, `cell2table`, and `struct2table` functions to convert arrays, cell arrays, and structures to tables. For example, you can convert an input cell array to a table.

```
function T = foo(C,vnames) %#codegen
    T = cell2table(C,'VariableNames',vnames);
end
```

For code generation, you must supply table variable names when you create a table. Table variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).

Allowed Operations on Tables

For code generation, you are restricted to the operations on tables listed below.

Operation	Example	Notes

assignment operator: =	`T = table(A,B,C,'VariableNames',vnames);`
	`T{:,1} = D;`
	`T = table(A,B,C,'VariableNames',vnames);`
	`T{:,1} = D;`
		Code generation does not support using the assignment operator = to:
		• Delete a variable or a row.
		• Add a variable or a row.

Operation	Example	Notes

indexing operation	`T = table(A,B,C,'VariableNames',vnames);`
	`T(1:5,1:3);`
	
	`T = table(A,B,C,'VariableNames',vnames);`
	`T(1:5,1:3);`
		Code generation supports indexing by position, variable or row name, and logical indexing.
		
		Code generation supports:
		• Table indexing with smooth parentheses, ().
		• Content indexing with curly braces, {}.
		• Dot notation to access a table variable.

concatenation	`T1 = table(A,B,C,'VariableNames',vnames);`
	`T2 = table(D,E,F,'VariableNames',vnames);`
	`T = [T1 ; T2];`
	
	`T1 = table(A,B,C,'VariableNames',vnames);`
	`T2 = table(D,E,F,'VariableNames',vnames);`
	`T = [T1 ; T2];`
		Code generation supports table concatenation.
		• For vertical concatenation, tables must have variables that have the same names in the same order.
		• For horizontal concatenation, tables must have the same number of rows. If the tables have row names, then they must have the same row names in the same order.

MATLAB Toolbox Functions That Support Tables

For code generation, you can use tables with these MATLAB toolbox functions:

- `addvars`
- `array2table`
- `cat`
- `cell2table`
- `convertvars`
- `height`
- `horzcat`
- `intersect`
- `isempty`
- `ismember`
- `issortedrows`
- `join`
- `movevars`
- `ndims`

- `numel`
- `removevars`
- `renamevars`
- `rows2vars`
- `setdiff`
- `setxor`
- `size`
- `sortrows`
- `splitvars`
- `stack`
- `struct2table`
- `table`
- `table2array`
- `table2cell`
- `table2struct`
- `union`
- `unique`
- `unstack`
- `varfun`
- `vertcat`
- `width`

See Also

More About

- "Define Table Inputs" on page 12-5
- "Table Limitations for Code Generation" on page 12-8

Define Table Inputs

You can define table inputs at the command line or in the MATLAB Coder app. Programmatic specification of table input types by using preconditioning (`assert` statements) is not supported.

Define Table Inputs at the Command Line

Use one of these procedures:

- “Provide an Example Table Input” on page 12-5
- “Provide a Table Type” on page 12-5
- “Provide a Constant Table Input” on page 12-5

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Table Input

Use the `-args` option:

```
T = table(A,B,C,'VariableNames',vnames);
codegen myFunction -args {T}
```

Provide a Table Type

To provide a type for a table to `codegen`:

1. Define a table. For example:

   ```
   T = table(A,B,C,'VariableNames',vnames);
   ```
2. Create a type from `T`.

   ```
   t = coder.typeof(T);
   ```
3. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Provide a Constant Table Input

To specify that a table input is constant, use `coder.Constant` with the `-args` option:

```
T = table(A,B,C,'VariableNames',vnames);
codegen myFunction -args {coder.Constant(T)}
```

Define Table Inputs in the MATLAB Coder App

Use one of these procedures:

- “Automatically Define Input Types by Using the App” on page 24-4
- “Define Input Parameter by Example by Using the App” on page 24-6
- “Define or Edit Input Parameter Type by Using the App” on page 24-14

Representation of Tables

A coder type object for a table describes the object and its properties. Use `coder.typeof` or pass `table` as a string scalar to `coder.newtype`.

The coder type object displays a succinct description of the object properties while excluding internal state values. Nonconstant properties display their type and size, while constant properties display only their values. For example:

```
A = [1 2 3]';
B = [4 5 6]';
C = [7 8 9]';
t = table(A,B,C);
tType = coder.typeof(t)
```

The representation of variable `t` is stored in coder type object `tType`.

```
tType =

   matlab.coder.type.TableType
     3x3 table
                    Data : 1x3 homogeneous cell
             Description : 1x0 char
                UserData : 0x0 double
          DimensionNames : {'Row'}    {'Variables'}
           VariableNames : {'A'}    {'B'}    {'C'}
    VariableDescriptions : 1x3 homogeneous cell
           VariableUnits : 1x3 homogeneous cell
      VariableContinuity : 1x3 matlab.internal.coder.tabular.Continuity
                RowNames : 0x0 homogeneous cell
```

If your workflow requires the legacy representation of coder type objects, use the `getCoderType` function on the variable that has the new representation of your class or object. See “Legacy Representation of Coder Type Objects” on page 4-13.

Resize Object Properties by Using coder.resize

You can resize most objects by using `coder.resize`. You can resize objects, its properties and create arrays within the properties.

For a `table` coder object, you can resize the object properties:

```
A = [1 2 3]';
B = [4 5 6]';
C = [7 8 9]';
t = table(A,B,C);
tType = coder.typeof(t)
tType.Description = coder.resize(tType.Description,[1 12],[0 1])
```

This code resizes the `Description` property to be a `1x:12 char` property which has an upper bound of `12`.

```
tType =

   matlab.coder.type.TableType
     3x3 table
                    Data : 1x3 homogeneous cell
```

```
            Description : 1x:12 char
               UserData : 0x0 double
         DimensionNames : {'Row'}    {'Variables'}
          VariableNames : {'A'}    {'B'}    {'C'}
   VariableDescriptions : 1x3 homogeneous cell
          VariableUnits : 1x3 homogeneous cell
     VariableContinuity : 1x3 matlab.internal.coder.tabular.Continuity
               RowNames : 0x0 homogeneous cell
```

You can also resize the object by using `coder.resize`. See “Edit and Represent Coder Type Objects and Properties” on page 4-12.

See Also

`coder.Constant` | `coder.typeof` | `table`

More About

- “Code Generation for Tables” on page 12-2
- “Table Limitations for Code Generation” on page 12-8

Table Limitations for Code Generation

If you create tables, modify them, or use table functions in MATLAB code that you intend for code generation, then code generation has limitations described in the next sections. Limitations that apply to classes also apply to tables. For more information on class limitations, see "MATLAB Classes Definition for Code Generation" on page 15-2.

Creating Tables Limitations

If your MATLAB code creates tables, then code generation has these limitations.

Inputs for Table Creation	Limitations
Any inputs	• Table variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, which can include commas, dashes, and space characters.
Table created from input arrays	• You must specify variables names by using the `'VariableNames'` name-value argument when creating tables from input arrays by using the `table`, `array2table`, or `cell2table` functions.
Table created with preallocated variables	• You do not have to specify the `'VariableNames'` argument when you preallocate a table by using the `table` function and the `'Size'` name-value argument. • You can specify only the following data types by using the `'VariableTypes'` name-value argument: • `'double'` • `'single'` • `'doublenan'` or `'doubleNaN'` • `'singlenan'` or `'singleNaN'` • `'int8'`, `'int16'`, `'int32'`, or `'int64'` • `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'` • `'logical'` • `'duration'` • `'cellstr'` • `'char'`

Modifying Tables Limitations

If your MATLAB code modifies data in a table or its properties, then code generation has these limitations.

Table Operation or Property	Limitations
`VariableNames`, `RowNames`, `DimensionNames`, or `UserData` properties	• You cannot change the `VariableNames`, `RowNames`, `DimensionNames`, or `UserData` properties of a table after you create it. You can specify the `'VariableNames'`, `'RowNames'`, and `'DimensionNames'` input arguments when you create a table. These input arguments specify the properties.
Table indices that specify variables as input arguments to generated code	• To pass table indices that specify variables as input arguments into generated code, first make the indices constant by using the `coder.Constant` function. If table indices are not constant, then indexing into variables produces an error.
Custom metadata	• You cannot add custom metadata to a table. The `addprop` and `rmprop` functions are not supported.

Assignments that change size of table	• You cannot change the size of a table by assignments. For example, adding a new row produces an error.
	
	`function T = foo() %#codegen`
	`    T = table((1:3)',(1:3)','VariableNames',{'Var1`
	`    T(4,2) = 5;`
	`end`
	
	Deleting a row or a variable also produces an error.

Vertical concatenation	• When you vertically concatenate tables, they must have the same variable names in the same order. In MATLAB, the variable names must be the same but can be in different orders in the tables.
Horizontal concatenation	• When you horizontally concatenate tables and the tables have row names, they must have the same row names in the same order. In MATLAB, the row names must be the same but can be in different orders in the tables.
Table variables that are N-D cell arrays	• If two tables have variables that are N-D cell arrays, then the tables cannot be vertically concatenated. • You cannot use curly braces to extract data from multiple table variables that are N-D cell arrays because this operation is horizontal concatenation.

Using Table Functions Limitations

If your MATLAB code uses the functions listed in the table, then code generation has these limitations.

Function	Limitations
`convertvars`	• Function handles are not supported. • The second and third input arguments (`vars` and `dataType`) must be constant. • You cannot specify `dataType` as `'char'`.

`intersect`
`setdiff`
`setxor`
`union`
	• These functions support unsorted tables in all cases. You do not have to specify the `'stable'` option.

`issortedrows`	• The input argument `vars` must be constant. • If any table variables have multiple columns, then those variables must have fixed widths.

`join`	• In general, input tables cannot have nonkey variables with the same names. However, you can specify duplicated variable names in the input tables by using:
	  • `'KeepOneCopy'`
	  • `'LeftVariables'` and `'RightVariables'`
	• The values of these name-value arguments must be constant:
	  • `'Keys'`
	  • `'LeftKeys'`
	  • `'RightKeys'`
	  • `'LeftVariables'`
	  • `'RightVariables'`
	  • `'KeepOneCopy'`
	• Nested tables are not supported.

`movevars`	• The input argument `vars` cannot contain duplicate variable names.

Function	Limitations
`rows2vars`	• The input table cannot be variable-size. • The `'VariableNamesSource'` name-value argument is not supported. • The value of the `'DataVariables'` name-value argument must be constant. • The value of the `'VariableNamingRule'` name-value argument must be constant. • If you assign row names to the input table, then the vector of row names must be constant.
`sortrows`	• The input argument `vars` must be constant. • If `tblA` has a variable that is a cell array of character vectors with multiple columns, then you cannot sort the table using the values in that variable.
`splitvars`	• The value of the `'NewVariableNames'` name-value argument must be constant. • The variables that are split cannot have a variable number of columns.
`stack`	• The second input argument, `vars`, must be constant. • The values of the `'ConstantVariables'`, `'NewDataVariableName'`, and `'IndexVariableName'` name-value arguments must be constant.

Function	Limitations
`unstack`	• The `'NewDataVariableNames'` name-value argument must be specified. Its value must be constant. • The `vars` and `ivars` input arguments (data variables and indicator variables) must be constant. • If you specify grouping variables and constant variables, then they must be constant. • If you specify an aggregation function, then it must be constant. • If a variable of the input table is a cell array of character vectors, then `unstack` fills empty cells in the corresponding output variable with 1-by-0 character arrays in the generated code. In MATLAB, `unstack` fills such gaps with 0-by-0 character arrays. • The `unstack` function does not support code generation when the input table has a variable that is a heterogeneous cell array that cannot be converted to a homogeneous cell array. • If the input has a variable that is a homogeneous cell array, or that can be converted to one, then the `'AggregationFunction'` name-value argument must be specified. The default value of `'AggregationFunction'` is `'unique'`. But the `unique` function does not support cell arrays.

Function	Limitations
varfun	• The function handle input, func, must be constant. • While function handles can be inputs to varfun itself, they cannot be inputs to your entry point functions. Specify func within the code meant for code generation. For more information, see “Function Handle Limitations for Code Generation” on page 17-2. • The values for all name-value arguments must be constant. • The 'ErrorHandler' name-value argument is not supported for code generation. • Variable-size input arguments are not supported. • Grouping variables cannot have duplicate values in generated code. • You cannot specify the value of 'OutputFormat' as 'cell' if you specify the 'GroupingVariables' name-value argument and the function returns a different data type for each variable specified by 'InputVariables'. • If you specify groups and the number of groups is not known at compile time, and that number is zero, then empty double variables in the output might have sizes of 1-by-0 in generated code. In MATLAB, such variables have sizes of 0-by-0.

See Also

array2table | cell2table | struct2table | table

More About

- “Code Generation for Tables” on page 12-2
- “Define Table Inputs” on page 12-5

Code Generation for Timetables

- “Code Generation for Timetables” on page 13-2
- “Define Timetable Inputs” on page 13-5
- “Timetable Limitations for Code Generation” on page 13-8

Code Generation for Timetables

In this section...

"Define Timetables for Code Generation" on page 13-2
"Allowed Operations on Timetables" on page 13-2
"MATLAB Toolbox Functions That Support Timetables" on page 13-3

The `timetable` data type is a data type suitable for tabular data with time-stamped rows. Like tables, timetables consist of rows and column-oriented variables. Each variable in a timetable can have a different data type and a different size with one restriction: each variable must have the same number of rows.

The *row times* of a timetable are time values that label the rows. You can index into a timetable by row time and variable. To index into a timetable, use smooth parentheses `()` to return a subtable or curly braces `{}` to extract the contents. You can refer to variables and to the vector of row times by their names. For more information, see "Timetables".

When you use timetables with code generation, adhere to these restrictions.

Define Timetables for Code Generation

For code generation, use the `timetable` function. For example, suppose the input arguments to your MATLAB function are three arrays that have the same number of rows (`A`, `B`, and `C`), a `datetime` or `duration` vector containing row times (`D`), and a cell array that has variable names (`vnames`). You can create a timetable that contains these arrays as timetable variables.

```
function TT = foo(A,B,C,D,vnames) %#codegen
    TT = table(A,B,C,'RowTimes',D,'VariableNames',vnames);
end
```

To convert arrays and tables to timetables, use the `array2timetable` and `table2timetable` functions. For example, you can convert an input `M`-by-`N` matrix to a timetable, where each column of the matrix becomes a variable in the timetable. Assign row times by using a `duration` vector.

```
function TT = foo(A,D,vnames) %#codegen
    TT = array2timetable(A,'RowTimes',D,'VariableNames',vnames);
end
```

For code generation, you must supply timetable variable names when you create a timetable. Timetable variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, but can include any ASCII characters (such as commas, dashes, and space characters).

The row times can have either the `datetime` or `duration` data type.

Allowed Operations on Timetables

For code generation, you are restricted to the operations on timetables listed in this table.

Operation	Example	Notes
Assignment operator: =	`TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);` `TT{:,1} = X;`	Code generation does not support using the assignment operator = to: • Delete a variable or a row. • Add a variable or a row.

Indexing operation	`D = seconds(1:10);`
	`TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);`
	`TT(seconds(3:7),1:3);`
		Code generation supports indexing by position, variable or row time, and logical indexing. Also, you can index using objects created by using the `timerange` or `withtol` functions.
		
		Code generation supports:
		• Timetable indexing with smooth parentheses, ().
		• Content indexing with curly braces, {}.
		• Dot notation to access a timetable variable.

Concatenation	`TT1 = timetable(A,B,C,'RowTimes',D1,'VariableNames',vnames);`
	`TT2 = timetable(D,E,F,'RowTimes',D2,'VariableNames',vnames);`
	`TT = [TT1 ; TT2];`
		Code generation supports timetable concatenation.
		• For vertical concatenation, timetables must have variables that have the same names in the same order.
		• For horizontal concatenation, timetables must have the same number of rows. They also must have the same row times in the same order.

MATLAB Toolbox Functions That Support Timetables

For code generation, you can use timetables with these MATLAB toolbox functions:

- `addvars`
- `array2timetable`
- `cat`
- `convertvars`
- `height`
- `horzcat`
- `intersect`

- `isempty`
- `ismember`
- `isregular`
- `issorted`
- `issortedrows`
- `join`
- `movevars`
- `ndims`
- `numel`
- `removevars`
- `renamevars`
- `rows2vars`
- `retime`
- `setdiff`
- `setxor`
- `size`
- `sortrows`
- `splitvars`
- `stack`
- `synchronize`
- `table2timetable`
- `timerange`
- `timetable`
- `timetable2table`
- `union`
- `unique`
- `unstack`
- `varfun`
- `vertcat`
- `width`
- `withtol`

See Also

More About

- “Define Timetable Inputs” on page 13-5
- “Timetable Limitations for Code Generation” on page 13-8

Define Timetable Inputs

You can define timetable inputs at the command line or in the MATLAB Coder app. Programmatic specification of timetable input types by using preconditioning (`assert` statements) is not supported.

Define Timetable Inputs at the Command Line

Use one of these procedures:

- "Provide an Example Timetable Input" on page 13-5
- "Provide a Timetable Type" on page 13-5
- "Provide a Constant Timetable Input" on page 13-5

Alternatively, if you have a test file that calls your entry-point function with example inputs, you can determine the input types by using `coder.getArgTypes`.

Provide an Example Timetable Input

Use the `-args` option:

```
TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);
codegen myFunction -args {TT}
```

Provide a Timetable Type

To provide a type for a timetable to `codegen`:

1. Define a timetable. For example:

   ```
   TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);
   ```
2. Create a type from T.

   ```
   t = coder.typeof(TT);
   ```
3. Pass the type to `codegen` by using the `-args` option.

   ```
   codegen myFunction -args {t}
   ```

Provide a Constant Timetable Input

To specify that a timetable input is constant, use `coder.Constant` with the `-args` option:

```
TT = timetable(A,B,C,'RowTimes',D,'VariableNames',vnames);
codegen myFunction -args {coder.Constant(TT)}
```

Define Timetable Inputs in the MATLAB Coder App

Use one of these procedures:

- "Automatically Define Input Types by Using the App" on page 24-4
- "Define Input Parameter by Example by Using the App" on page 24-6
- "Define or Edit Input Parameter Type by Using the App" on page 24-14

Representation of Timetables

A coder type object for a timetable describes the object and its properties. Use `coder.typeof` or pass `timetable` as a string scalar to `coder.newtype`.

The coder type object displays a succinct description of the object properties while excluding internal state values. Nonconstant properties display their type and size, while constant properties display only their values. For example:

```
t = timetable((1:5)',(11:15)','SampleRate',1);
tType = coder.typeof(t)
```

The representation of variable `t` is stored in coder type object `tType`.

```
tType = 

   matlab.coder.type.RegularTimetableType
     5x2 timetable
                    Data : 1x2 homogeneous cell
             Description : 1x0 char
                UserData : 0x0 double
          DimensionNames : {'Time'}    {'Variables'}
           VariableNames : {'Var1'}    {'Var2'}
    VariableDescriptions : 1x2 homogeneous cell
           VariableUnits : 1x2 homogeneous cell
      VariableContinuity : 1x2 matlab.internal.coder.tabular.Continuity
               StartTime : 1x1 matlab.coder.type.DurationType
              SampleRate : 1x1 double
                TimeStep : 1x1 matlab.coder.type.DurationType
```

Define a regular `timetable` by specifying the `SampleRate` or `TimeStep`. You can also define an irregular `timetable` by specifying the `RowTimes`. For example:

```
t1 = timetable((1:3),'RowTimes',seconds(1:3));
t1Type = coder.typeof(t)
```

The representation of irregular table `t1` is stored in coder type object `t1Type`.

```
t1Type = 

   matlab.coder.type.TimetableType
     3x1 timetable
                    Data : 1x1 homogeneous cell
             Description : 1x0 char
                UserData : 0x0 double
          DimensionNames : {'Time'}    {'Variables'}
           VariableNames : {'Var1'}
    VariableDescriptions : 1x1 homogeneous cell
           VariableUnits : 1x1 homogeneous cell
      VariableContinuity : 1x1 matlab.internal.coder.tabular.Continuity
                RowTimes : 3x1 matlab.coder.type.DurationType
```

If your workflow requires the legacy representation of coder type objects, use the `getCoderType` function on the variable that has the new representation of your class or object. See “Legacy Representation of Coder Type Objects” on page 4-13.

Resize Object Properties by Using coder.resize

You can resize most objects by using `coder.resize`. You can resize objects, its properties and create arrays within the properties.

For a `timetable` coder object, you can resize the object properties:

```
t = timetable((1:5)',(11:15)','SampleRate',1);
tType = coder.typeof(t);
tType.UserData = coder.resize(tType.UserData,[10 1],[1 0])
```

This code resizes the `UserData` property to be a `:10x1 double` property. The first dimension is upper-bound at`10`.

```
tType =

   matlab.coder.type.RegularTimetableType
     5x2 timetable
                    Data : 1x2 homogeneous cell
             Description : 1x0 char
                UserData : :10x1 double
          DimensionNames : {'Time'}    {'Variables'}
           VariableNames : {'Var1'}    {'Var2'}
    VariableDescriptions : 1x2 homogeneous cell
           VariableUnits : 1x2 homogeneous cell
      VariableContinuity : 1x2 matlab.internal.coder.tabular.Continuity
               StartTime : 1x1 matlab.coder.type.DurationType
              SampleRate : 1x1 double
                TimeStep : 1x1 matlab.coder.type.DurationType
```

You can also resize the object by using `coder.resize`. See “Edit and Represent Coder Type Objects and Properties” on page 4-12.

See Also
`coder.Constant` | `coder.typeof` | `timetable`

More About

- “Code Generation for Timetables” on page 13-2
- “Timetable Limitations for Code Generation” on page 13-8

Timetable Limitations for Code Generation

If you create timetables, modify them, or use timetable functions in MATLAB code that you intend for code generation, then code generation has limitations described in the next sections. Limitations that apply to classes also apply to timetables. For more information on class limitations, see “MATLAB Classes Definition for Code Generation” on page 15-2.

Creating Timetables Limitations

If your MATLAB code creates timetables, then code generation has these limitations.

Inputs for Timetable Creation	Limitations
Any inputs	• The name of the first dimension of a timetable is `'Time'` unless you specify it by using the `'DimensionNames'` name-value argument. The name of the first dimension is also the name of the vector of row times, which you can refer to by using dot notation. • To create a regular timetable when the `'SampleRate'`, `'StartTime'`, or `'TimeStep'` name-value arguments are passed in by an entry point function, first use the `coder.Constant` function to make the values constant. If you do not make them constant, then the row times are considered to be irregular. • If you create a regular timetable, and you attempt to set irregular row times, then an error is produced. • If you create an irregular timetable, then it remains irregular even if you set its sample rate or time step. • Timetable variable names do not have to be valid MATLAB identifiers. The names must be composed of ASCII characters, which can include commas, dashes, and space characters.
Timetable created from input arrays	• You must specify variables names by using the `'VariableNames'` name-value argument when creating timetables from input arrays by using the `timetable` or `array2timetable` functions.

Inputs for Timetable Creation	Limitations
Timetable created with preallocated variables	• You do not have to specify the `'VariableNames'` argument when you preallocate a timetable by using the `timetable` function and the `'Size'` name-value argument. • You can specify only the following data types by using the `'VariableTypes'` name-value argument: • `'double'` • `'single'` • `'doublenan'` or `'doubleNaN'` • `'singlenan'` or `'singleNaN'` • `'int8'`, `'int16'`, `'int32'`, or `'int64'` • `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'` • `'logical'` • `'datetime'` • `'duration'` • `'cellstr'` • `'char'`

Modifying Timetables Limitations

If your MATLAB code modifies data in a timetable, its row times, or its properties, then code generation has these limitations.

Timetable Operation or Property	Limitations
`VariableNames`, `DimensionNames`, or `UserData` properties	• After you create a timetable, you cannot change the `VariableNames`, `DimensionNames`, or `UserData` properties. When you create a timetable, you can specify the `'VariableNames'`, `'DimensionNames'`, and `'RowTimes'` input arguments to set the properties having those names.

<table>
<tr><th>Timetable Operation or Property</th><th>Limitations</th></tr>
<tr><td>Timetable indices as input arguments to generated code</td><td>

To pass timetable indices that specify variables into generated code as input arguments, first use the <code>coder.Constant</code> function to make the indices into the second dimension of the timetable constant. If indices into the second dimension are not constant, then indexing into variables produces an error.
If a timetable has row times that are <code>duration</code> values, and you index into it by using either <code>duration</code> values or an object produced by the <code>timerange</code> or <code>withtol</code> functions, then the output is nonconstant with a variable number of rows.
If a regular timetable has row times that are <code>duration</code> values, and you index into it by using either <code>duration</code> values or an object produced by the <code>timerange</code> or <code>withtol</code> functions, then the output is considered to be irregular.

</td></tr>
<tr><td>Custom metadata</td><td>

You cannot add custom metadata to a timetable. The <code>addprop</code> and <code>rmprop</code> functions are not supported.

</td></tr>
<tr><td>Assignments that change size of timetable</td><td>

You cannot change the size of a timetable by assignments. For example, this call to add a new row produces an error.
<pre>
function TT = foo() %#codegen
    TT = timetable((1:3)',(1:3)','RowTimes',second
                   'VariableNames',{'Var1','Var2'}
    TT{4,:} = [5,5];
end
</pre>
Deleting a row or a variable by assignment also produces an error.
You cannot add a new row by using a new row time in an assignment. For example, this call to add a new row by using a new row time instead of a numeric index does not produce an error, but also does not add the new row.
<pre>
function TT = foo() %#codegen
    TT = timetable((1:3)',(1:3)','RowTimes',second
                   'VariableNames',{'Var1','Var2'}
    TT{seconds(15),:} = [5,5];
end
</pre>

</td></tr>
</table>

Timetable Operation or Property	Limitations
Vertical concatenation	• When you vertically concatenate timetables, they must have the same variable names in the same order. In MATLAB, the variable names must be the same but can be in different orders in the timetables.
Horizontal concatenation	• When you horizontally concatenate timetables, they must have the same row times in the same order. In MATLAB, the row times must be the same but can be in different orders in the timetables.
Timetable variables that are N-D cell arrays	• If two timetables have variables that are N-D cell arrays, then you cannot vertically concatenate the timetables. • You cannot use curly braces to extract data from multiple timetable variables that are N-D cell arrays because this operation is horizontal concatenation.

Using Timetable Functions Limitations

If your MATLAB code uses the functions listed in the table, then code generation has these limitations.

Function	Limitations
`convertvars`	• Function handles are not supported. • The second and third input arguments (`vars` and `dataType`) must be constant. • You cannot specify `dataType` as `'char'`.

`intersect`
`setdiff`
`setxor`
`union`
	• These functions support unsorted timetables in all cases. You do not have to specify the `'stable'` option.

`isregular`	• Use `coder.Constant` to make the input argument `timeComponent` constant. • The input argument `timeComponent` cannot be a calendar unit. If you specify it, then its value must be `'time'`.
`issortedrows`	• The input argument `vars` must be constant. • If any timetable variables have multiple columns, then those variables must have fixed widths.

Function	Limitations

`join`	• In general, input tables cannot have nonkey variables with the same names. However, you can specify duplicated variable names in the input tables by using:
	  • `'KeepOneCopy'`
	  • `'LeftVariables'` and `'RightVariables'`
	• The values of these name-value arguments must be constant:
	  • `'Keys'`
	  • `'LeftKeys'`
	  • `'RightKeys'`
	  • `'LeftVariables'`
	  • `'RightVariables'`
	  • `'KeepOneCopy'`
	• Nested timetables are not supported.

`movevars`	• The input argument `vars` cannot contain duplicate variable names.

`retime`
 	
`synchronize`
	• The row times of the output timetable are considered to be irregular, even when synchronized to row times that have a regular time step.
	• The `'makima'` interpolation method is not supported.
	• If the `VariableContinuity` properties of the input timetables are not constant, then this function ignores them.
	• The `'weekly'`, `'monthly'`, and `'quarterly'` time steps are not supported.
	  • If the input timetables have row times that are `datetime` values, then the `'daily'` and `'yearly'` time steps also are not supported.

`sortrows`	• The input argument `vars` must be constant. • If `tblA` has a variable that is a cell array of character vectors with multiple columns, then you cannot sort the timetable using the values in that variable.
`splitvars`	• The value of the `'NewVariableNames'` name-value argument must be constant. • The variables that are split cannot have a variable number of columns.

Function	Limitations
`stack`	• The second input argument, `vars`, must be constant. • The values of the `'ConstantVariables'`, `'NewDataVariableName'`, and `'IndexVariableName'` name-value arguments must be constant.
`timerange`	• The input argument `unitOfTime` is not supported.
`unstack`	• The `'NewDataVariableNames'` name-value argument must be specified. Its value must be constant. • The `vars` and `ivars` input arguments (data variables and indicator variables) must be constant. • If you specify grouping variables and constant variables, then they must be constant. • If you specify an aggregation function, then it must be constant. • If the input is a timetable with regular row times and you specify grouping variables that do not include the row times, then the output timetable might have irregular row times. Even though the intervals between output row times might look the same, the output timetable considers the vector of row times to be irregular. • If a variable of the input timetable is a cell array of character vectors, then `unstack` fills empty cells in the corresponding output variable with 1-by-0 character arrays in the generated code. In MATLAB, `unstack` fills such gaps with 0-by-0 character arrays. • The `unstack` function does not support code generation when the input timetable has a variable that is a heterogeneous cell array that cannot be converted to a homogeneous cell array. • If the input has a variable that is a homogeneous cell array, or that can be converted to one, then the `'AggregationFunction'` name-value argument must be specified. The default value of `'AggregationFunction'` is `'unique'`. But the `unique` function does not support cell arrays.

Function	Limitations
`varfun`	• The function handle input, `func`, must be constant. • While function handles can be inputs to `varfun` itself, they cannot be inputs to your entry point functions. Specify `func` within the code meant for code generation. For more information, see “Function Handle Limitations for Code Generation” on page 17-2. • The values for all name-value arguments must be constant. • The `'ErrorHandler'` name-value argument is not supported for code generation. • Variable-size input arguments are not supported. • If you specify `'GroupingVariables'`, then the output is always an irregular timetable. • Grouping variables cannot have duplicate values in generated code. • You cannot specify the value of `'OutputFormat'` as `'cell'` if you specify the `'GroupingVariables'` name-value arguments and the function returns a different data type for each variable specified by `'InputVariables'`. • If you specify groups and the number of groups is not known at compile-time, and that number turns out to be zero, then empty double variables in the output might have sizes of 1-by-0 in generated code. In MATLAB, such variables have sizes of 0-by-0.

See Also

`array2timetable` | `table2timetable` | `timetable`

More About

- “Code Generation for Timetables” on page 13-2
- “Define Timetable Inputs” on page 13-5

Code Generation for Enumerated Data

- “Code Generation for Enumerations” on page 14-2
- “Customize Enumerated Types in Generated Code” on page 14-7

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:

- Calls to methods of enumeration classes are not supported.
- Passing strings or character vectors to constructors of enumerations is not supported.
- The enumeration class must derive from one of these base types: `int8`, `uint8`, `int16`, `uint16`, or `int32`. See “Define Enumerations for Code Generation” on page 14-2.
- You can use only a limited set of operations on enumerations. See “Allowed Operations on Enumerations” on page 14-4.
- Use enumerations with functions that support enumerated types for code generation. See “MATLAB Toolbox Functions That Support Enumerations” on page 14-5.

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.

Representation of Enumerated Type in Generated Code

The base type determines the representation of the enumerated type in generated C/C++ code.

If the base type is the native integer type for the target platform (for example, `int32`), the code generator produces a C/C++ enumerated type. Consider this MATLAB enumerated type definition:

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

This enumerated type definition results in the following C/C++ code:

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

typedef enum LEDcolor LEDcolor;
```

For built-in integer base types that are different from the native integer type for the target platform:

- If you generate C code, the code generator produces a `typedef` statement for the enumerated type and `#define` statements for the enumerated values. Consider this MATLAB enumerated type definition:

  ```
  classdef LEDcolor < int16
      enumeration
          GREEN(1),
          RED(2)
      end

  end
  ```

  The enumerated type definition produces this C code:

  ```
  typedef short LEDcolor;
  #define GREEN ((LEDcolor)1)
  #define RED ((LEDcolor)2)
  ```

- If you generate C++ code, the enumeration members are converted to constants. These constants belong to the namespace that contains the enumeration type definition in the generated C++ code.

  For example, suppose that you place this MATLAB enumerated type definition inside the package `pkg`:

  ```
  classdef LEDcolor < int16
      enumeration
          GREEN(1),
          RED(2)
      end

  end
  ```

  The default behavior of the code generator is to convert MATLAB packages to C++ namespaces. The generated C++ code is placed inside the namespace `pkg`:

  ```
  namespace pkg {
  typedef short LEDcolor;

  // enum pkg_LEDcolor
  const LEDcolor GREEN{1};
  const LEDcolor RED{2};

  }
  ```

The C/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 27-24 and “Mapping MATLAB Types to Types in Generated Code” on page 33-15.

Allowed Operations on Enumerations

For code generation, you are restricted to the operations on enumerations listed in this table.

Operation	Example	Notes
assignment operator: =		—
relational operators: `<  >  <=  >=  ==  ~=`	`xon == xoff`	Code generation does not support using == or ~= to test equality between an enumeration member and a string array, a character array, or a cell array of character arrays.
cast operation	`double(LEDcolor.RED)`	—

conversion to character array or string	`y = char(LEDcolor.RED);`
	`y1 = cast(LEDcolor.RED,'char');`
	`y2 = string(LEDcolor.RED);`
		• You can convert only compile-time scalar valued enumerations. For example, this code runs in MATLAB, but produces an error in code generation:
		`y2 = string(repmat(LEDcolor.RED,1,2));`
		• The code generator preserves enumeration names when the conversion inputs are constants. For example, consider this enumerated type definition:
		`classdef AnEnum < int32`
		`    enumeration`
		`        zero(0),`
		`        two(2),`
		`        otherTwo(2)`
		`    end`
		`end`
		Generated code produces `"two"` for
		`y = string(AnEnum.two)`
		and `"otherTwo"` for
		`y = string(AnEnum.two)`

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`
		—

MATLAB Toolbox Functions That Support Enumerations

For code generation, you can use enumerations with these MATLAB toolbox functions:

- `cast`
- `cat`
- `char`
- `circshift`
- `enumeration`
- `fliplr`
- `flipud`
- `histc`
- `intersect`
- `ipermute`
- `isequal`
- `isequaln`
- `isfinite`
- `isinf`
- `ismember`
- `isnan`
- `issorted`
- `length`
- `permute`
- `repmat`
- `reshape`
- `rot90`
- `setdiff`
- `setxor`
- `shiftdim`
- `sort`
- `sortrows`
- `squeeze`
- `string`
- `union`
- `unique`

See Also

More About

- “Generate Code for an LED Control Function That Uses Enumerated Types” on page 27-131

- "Customize Enumerated Types in Generated Code" on page 14-7

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 14-7.
`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 14-8
`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 `addClassNameToEnumNames` method to `true`. See "Include Class Name Prefix in Generated Enumerated Type Value Names" on page 14-8.

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;
```

See Also

More About

- Modifying Superclass Methods and Properties
- “Code Generation for Enumerations” on page 14-2

Code Generation for MATLAB Classes

- “MATLAB Classes Definition for Code Generation” on page 15-2
- “Classes That Support Code Generation” on page 15-7
- “Generate Code for MATLAB Value Classes” on page 15-8
- “Generate Code for MATLAB Handle Classes and System Objects” on page 15-12
- “Code Generation for Handle Class Destructors” on page 15-15
- “Class Does Not Have Property” on page 15-18
- “Passing By Reference Not Supported for Some Properties” on page 15-20
- “Handle Object Limitations for Code Generation” on page 15-21
- “System Objects in MATLAB Code Generation” on page 15-24
- “Specify Objects as Inputs at the Command Line” on page 15-27
- “Specify Objects as Inputs in the MATLAB Coder App” on page 15-30

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
Restricted set of language features.	"Language Limitations" on page 15-2
Restricted set of code generation features.	"Code Generation Features Not Compatible with Classes" on page 15-3
Definition of class properties.	"Defining Class Properties for Code Generation" on page 15-3
Use of handle classes.	"Generate Code for MATLAB Handle Classes and System Objects" on page 15-12 "Code Generation for Handle Class Destructors" on page 15-15 "Handle Object Limitations for Code Generation" on page 15-21
Global variables containing MATLAB handle 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 15-6

Language Limitations

Although code generation support is provided for common features of classes such as properties and methods, there are a number of advanced features which are not supported, such as:

- Events
- Listeners
- Arrays of objects
- Recursive data structures
  - Linked lists
  - Trees
  - Graphs
- Nested functions in constructors
- 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`
  - `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 global variables that are handle classes.
- 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 `coder.ceval`. You can pass class properties to `coder.ceval`.
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties” on page 15-20.
- If an object has duplicate property names and the code generator tries to constant-fold the object, code generation can fail. The code generator constant-folds an object when it is used with `coder.Constant` or `coder.const`, or when it is an input to or output from a constant-folded extrinsic function.

  Duplicate property names occur in an object of a subclass in these situations:

  - The subclass has a property with the same name as a property of the superclass.
  - The subclass derives from multiple superclasses that use the same name for a property.

  For information about when MATLAB allows duplicate property names, see “Subclassing Multiple Classes”.

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:

- MEX functions report errors that result from property validation. Standalone C/C++ code reports these errors only if you enable run-time error reporting. See “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20. Before you generate standalone C/C++ code, it is a best practice to test property validation by running a MEX function over the full range of input values.
- 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)`.
- Variable-size properties:

  - Code generation supports upper-bounded and unbounded variable-size properties for both value and handle classes.
  - To generate unbounded variable-size class properties, enable dynamic memory allocation.

- To make a variable-size class property, make two sequential assignments of a class property, one to a scalar and the next to an array.

  ```
  classdef varSizeProp1 < handle
      properties
          prop
          varProp
      end
  end

  function extFunc(n)
      obj = varSizeProp1;
      % Assign a scalar value to the property.
      obj.prop = 1;
      obj.varProp = 1;
      % Assign an array to the same property to make it variable-sized.
      obj.prop = 1:98;
      obj.varProp = 1:n;
  end
  ```

  In the preceding code, the first assignment to `prop` and `varProp` is scalar, and their second assignment is to an array with the same base type. The size of `prop` has an upper bound of `98`, making it an upper-bounded, variable-size property.

  If `n` is unknown at compile time, `obj.varProp` is an unbounded variable-size property. If it is known, it is an upper-bounded, variable-size class property.

- If the class property is initialized with a variable-size array, the property is variable-size.

  ```
  classdef varSizeProp2
      properties
          prop
      end
      methods
          function obj = varSizeProp2(inVar)
              % Assign incoming value to local variable
              locVar = inVar;

              % Declare the local variable to be a variable-sized column
              % vector with no size limit
              coder.varsize('locVar',[inf 1],[1 0]);

              % Assign value
              obj.prop = locVar;
           end
      end
  end
  ```

  In the preceding code, `inVar` is passed to the class constructor and stored in `locVar`. `locVar` is modified to be variable-size by `coder.varsize` and assigned to the class property `obj.prop`, which makes the property variable-size.

  - If the input to the function call `varSizeProp2` is variable-size, `coder.varsize` is not required.

    ```
    function z = constructCall(n)
        z = varSizeProp2(1:n);
    end
    ```

  - If the value of `n` is unknown at compile-time and has no specified bounds, `z.prop` is an unbounded variable-size class property.
  - If the value of `n` is unknown at compile-time and has specified bounds, `z.prop` is an upper-bounded variable-size class property.

- 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;
```

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 15-8
Handle classes including user-defined System objects	“Generate Code for MATLAB Handle Classes and System Objects” on page 15-12

For more information, see:

- “Role of Classes in MATLAB”
- “MATLAB Classes Definition for Code Generation” on page 15-2

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.

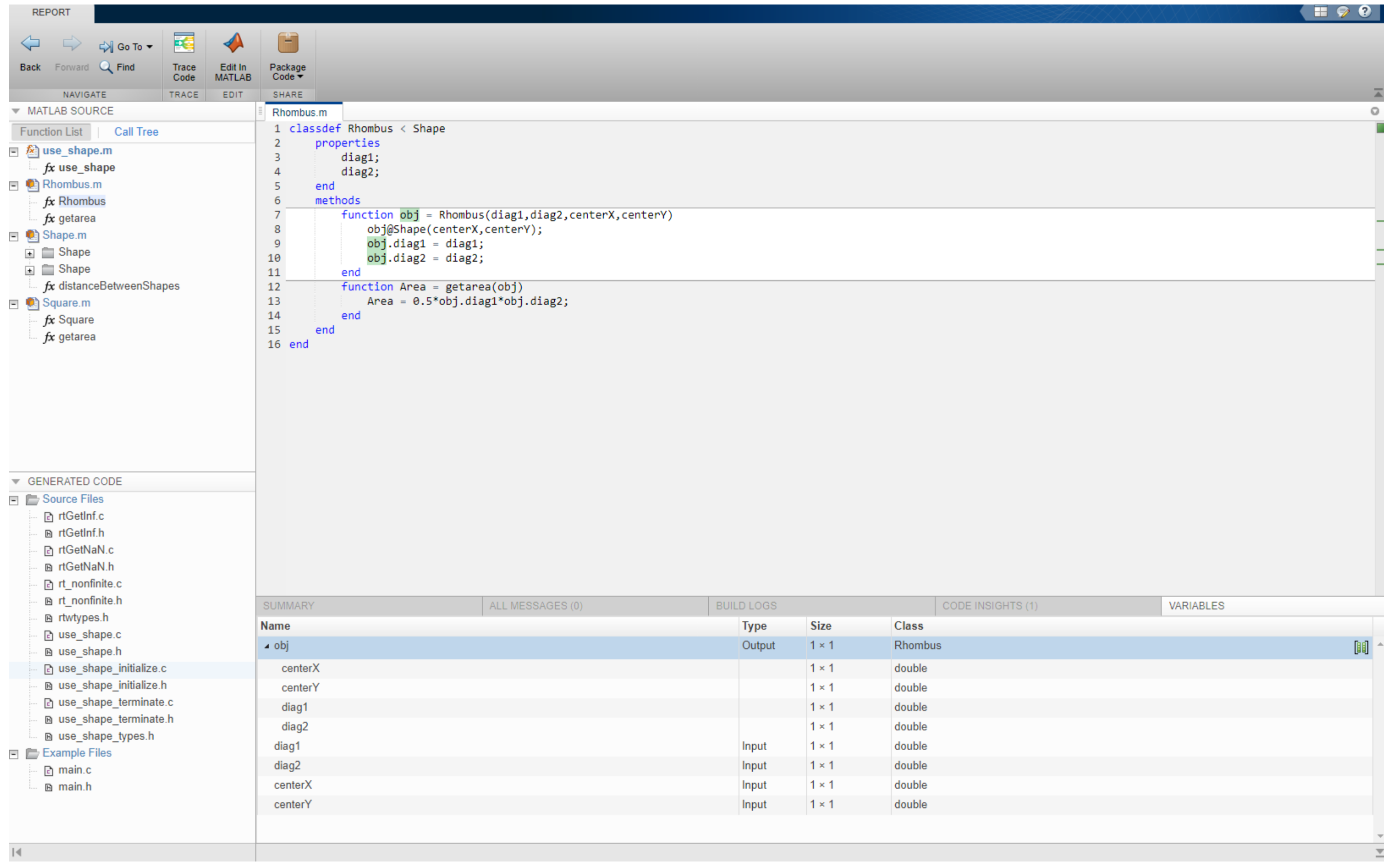
```
codegen -config:lib -report use_shape
```

`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.

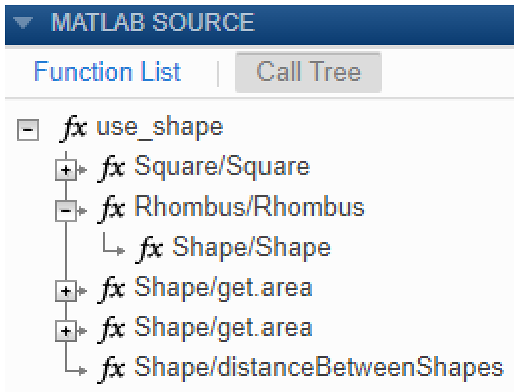
7 To see the `Rhombus` class definition, on the **MATLAB Source** pane, under `Rhombus.m`, click `Rhombus`. The `Rhombus` class constructor is highlighted.

8 Click the **Variables** tab. You see that the variable `obj` is an object of the `Rhombus` class. To see its properties, expand `obj`.



9 In the **MATLAB Source** pane, click **Call Tree**.

The **Call Tree** view shows that `use_shape` calls the `Rhombus` constructor and that the `Rhombus` constructor calls the `Shape` constructor.



10 In the code pane, in the `Rhombus` class constructor, move your pointer to this line:

```
obj@Shape(centerX,centerY)
```

The `Rhombus` class constructor calls the `Shape` method of the base `Shape` class. To view the `Shape` class definition, in `obj@Shape`, double-click `Shape`.

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
```

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.

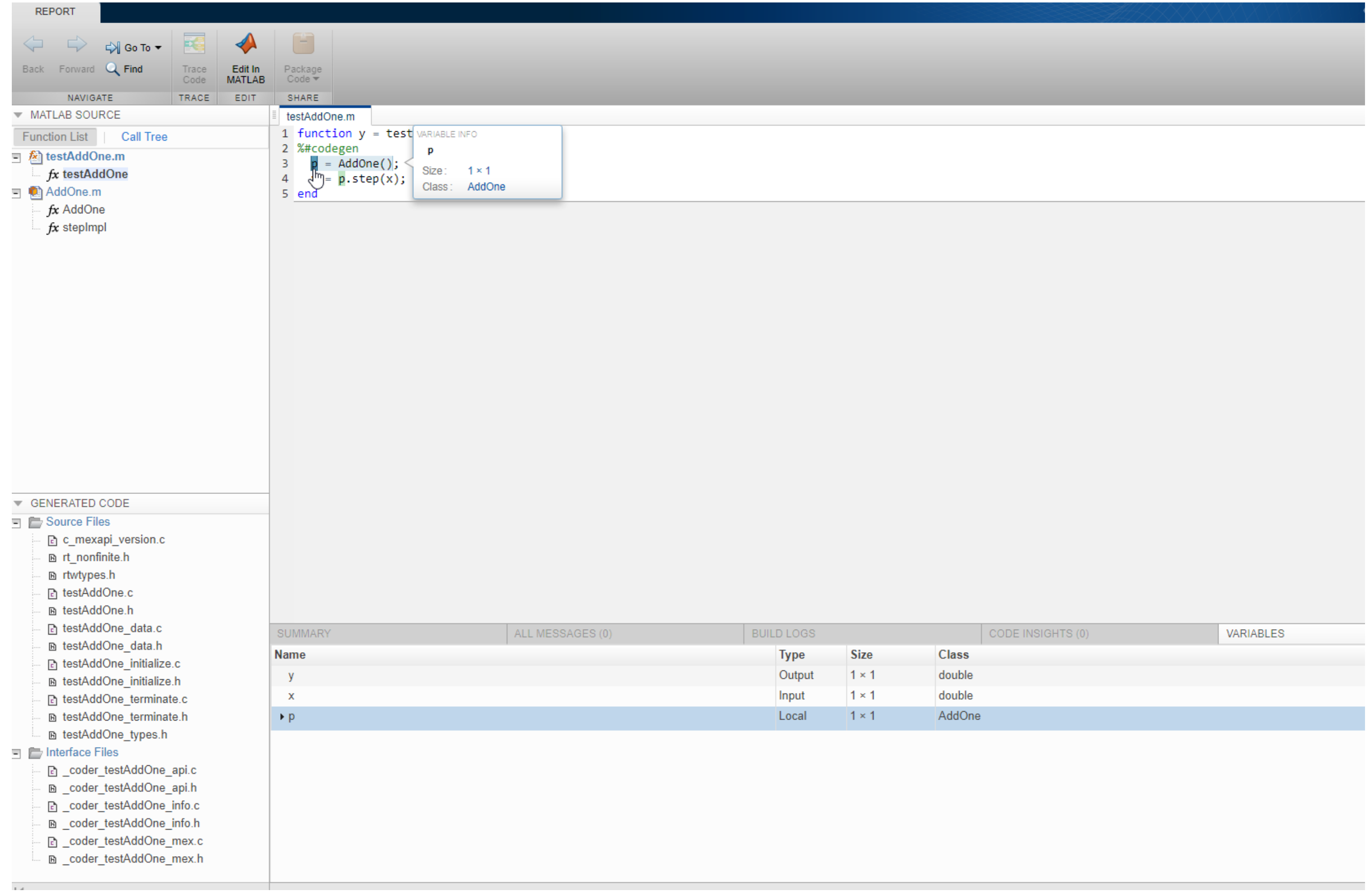
```
function y = testAddOne(x)
%#codegen
  p = AddOne();
  y = p.step(x);
end
```

3. Generate a MEX function for this code.

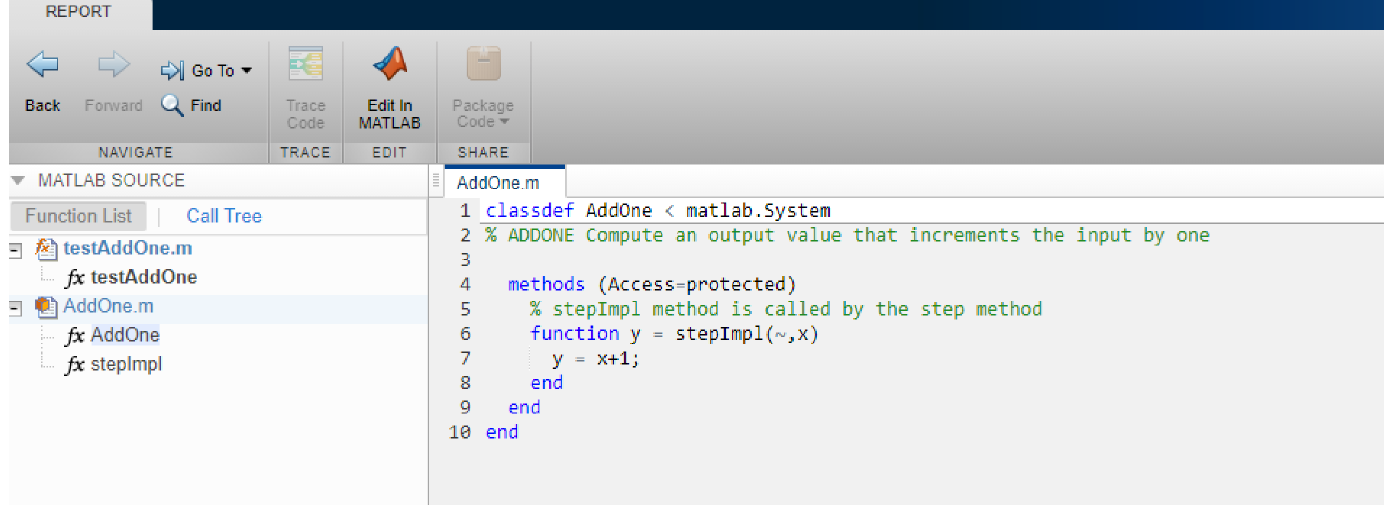
```
codegen -report testAddOne -args {0}
```

   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 **MATLAB Source** pane, click `testAddOne`. To see information about the variables in `testAddOne`, click the **Variables** tab.



6 To view the class definition for `addOne`, in the **MATLAB Source** pane, click `AddOne`.



See Also

More About

- "Code Generation for Handle Class Destructors" on page 15-15

Code Generation for Handle Class Destructors

You can generate code for MATLAB code that uses `delete` methods (destructors) for handle classes. To perform clean-up operations, such as closing a previously opened file before an object is destroyed, use a `delete` method. The generated code calls the `delete` method at the end of an object's lifetime, even if execution is interrupted by a run-time error. When System objects are destroyed, `delete` calls the `release` method, which in turn calls the user-defined `releaseImpl`. For more information on when to define a `delete` method in a MATLAB code, see “Handle Class Destructor”.

Guidelines and Restrictions

When you write the MATLAB code, adhere to these guidelines and restrictions:

- Code generation does not support recursive calls of the `delete` method. Do not create an object of a certain class inside the `delete` method for the same class. This usage might cause a recursive call of `delete` and result in an error message.
- The generated code always calls the `delete` method, when an object goes out of scope. Code generation does not support explicit calls of the `delete` method.
- Initialize all properties of `MyClass` that the `delete` method of `MyClass` uses either in the constructor or as the default property value. If `delete` tries to access a property that has not been initialized in one of these two ways, the code generator produces an error message.
- Suppose a property `prop1` of `MyClass1` is itself an object (an instance of another class `MyClass2`). Initialize all properties of `MyClass2` that the `delete` method of `MyClass1` uses. Perform this initialization either in the constructor of `MyClass2` or as the default property value. If `delete` tries to access a property of `MyClass2` that has not been initialized in one of these two ways, the code generator produces an error message. For example, define the two classes `MyClass1` and `MyClass2`:

  ```
  classdef MyClass1 < handle
      properties
          prop1
      end
      methods
          function h = MyClass1(index)
              h.prop1 = index;
          end
          function delete(h)
              fprintf('h.prop1.prop2 is: %1.0f\n',h.prop1.prop2);
          end
      end
  end

  classdef MyClass2 < handle
      properties
          prop2
      end
  end
  ```

  Suppose you try to generate code for this function:

  ```
  function MyFunction
  obj2 = MyClass2;
  ```

```
obj1 = MyClass1(obj2); % Assign obj1.prop1 to the input (obj2)
end
```

The code generator produces an error message because you have not initialized the property `obj2.prop2` that the `delete` method displays.

Behavioral Differences of Objects in Generated Code and in MATLAB

The behavior of objects in the generated code can be different from their behavior in MATLAB in these situations:

- The order of destruction of several independent objects might be different in MATLAB than in the generated code.
- The lifetime of objects in the generated code can be different from their lifetime in MATLAB. MATLAB calls the `delete` method when an object can no longer be reached from any live variable. The generated code calls the `delete` method when an object goes out of scope. In some situations, this difference causes `delete` to be called later on in the generated code than in MATLAB. For example, define the class:

  ```
  classdef MyClass < handle
      methods
          function delete(h)
              global g
              % Destructor displays current value of global variable g
              fprintf('The global variable is: %1.0f\n',g);
          end
      end
  end
  ```

  Run the function:

  ```
  function MyFunction
  global g
  g = 1;
  obj = MyClass;
  obj = MyClass;
  % MATLAB destroys the first object here
  g = 2;
  % MATLAB destroys the second object here
  % Generated code destroys both objects here
  end
  ```

  The first object can no longer be reached from any live variable after the second instance of `obj = MyClass` in `MyFunction`. MATLAB calls the `delete` method for the first object after the second instance of `obj = MyClass` in `MyFunction` and for the second object at the end of the function. The output is:

  ```
  The global variable is: 1
  The global variable is: 2
  ```

  In the generated code, both `delete` method calls happen at the end of the function when the two objects go out of scope. Running `MyFunction_mex` results in a different output:

  ```
  The global variable is: 2
  The global variable is: 2
  ```

- In MATLAB, `persistent` objects are automatically destroyed when they cannot be reached from any live variable. In the generated code, you have to call the `terminate` function explicitly to destroy the `persistent` objects.
- The generated code does not destroy partially constructed objects. If a handle object is not fully constructed at run time, the generated code produces an error message but does not call the `delete` method for that object. For a System object, if there is a run-time error in `setupImpl`, the generated code does not call `releaseImpl` for that object.

  MATLAB does call the `delete` method to destroy a partially constructed object.

See Also

More About

- “Generate Code for MATLAB Handle Classes and System Objects” on page 15-12
- “System Objects in MATLAB Code Generation” on page 15-24

Class Does Not Have Property

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;
```

See Also

More About

- “MATLAB Classes Definition for Code Generation” on page 15-2

Passing By Reference Not Supported for Some Properties

The code generator does not support passing a property by reference to an external function for these types of properties:

- A property with a get method or a set method.
- A property that uses validation functions.
- A System object property with an attribute, such as `Logical` or `PositiveInteger`, that constrains or modifies the property value.

Instead of passing a property by reference, save the property value in a temporary variable. Then, pass the temporary variable by reference to the external function. After the external function call, assign the temporary variable to the property. For example:

```
tmp = myObj.prop;
coder.ceval('myFcn', coder.ref(tmp));
myObj.prop = tmp;
```

The assignment after the `coder.ceval` call validates or modifies the property value according to the property access methods, validation functions, or attributes.

See Also

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

More About

- “Call C/C++ Code from MATLAB Code” on page 33-2
- “MATLAB Classes Definition for Code Generation” on page 15-2

Handle Object Limitations for Code Generation

The code generator statically determines the lifetime of a handle object. When you use handle objects, this static analysis has certain restrictions.

With static analysis the generated code can reuse memory rather than rely on a dynamic memory management scheme, such as reference counting or garbage collection. The code generator can avoid dynamic memory allocation and run-time automatic memory management. These generated code characteristics are important for some safety-critical and real-time applications.

For limitations, see:

- “A Variable Outside a Loop Cannot Refer to a Handle Object Allocated Inside a Loop” on page 15-21
- “A Handle Object That a Persistent Variable Refers To Must Be a Singleton Object” on page 15-22

The code generator analyzes whether all variables are defined prior to use. Undefined variables or data types cause an error during code generation. In certain circumstances, the code generator cannot determine if references to handle objects are defined. See “References to Handle Objects Can Appear Undefined” on page 15-23.

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

Consider the handle class `mycls` and the function `usehandle1`.

```
classdef mycls < handle
   properties
       prop
   end

   methods
       function obj = mycls(x)
       obj.prop = x;
       end
   end
end

function y = usehandle1
p = mycls(0); % Instance of mycls with prop value 10 created

for i = 1:10
    p = mycls(i); % Handle object allocated inside loop
end

y = p.prop; % Handle object referenced outside loop
end
```

If you try to generate code for the `usehandle1` function, the code generator produces an error. The error occurs because:

- A handle object is allocated inside the `for` loop. The variable `p.prop` refers to this handle object.
- Outside the loop, the variable `x` refers to the property `prop` handle object.

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
```

References to Handle Objects Can Appear Undefined

Consider the function `refHandle` that copies a handle object property to another object. The function uses a simple handle class and value class. In MATLAB, the function runs without error.

```
function [out1, out2, out3] = refHandle()
  x = myHandleClass;
  y = x;
  v = myValueClass();
  v.prop = x;
  x.prop = 42;
  out1 = x.prop;
  out2 = y.prop;
  out3 = v.prop.prop;
end

classdef myHandleClass < handle
    properties
        prop
    end
end

classdef myValueClass
    properties
        prop
    end
end
```

During code generation, an error occurs:

```
Property 'v.prop.prop' is undefined on some execution paths.
```

Three variables reference the same memory location: `x`, `y`, and `v.prop`. The code generator determines that `x.prop` and `y.prop` share the same value. The code generator cannot determine that the handle object property `v.prop.prop` shares its definition with `x.prop` and `y.prop`. To avoid the error, define `v.prop.prop` directly.

System Objects in MATLAB Code Generation

In this section...

"Usage Rules and Limitations for System Objects for Generating Code" on page 15-24

"System Objects in codegen" on page 15-26

"System Objects in the MATLAB Function Block" on page 15-26

"System Objects in the MATLAB System Block" on page 15-26

"System Objects and MATLAB Compiler Software" on page 15-26

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.
- Initialize all System objects properties that `releaseImpl` uses before the end of `setupImpl`.
- 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.
- The complexity 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.
- For fixed-point inputs, if a tunable property has dependent 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. See “Generate Code for Global Data” on page 27-88.

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 "Get 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 "Implementing MATLAB Functions Using Blocks" (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.

See Also

More About

- "Generate Code That Uses Row-Major Array Layout" on page 37-4

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
```

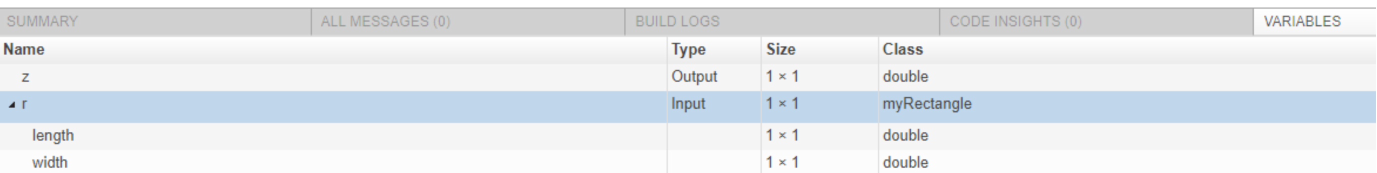
3 Create an object of the class.

```
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`.



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.

- An object cannot be a global variable.
- If an object has duplicate property names, you cannot use it with `coder.Constant`. Duplicate property names occur in an object of a subclass in these situations:
  - The subclass has a property with the same name as a property of the superclass.
  - The subclass derives from multiple superclasses that use the same name for a property.

  For information about when MATLAB allows duplicate property names, see "Subclassing Multiple Classes".

See Also

`coder.ClassType`

More About

- "Automatically Define Input Types by Using the App" on page 24-4
- "Define Input Parameter by Example by Using the App" on page 24-6
- "MATLAB Classes Definition for Code Generation" on page 15-2
- "Specify Objects as Inputs in the MATLAB Coder App" on page 15-30

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 on page 15-30.
- Provide an Example Object on page 15-30.

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(rect_area);
```

- 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.

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.
- An object cannot be a global variable.
- If an object has duplicate property names, you cannot use it with `coder.Constant`. Duplicate property names occur in an object of a subclass in these situations:

  - The subclass has a property with the same name as a property of the superclass.
  - The subclass derives from multiple superclasses that use the same name for a property.

  For information about when MATLAB allows duplicate property names, see “Subclassing Multiple Classes”.

See Also

`coder.ClassType`

More About

- “Automatically Define Input Types by Using the App” on page 24-4
- “Define Input Parameter by Example by Using the App” on page 24-6
- “Specify Objects as Inputs at the Command Line” on page 15-27
- “MATLAB Classes Definition for Code Generation” on page 15-2

16

Generating C++ Classes

Generate C++ Classes for MATLAB Classes

When you generate C++ code, the default behavior of the code generator produces C++ classes for the classes in your MATLAB code. These include all MATLAB classes such as value classes, handle classes, and system objects.

You can change the default behavior of the code generator to produce structures for MATLAB classes. To change the default behavior, do one of the following:

- In a code configuration object, set `TargetLang` to `'C++'` and `CppPreserveClasses` to `false`.
- In the MATLAB Coder app, in the **Generate** step, set **Language** to **C++**. In the project build settings, on the **Code Appearance** tab, clear the **Generate C++ classes from MATLAB classes** check box.

These examples illustrate certain rules that the code generator follows when mapping MATLAB classes to C++ classes.

Example: Generate Code for a Handle Class That Has Private and Public Members

Define a MATLAB handle class `MyClass`:

```
classdef MyClass < handle
    properties
        publicProp = 1;
    end
    properties(Access = private)
        privateProp
    end
    methods
        function obj = MyClass(value)
            obj.privateProp = value;
        end
        function publicMethod(obj,value)
            obj.privateMethod(value);
        end
        function res = calculateSomeValue(obj)
            res = obj.publicProp*obj.privateProp;
        end
    end
    methods (Access = private)
        function privateMethod(obj,value)
            obj.publicProp = obj.publicProp + value;
            obj.privateProp = obj.privateProp + obj.doubleThisValue(value);
        end
    end
    methods(Static)
        function res = doubleThisValue(val)
            res = 2 * val;
        end
    end
end
```

Define a MATLAB function `foo` that uses `MyClass`:

```
function out = foo(x,y)
obj = MyClass(x);
obj.publicMethod(y);
out = obj.calculateSomeValue;
end
```

Generate a static C++ library for `foo`. Specify the input argument to be a double scalar. Set the code generation configuration property `InlineBetweenUserFunctions` to `'Readability'`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.InlineBetweenUserFunctions = 'Readability';
codegen -config cfg foo -args {0,0} -report
```

```
Code generation successful: View report
```

Open the code generation report and inspect the generated code. The file `MyClass.h` contains the definition of the generated C++ class `MyClass`:

```
class MyClass
{
 public:
  MyClass *init(double value);
  void publicMethod(double value);
  static double doubleThisValue(double val);
  double calculateSomeValue() const;
  double publicProp;
 private:
  double privateProp;
};
```

This is the code generated for the function `foo`:

```
double foo(double x, double y)
{
  MyClass obj;
  obj.init(x);
  obj.publicMethod(y);
  return obj.calculateSomeValue();
}
```

This table lists some of the rules that the code generator follows when generating C++ classes and the corresponding snippets from the code generated for `MyClass`.

Rule	Code Snippet

The class constructor in MATLAB is mapped onto an `init` method. When an instance of the class is created, the generated code explicitly calls the `init` method.	The file `MyClass.cpp` contains the definition of `init`.
	`MyClass *MyClass::init(double value)`
	`{`
	`  MyClass *obj;`
	`  obj = this;`
	`  obj->publicProp = 1.0;`
	`  obj->privateProp = value;`
	`  return obj;`
	`}`

<table>
<tr><th>Rule</th><th>Code Snippet</th></tr>
<tr>
<td>
In most cases, if a class member is set as private in MATLAB, it is also set as private in the generated C++ code.
In certain situations, inlining a public method in the generated C++ code changes a private property in the your MATLAB code to a public property in the generated code and breaks data encapsulation. For example, suppose that a public method <code>myMethod</code> that uses a private property <code>prop</code> of the object is called by an entry-point function. If <code>myMethod</code> is inlined in the generated code, the property <code>prop</code> must be visible from outside the object and changed to a public property.
To limit this occurrence, the code generator uses a special inlinig rule for public methods in this situation:

If the code configuration property <code>InlineBetweenUserFunctions</code> or the equivalent code generation setting Inline between user functions in the MATLAB Coder app is set to <code>'Readability'</code>, the code generator does not inline the public method calls that appear outside the class definition.

In these situations, the same inlining rules apply to both ordinary functions and public methods:

The body of the function or the method contains an explicit <code>coder.inline('always')</code> or <code>coder.inline('never')</code> directive. This directive gets the highest precedence.
You set the code configuration property <code>InlineBetweenUserFunctions</code> or the equivalent code generation setting Inline between user functions in the MATLAB Coder app to <code>'Never'</code>, <code>'Speed'</code>, or <code>'Always'</code>.
A call to a method appears inside another method of the same class.

See “Control Inlining to Fine-Tune Performance and Readability of Generated Code” on page 34-9.
</td>
<td>
The definition of the generated C++ class <code>MyClass</code> is:
<pre>
class MyClass
{
 public:
  MyClass *init(double value);
  void publicMethod(double value);
  static double doubleThisValue(double val);
  double calculateSomeValue() const;
  double publicProp;
 private:
  double privateProp;
};
</pre>
The visibility of all data and member functions is preserved between MATLAB and the generated code.
The private method <code>privateMethod</code> does not appear in this definition. <code>privateMethod</code> is inlined in the definition of <code>publicMethod</code> (see in the file <code>MyClass.cpp</code>) :
<pre>
void MyClass::publicMethod(double value)
{
  this->publicProp += value;
  this->privateProp += MyClass::doubleThisValue((valu
}
</pre>
</td>
</tr>
</table>

Rule	Code Snippet
Static methods in MATLAB are mapped onto static C++ methods.	The generated code for the static method `doubleThisValue` has this signature: `static double doubleThisValue(double val);`
Methods that do not mutate the object are marked with the `const` qualifier in the generated code.	The public method `calculateSomeValue` does not mutate the object. The generated method has this signature: `double calculateSomeValue() const;`

Additional Usage Notes and Limitations

These are some additional usage notes and limitations for generating C++ classes from MATLAB classes:

- The class prototype for `MyClass` is contained in the header file `MyClass.h`. The implementations of the methods of the class are contained in the file `MyClass.cpp`.
- In the generated code, class hierarchies are flattened. For example, suppose that in your MATLAB code, class `B` inherits from class `A`. In the generated C++ code, classes `B` and `A` have no inheritance relationship between them. In the generated code, all properties and methods of class `A` are reproduced in the definition of class `B`.
- When a MATLAB class uses different types for its properties, the code generator produces a separate C++ class for each type usage.
- If a MATLAB class member has different `GetAccess` and `SetAccess` attributes, the corresponding member of the generated class has the more permissive of the two attributes. For example, if a property `prop` has the attributes `(GetAccess = public, SetAccess = private)`, `prop` is defined to be a public property in the generated code.
- While attempting to generate standalone code that contains C++ classes for MATLAB classes, you might get a warning message if both of these conditions are true:

  - You choose to generate reentrant code by enabling the `MultiInstanceCode` parameter in a code configuration object or by enabling the **Generate re-entrant code** parameter in the MATLAB Coder app.
  - The destructor of a class in your MATLAB code has a persistent variable or calls another function that declares and uses a persistent variable.

  In such situations, to generate code that contains C++ classes for MATLAB classes, disable the `MultiInstanceCode` or the **Generate re-entrant code** parameter.

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

- "Create a Simple Class"
- "MATLAB Classes Definition for Code Generation" on page 15-2
- "System Objects in MATLAB Code Generation" on page 15-24

- “Control Inlining to Fine-Tune Performance and Readability of Generated Code” on page 34-9

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.

See Also

More About

- “Using the coder.extrinsic Construct” on page 20-9

Code Generation for Deep Learning Arrays

- "Code Generation for dlarray" on page 18-2
- "dlarray Limitations for Code Generation" on page 18-8

Code Generation for dlarray

In this section...

"Define dlarray for Code Generation" on page 18-2
"dlarray Object Functions with Code Generation Support" on page 18-3
"Deep Learning Toolbox Functions with dlarray Code Generation Support" on page 18-4
"MATLAB Functions with dlarray Code Generation Support" on page 18-4

A deep learning array stores data with optional data format labels for custom training loops, and enables functions to compute and use derivatives through automatic differentiation. To learn more about custom training loops, automatic differentiation, and deep learning arrays, see "Deep Learning Custom Training Loops" (Deep Learning Toolbox).

Code generation supports both formatted and unformatted deep learning arrays. `dlarray` objects containing `gpuArrays` are also supported for code generation. When you use deep learning arrays with CPU and GPU code generation, adhere to these restrictions:

Define dlarray for Code Generation

For code generation, use the `dlarray` function to create deep learning arrays. For example, suppose you have a pretrained `dlnetwork` network object in the `mynet.mat` MAT-file. To predict the responses for this network, create an entry-point function in MATLAB.

There are two possibilities:

Note For code generation, the `dlarray` input to the `predict` method of the `dlnetwork` object must be `single` data type.

Design 1 (Not recommended)

In this design example, the input and output to the entry-point function, `foo` are of `dlarray` types. This type of entry-point function is not recommended for code generation because in MATLAB, `dlarray` enforces the order of labels `'SCBTU'`. This behavior is replicated for MEX code generation. However, for standalone code generation such as static, dynamic libraries, or executables, the data format follows the specification of the `fmt` argument of the `dlarray` object. As a result, if the input or output of an entry-point function is a `dlarray` object and its order of labels is not `'SCBTU'`, then the data layout will be different between the MATLAB environment and standalone code.

```
function dlOut = foo(dlIn)

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dlOut = predict(dlnet, dlIn);

end
```

Design 2 (Recommended)

In this design example, the input and output to foo are of primitive datatypes and the dlarray object is created within the function. The extractdata method of the dlarray object returns the data in the dlarray dlA as the output of foo. The output a has the same data type as the underlying data type in dlA.

When compared to Design 1, this entry-point design has the following advantages:

- Easier integration with standalone code generation workflows such as static, dynamic libraries, or executables.
- The data format of the output from the extractdata function has the same order ('SCBTU') in both the MATLAB environment and the generated code.
- Improves performance for MEX workflows.
- Simplifies Simulink workflows using MATLAB Function blocks as Simulink does not natively support dlarray objects.

```
function a = foo(in)
dlIn = dlarray(in, 'SSC');

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dlA = predict(dlnet, dlIn);

a = extractdata(dlA);

end
```

To see an example of dlnetwork and dlarray usage with MATLAB Coder, see “Generate Digit Images Using Variational Autoencoder on Intel CPUs” on page 38-124.

dlarray Object Functions with Code Generation Support

For code generation, you are restricted to the deep learning array object functions listed in this table.

dims	Dimension labels for dlarray
extractdata	Extract data from dlarray
finddim	Find dimensions with specified label
stripdims	Remove dlarray labels

Deep Learning Toolbox Functions with dlarray Code Generation Support

Deep Learning Operations

Function	Description
`fullyconnect`	The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.
`sigmoid`	The sigmoid activation operation applies the sigmoid function to the input data.
`softmax`	The softmax activation operation applies the softmax function to the channel dimension of the input data.

MATLAB Functions with dlarray Code Generation Support

Unary Element-wise Functions

<table>
<tr><th>Function</th><th>Notes and Limitations</th></tr>
<tr><td><code>abs</code></td><td>The output <code>dlarray</code> has the same data format as the input <code>dlarray</code>.</td></tr>
<tr><td><code>cos</code></td><td rowspan="5">The output <code>dlarray</code> has the same data format as the input <code>dlarray</code>.</td></tr>
<tr><td><code>cosh</code></td></tr>
<tr><td><code>cot</code></td></tr>
<tr><td><code>csc</code></td></tr>
<tr><td><code>exp</code></td></tr>
<tr><td><code>log</code></td><td>The output <code>dlarray</code> has the same data format as the input <code>dlarray</code>.
Because <code>dlarray</code> does not support complex numbers, the input <code>dlarray</code> must have nonnegative values.</td></tr>
<tr><td><code>sec</code></td><td rowspan="4">The output <code>dlarray</code> has the same data format as the input <code>dlarray</code>.</td></tr>
<tr><td><code>sign</code></td></tr>
<tr><td><code>sin</code></td></tr>
<tr><td><code>sinh</code></td></tr>
<tr><td><code>sqrt</code></td><td>The output <code>dlarray</code> has the same data format as the input <code>dlarray</code>.
Because <code>dlarray</code> does not support complex numbers, the input <code>dlarray</code> must have nonnegative values.</td></tr>
<tr><td><code>tan</code></td><td rowspan="3">The output <code>dlarray</code> has the same data format as the input <code>dlarray</code>.</td></tr>
<tr><td><code>tanh</code></td></tr>
<tr><td><code>uplus</code>, <code>+</code></td></tr>
</table>

Function	Notes and Limitations
`uminus`, `-`	

Extrema Functions

Function	Notes and Limitations
`ceil`	The output `dlarray` has the same data format as the input `dlarray`.
`eps`	• The output `dlarray` has the same data format as the input `dlarray`. • Use `eps(ones('like', x))` to get a scalar epsilon value based on the data type of a `dlarray` `x`.
`fix`	The output `dlarray` has the same data format as the input `dlarray`.
`floor`	The output `dlarray` has the same data format as the input `dlarray`.
`round`	• Only the syntax `Y = round(X)` is supported. • The output `dlarray` has the same data format as the input `dlarray`.

Conversion Functions

Function	Notes and Limitations
`double`	The output is a `dlarray` that contains data of type `double`.
`logical`	The output is a `dlarray` that contains data of type `logical`.
`single`	The output is a `dlarray` that contains data of type `single`.

Comparison Functions

Function	Notes and Limitations
`isequal`	• The syntax with more than two input arguments is not supported. • Two `dlarray` inputs are equal if the numeric data they represent are equal and if they both are either formatted with the same data format or unformatted.
`isequaln`	• The syntax with more than two input arguments is not supported. • Two `dlarray` inputs are equal if the numeric data they represent are equal (treating `NaN`s as equal) and if they both are either formatted with the same data format or unformatted.

Data Type and Value Identification Functions

Function	Notes and Limitations
isfloat	The software applies the function to the underlying data of an input dlarray.
islogical	
isnumeric	
isreal	Because dlarray does not support complex numbers, this function always returns true for a dlarray input.

Size Identification Functions

Function	Notes and Limitations
length	N/A
ndims	If the input dlarray dlX is formatted, then ndims(dlX) returns the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.
numel	N/A
size	If the input dlarray dlX is formatted, then size(dlX) returns a vector of length equal to the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.

Creator Functions

Function	Notes and Limitations
false	Only the 'like' syntax is supported for dlarray.
inf	
nan	
ones	
rand	
true	
zeros	

See Also

Objects
dlarray | dlnetwork

Related Examples

- "Generate Digit Images Using Variational Autoencoder on Intel CPUs" on page 38-124

More About

- “dlarray Limitations for Code Generation” on page 18-8
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Train Network Using Custom Training Loop” (Deep Learning Toolbox)
- “Make Predictions Using dlnetwork Object” (Deep Learning Toolbox)

dlarray Limitations for Code Generation

In this section...

"Recommended Usage" on page 18-8
"Limitations" on page 18-8

Recommended Usage

For code generation, use the `dlarray` function to create deep learning arrays. For example, suppose you have a pretrained `dlnetwork` network object in the `mynet.mat` MAT-file. To predict the responses for this network, create an entry-point function in MATLAB as shown in this code.

```
function a = foo(in)
dlIn = dlarray(in, 'SSC');

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dlA = predict(dlnet, dlIn);

a = extractdata(dlA);

end
```

Limitations

For deep learning arrays, code generation has the following limitations:

- The data format argument of the `dlarray` object must be a compile-time constant. For example,

  ```
  function out = foo()

  dlA = dlarray(ones(5,4),'SSC'); %fmt 'SSC' is constant
   .
   .
   .
  end
  ```

- The data input to the `dlarray` object must be fixed-size. For example, the `dlarray` `dlA` is not supported as `A` is variable-sized.

  ```
  function dlA = foo()

  A = ones(5,4);
  coder.varsize('A') %'A' is variable sized.

  dlA = dlarray(A, 'SSC'); % Error: not supported.

  end
  ```

- Code generation does not support creating a `dlarray` type object by using the `coder.typeof` function with upper bound size and variable dimensions specified. For example,

```
function dlA = foo()

A = dlarray(ones(5,4),'SC');
A_type = coder.typeof(A,[5 10],[1 0]); % Error: not supported.

end
```

Code generation supports use of `coder.typeof` without the size arguments. For example,

```
A = dlarray(ones(5,4),'SC');
A_type = coder.typeof(A);
```

- The code generation report does not display the size of the `dlarray` object. The size is always displayed as `1x1`.

```
 7 end
 8
 9 % generate random noise
10 randomNoise = dlarray(randn(1,1,latentDim,25),'SSCB');
11
12 if coder.target('MATLAB') && strcmp(Environment, 'gpu')
13     randomNoise = gpuArray(randomNoise);
```

EXPRESSION INFO
dlarray(randn(1,1,latentDim,25),'SSCB')
Size: 1 × 1
Class: dlarray

- In MATLAB, `dlarray` enforces the order of labels `'SCBTU'`. This enforcement eliminates ambiguous semantics in operations, which implicitly match labels between inputs. This behavior is mimicked during MEX code generation. However, for standalone code generation such as static, dynamic libraries, or executables, the data format follows the specification of the `fmt` argument of the `dlarray` object. As a result, if the input or output of an entry-point function is a `dlarray` object and its order of labels is not `'SCBTU'`, then the data layout will be different between the MATLAB environment and standalone code.

  For example, consider a function `foo` with a `dlarray` object as an output.

```
function dlA = foo()
rng default
dlA = dlarray(rand(5,4), 'BC');

end
```

  In MATLAB, `dlA` is `4(C)`-by-`5(B)`.

```
dlA =

  4(C) × 5(B) dlarray

    0.8147    0.9058    0.1270    0.9134    0.6324
    0.0975    0.2785    0.5469    0.9575    0.9649
    0.1576    0.9706    0.9572    0.4854    0.8003
    0.1419    0.4218    0.9157    0.7922    0.9595
```

  For standalone code generation, `dlA` is `5(B)`-by-`4(C)`.

- Code generation does not support indexing with `dlarray` objects.
- For code generation, the `dlarray` input to the `predict` method of the `dlnetwork` object must be `single` data type.

See Also

Objects
`dlarray` | `dlnetwork`

Related Examples

- "Generate Digit Images Using Variational Autoencoder on Intel CPUs" on page 38-124

More About

- "Code Generation for dlarray" on page 18-2

- "Define Custom Training Loops, Loss Functions, and Networks" (Deep Learning Toolbox)
- "Train Network Using Custom Training Loop" (Deep Learning Toolbox)
- "Make Predictions Using dlnetwork Object" (Deep Learning Toolbox)

Defining Functions for Code Generation

- “Code Generation for Variable Length Argument Lists” on page 19-2
- “Specify Number of Entry-Point Function Input or Output Arguments to Generate” on page 19-3
- “Code Generation for Anonymous Functions” on page 19-6
- “Code Generation for Nested Functions” on page 19-7

Code Generation for Variable Length Argument Lists

When you use `varargin` and `varargout` for code generation, there are these restrictions:

- 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`.

See Also

More About

- “Nonconstant Index into varargin or varargout in a for-Loop” on page 36-14
- “Specify Number of Entry-Point Function Input or Output Arguments to Generate” on page 19-3

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`.

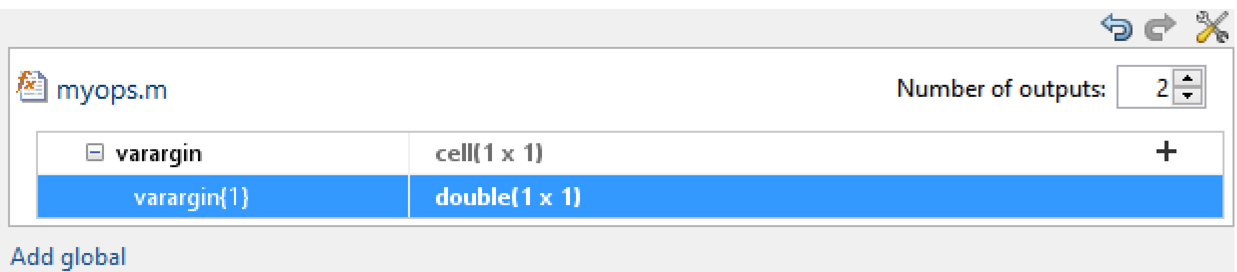
```
codegen myops -args {3} -report
```

If you use the MATLAB Coder app:

1 On the **Define Input Types** page, click **Let me enter input or global types directly**.

2 To add an argument, in the variables table, to the right of `varargin`, click + .

3 Specify the properties for each argument.



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
x = a + b;
y = a * b;
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 output arguments for an entry-point function that uses `varargout`.

Rewrite `myops` to use `varargout`.

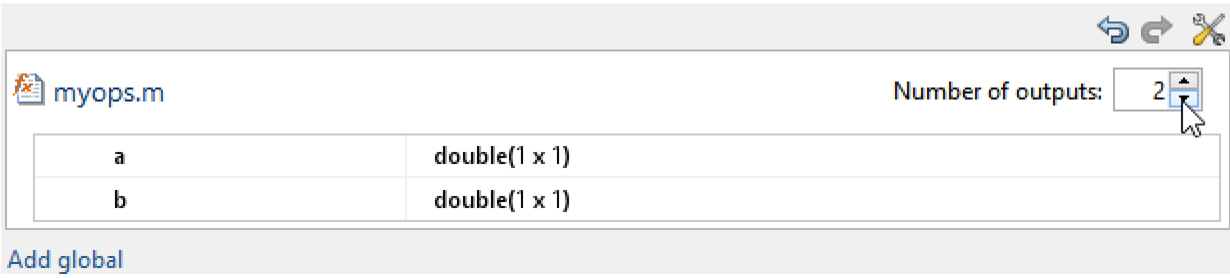
```
function varargout = myops(a,b)
%#codegen
varargout{1} = a + b;
varargout{2} = a * b;
end
```

Generate code for one output argument.

```
codegen myops -args {2 3} -nargout 1 -report
```

If you use the MATLAB Coder app, to specify the number of outputs when a function returns `varargout` or to generate fewer outputs than the function defines:

1. On the **Define Input Types** page, define the input types manually or by using **Autodefine Input Types**.
2. In **Number of outputs**, select the number.



See Also

More About

- “Code Generation for Variable Length Argument Lists” on page 19-2
- “Specify Properties of Entry-Point Function Inputs” on page 27-43

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.

See Also

More About

- “MATLAB Classes Definition for Code Generation” on page 15-2
- “Cell Array Limitations for Code Generation” on page 9-8
- “Parameterizing Functions”

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.

See Also

More About

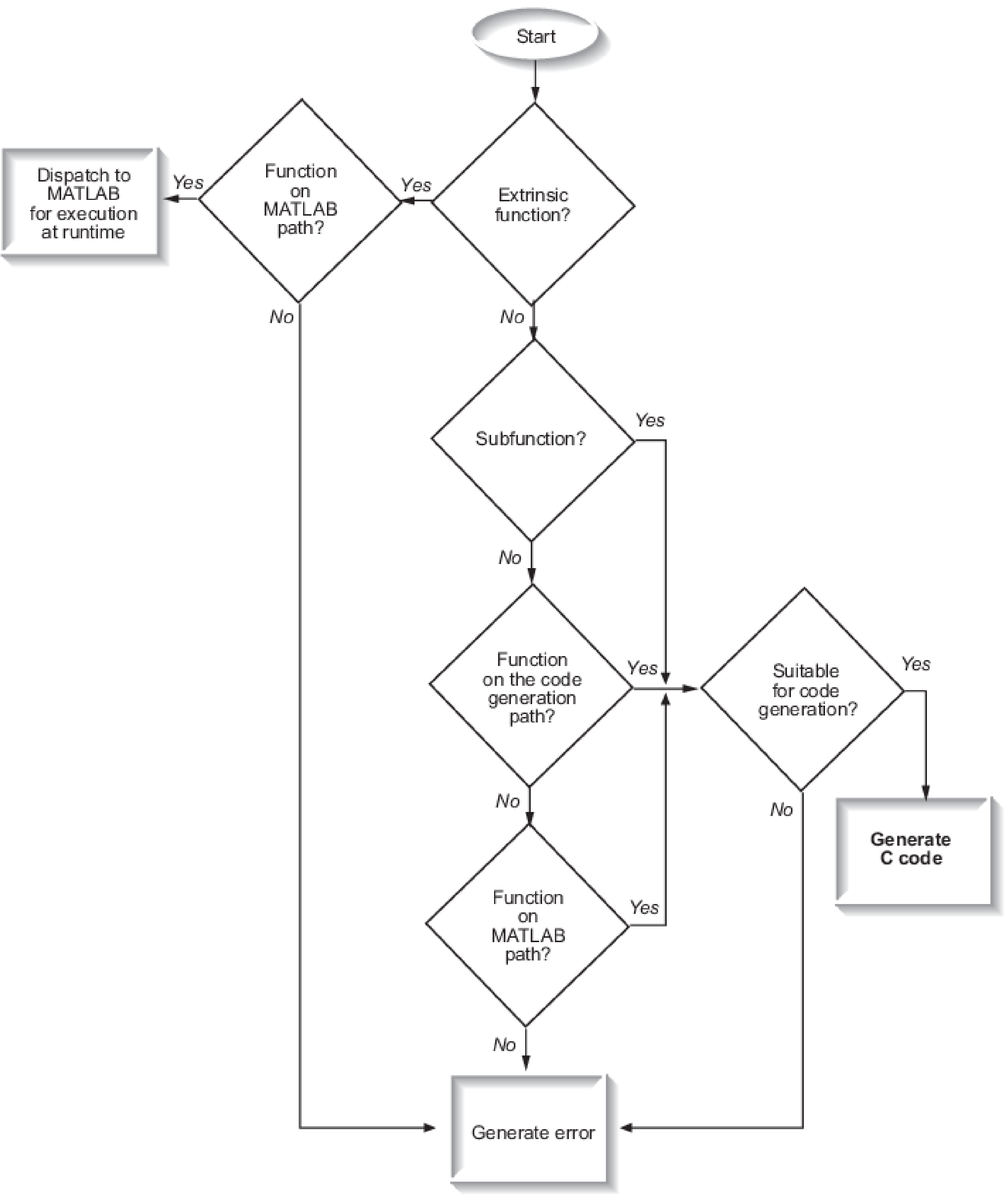
- “MATLAB Classes Definition for Code Generation” on page 15-2
- “Handle Object Limitations for Code Generation” on page 15-21
- “Cell Array Limitations for Code Generation” on page 9-8
- “Code Generation for Recursive Functions” on page 20-14

Calling Functions for Code Generation

- "Resolution of Function Calls for Code Generation" on page 20-2
- "Resolution of File Types on Code Generation Path" on page 20-5
- "Compilation Directive %#codegen" on page 20-7
- "Use MATLAB Engine to Execute a Function Call in Generated Code" on page 20-8
- "Code Generation for Recursive Functions" on page 20-14
- "Force Code Generator to Use Run-Time Recursion" on page 20-17
- "Avoid Duplicate Functions in Generated Code" on page 20-20

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
Suitable for code generation?
Yes
Generate C code
No
No
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 20-4.
- 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 “Using the coder.extrinsic Construct” on page 20-9. 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. If 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 20-5

Compile Path Search Order

During code generation, function calls are resolved on two paths:

1. Code generation path

   MATLAB searches this path first during code generation. The code generation path contains the toolbox functions supported for code generation.
2. MATLAB path

   If the function is not on the code generation path, MATLAB searches this path.

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

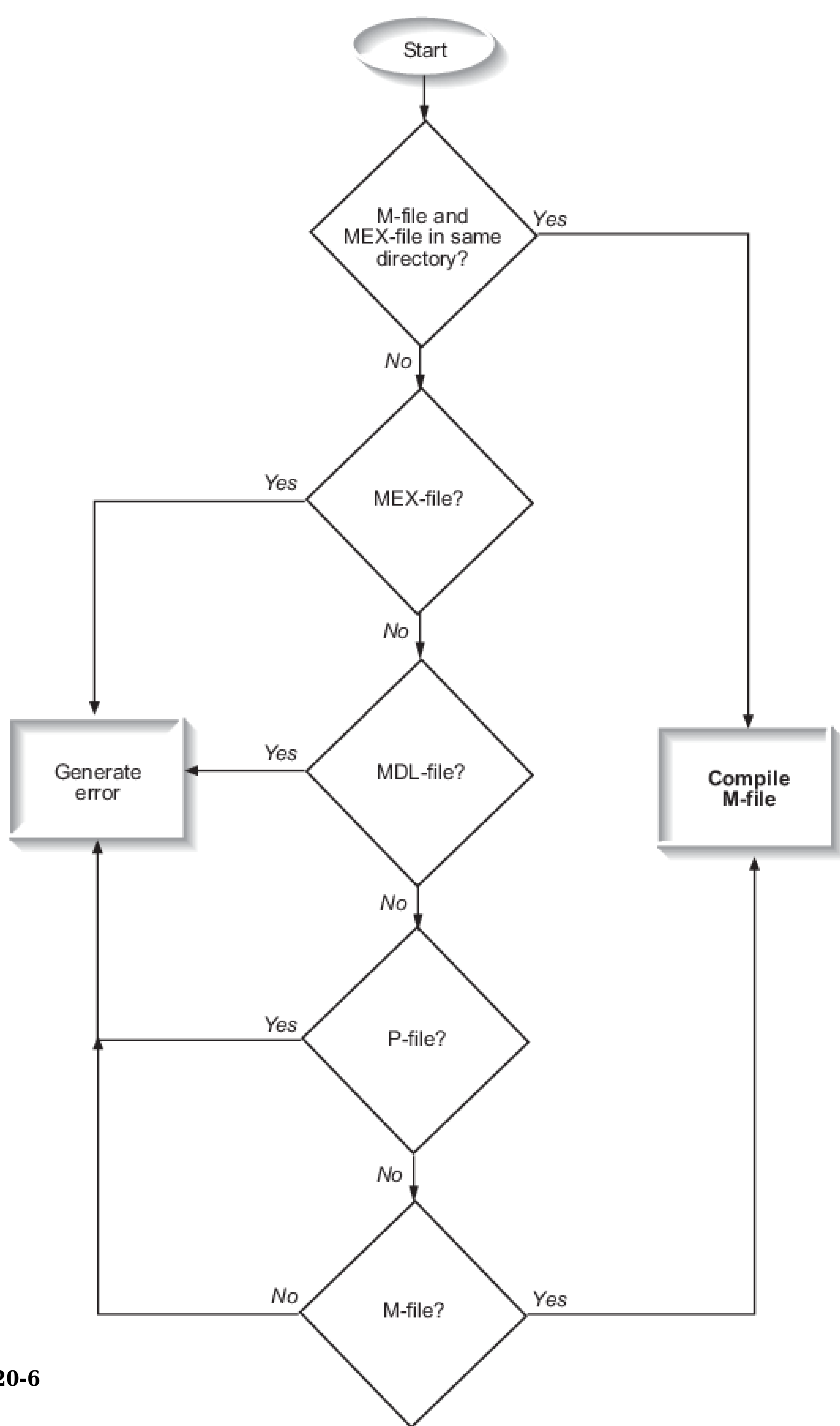
When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. A file on the code generation path shadows a file of the same name on the MATLAB path.

For more information on how to add additional folders to the code generation path, see “Paths and File Infrastructure Setup” on page 27-76.

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
MDL-file?
Yes
Generate
error
No
P-file?
Yes
No
M-file?
No
Yes
Compile
M-file

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

....
```

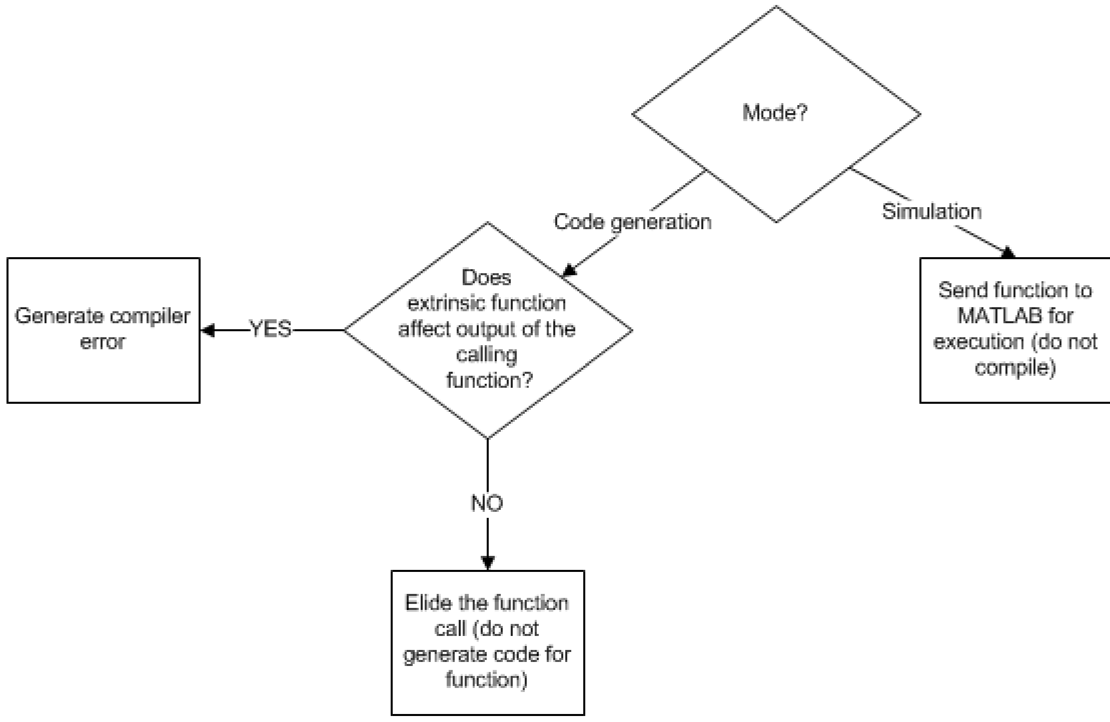
Note The `%#codegen` directive is not necessary for MATLAB Function blocks. Code inside a MATLAB Function block is always intended for code generation. The `%#codegen` directive, or the absence of it, does not change the error checking behavior.

Use MATLAB Engine to Execute a Function Call in Generated Code

When processing a call to a function `foo` in your MATLAB code, the code generator finds the definition of `foo` and generates code for its body. In some cases, you might want to bypass code generation and instead use the MATLAB engine to execute the call. Use `coder.extrinsic('foo')` to declare that calls to `foo` do not generate code and instead use the MATLAB engine for execution. In this context, `foo` is referred to as an extrinsic function. This functionality is available only when the MATLAB engine is available during execution. Examples of such situations include execution of MEX functions, Simulink simulations, or function calls at the time of code generation (also known as compile time).

If you generate standalone code for a function that calls `foo` and includes `coder.extrinsic('foo')`, the code generator attempts to determine whether `foo` affects the output. If `foo` does not affect the output, the code generator proceeds with code generation, but excludes `foo` from the generated code. Otherwise, the code generator produces a compilation error.

Including the `coder.extrinsic('foo')` directive inside a certain MATLAB function declares all calls to `foo` inside that MATLAB function as extrinsic. Alternatively, you might want to narrow the scope of extrinsic declaration to just one call to `foo`. See “Calling MATLAB Functions Using feval” on page 20-11.



When To Declare a Function as Extrinsic

These are some common situations in which you might consider declaring a MATLAB function as extrinsic:

- The function performs display or logging actions. Such functions are useful primarily during simulation and are not used in embedded systems.
- In your MEX execution or Simulink simulation, you want to use a MATLAB function that is not supported for code generation. This workflow does not apply to non-simulation targets.
- You instruct the code generator to constant fold a function call by using `coder.const`. In such situations, the function is called only during code generation when the MATLAB engine is available for executing the call.

Using the coder.extrinsic Construct

To declare a function `foo` as extrinsic, include this statement in your MATLAB code.

```
coder.extrinsic('foo')
```

When declaring functions as extrinsic for code generation, adhere to these rules:

- Declare the function as extrinsic before you call it.
- Do not use the extrinsic declaration in conditional statements.
- Assign the return value of an extrinsic function to a known type. See “Working with mxArrays” on page 20-11.

For additional information and examples, see `coder.extrinsic`.

The code generator automatically treats many common MATLAB visualization functions, such as `plot`, `disp`, and `figure`, as extrinsic. You do not have to explicitly declare them as extrinsic functions by using `coder.extrinsic`. 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 the MATLAB engine. If you generate a library or executable, the generated code does not contain calls to the `plot` function.

If you generate MEX or standalone C/C++ code by using MATLAB Coder, the code generation report highlights calls from your MATLAB code to extrinsic functions. By inspecting the report, you can determine which functions are supported only in the MATLAB environment.

mystats.m

```
1  function [mean, stdev] = mystats(vals)
2  %#codegen
3
4  % Calculates a statistical mean and a standard
5  % deviation for the values in vals.
6
7  len = length(vals)
8  mean = avg(vals, l
9  stdev = sqrt(sum((                         ))/len);
10 plot(vals,'-+');
11
12 function z = avg(v
13 z = sum(v)/l;
```

EXPRESSION INFO
plot(vals,'-+')
Size: 1 × 1
Class: mxArray
plot is an extrinsic function.

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.
- Instead of using the coder.extrinsic construct, call the MATLAB function using feval. This approach is described in the next section.

Extrinsic Declaration for Nonstatic Methods

Suppose that you define a class myClass that has a nonstatic method foo, and then create an instance obj of this class. If you want to declare the method obj.foo as extrinsic in your MATLAB code that you intend for code generation, follow these rules:

- Write the call to foo as a function call. Do not write the call by using the dot notation.
- Declare foo to be extrinsic by using the syntax coder.extrinsic('foo').

For example, define myClass as:

```
classdef myClass
    properties
        prop = 1
    end
    methods
        function y = foo(obj,x)
            y = obj.prop + x;
        end
    end
end
```

Here is an example MATLAB function that declares foo as extrinsic.

```
function y = myFunction(x) %#codegen
coder.extrinsic('foo');
obj = myClass;
```

```
y = foo(obj,x);
end
```

Nonstatic methods are also known as ordinary methods. See “Define Class Methods and Functions”.

Additional Uses

Use the `coder.extrinsic` construct to:

- Call MATLAB functions that do not produce output during simulation, without generating unnecessary code.
- 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 20-11.

Calling MATLAB Functions Using feval

To narrow the scope of extrinsic declaration to just one function call, use the function `feval`. `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 compile 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`.

The code generator does not support the use of `feval` to call local functions or functions that are located in a private folder.

Working with mxArrays

The output of an extrinsic function is an `mxArray`, also known as a MATLAB array. The only valid operations for `mxArrays` are:

- Storing an `mxArray` in a variable.
- Passing an `mxArray` to a function.
- Returning an `mxArray` from a function back to MATLAB.
- Converting an `mxArray` to a known type at run time. To perform this action, assign the `mxArray` to a variable whose type is already defined by a prior assignment. See example below.

To use an `mxArray` returned by an extrinsic function in other operations (for example, returning it from a MATLAB Function block to Simulink execution), you must first convert it to a known type.

If the input arguments of a function are `mxArrays`, the code generator automatically treats the function as extrinsic.

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 that it is assigned to. 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. You can pass these mxArrays to another MATLAB function — in this case, min. Because the inputs passed to min are mxArrays, the code generator automatically treats min as an extrinsic function. As a result, min returns an mxArray.

While generating a MEX function by using MATLAB Coder, you can directly assign this mxArray returned by min to the output y because the MEX function returns its output to MATLAB.

But if you put foo in a MATLAB Function block in a Simulink model and then update or run the model, you get this error:

```
Function output 'y' cannot be an mxArray in this context.
Consider preinitializing the output variable with a known type.
```

This error occurs because returning an mxArray back to Simulink is not supported. 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 Using Extrinsic Functions

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
- 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
- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.
- You can call extrinsic functions with up to 64 inputs and 64 outputs.

See Also
`coder.const` | `coder.extrinsic`

Code Generation for Recursive Functions

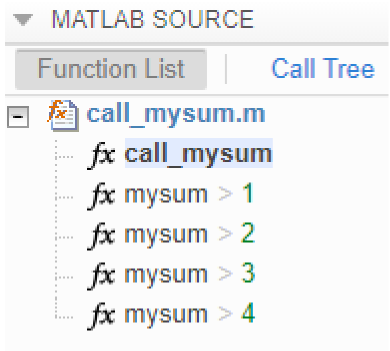
To generate code for recursive MATLAB functions, the code generator uses compile-time recursion on page 20-14 or run-time recursion on page 20-15. You can influence whether the code generator uses compile-time or run-time recursion by modifying your MATLAB code. See “Force Code Generator to Use Run-Time Recursion” on page 20-17.

You can disallow recursion on page 20-15 or disable run-time recursion on page 20-15 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 20-16.

Compile-Time Recursion

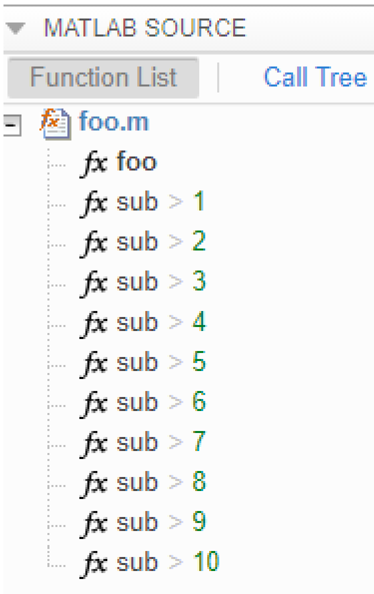
With compile-time recursion, the code generator creates multiple versions of a recursive function in the generated code. The inputs to each version have values or sizes that are customized for that version. These versions are known as function specializations. You can tell that the code generator used 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.



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
```

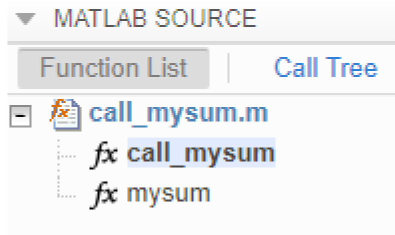
In the code generation report, on the **Function List** tab, you see the function specializations for MATLAB function `sub`.



However, the C code does not contain the specializations. It contains one function that returns the value 55.

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.



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 34-15 setting is ignored for run-time recursion.

See Also

More About

- “Force Code Generator to Use Run-Time Recursion” on page 20-17
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 36-4
- “Compile-Time Recursion Limit Reached” on page 36-7
- “Configure Build Settings” on page 27-13
- “Code Generation Reports” on page 28-7

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 on page 20-14, 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 on page 20-15, 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:

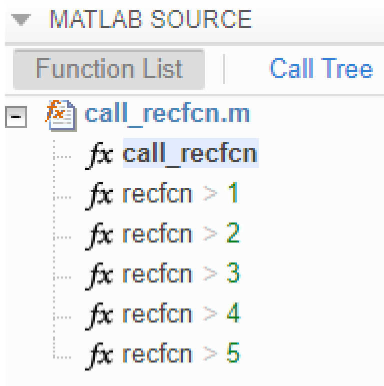
- “Treat the Input to the Recursive Function as a Nonconstant” on page 20-17
- “Make the Input to the Recursive Function Variable-Size” on page 20-18
- “Assign Output Variable Before the Recursive Call” on page 20-19

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
```

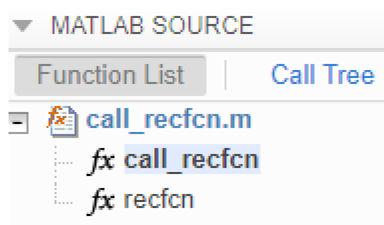
`call_recfcn` calls `recfcn` with the value 5 for the second argument. `recfcn` calls itself recursively until `x` is 1. For each `recfcn` call, the input argument `x` has a different value. The code generator produces five specializations of `recfcn`, one for each call. After you generate code, you can see the specializations in the code generation report.



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.



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 20-14
- "Output Variable Must Be Assigned Before Run-Time Recursive Call" on page 36-4
- "Compile-Time Recursion Limit Reached" on page 36-7

Avoid Duplicate Functions in Generated Code

Issue

You generate code and it contains multiple, duplicate copies of the same functions, with only slight differences, such as modifications to the function signature. For example, your generated code might contain functions called `foo` and `b_foo`. Duplicate functions can make the generated code more difficult to analyze and manage.

Cause

Duplicate functions in the generated code are the result of function specializations. The code generator specializes functions when it detects that they differ at different call sites by:

- Number of input or output variables.
- Type of input or output variables.
- Size of input or output variables.
- Values of input variables.

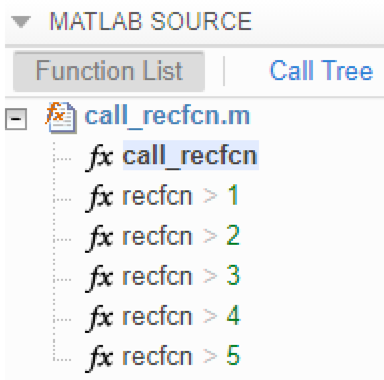
In some cases, these specializations are necessary for the generated C/C++ code because C/C++ functions do not have the same flexibility as MATLAB functions. In other cases, the code generator specializes functions to optimize the generated code or because of a lack of information.

Solution

In certain cases, you can alter your MATLAB code to avoid the generation of duplicate functions.

Identify Duplicate Functions by Using Code Generation Report

You can determine whether the code generator created duplicate functions by inspecting the code generation report or in Simulink, the MATLAB Function report. The report shows a list of the duplicate functions underneath the entry-point function. For example:



Duplicate Functions Generated for Multiple Input Sizes

If your MATLAB code calls a function multiple times and passes inputs of different sizes, the code generator can create specializations of the function for each size. To avoid this issue, use

coder.ignoreSize on the function input. For example, this code uses coder.ignoreSize to avoid creating multiple copies of the function indexOf:

```
function [out1, out2] = test1(in)
  a = 1:10;
  b = 2:40;
  % Without coder.ignoreSize duplicate functions are generated
  out1 = indexOf(coder.ignoreSize(a), in);
  out2 = indexOf(coder.ignoreSize(b), in);
end

function index = indexOf(array, value)
  coder.inline('never');
  for i = 1:numel(array)
      if array(i) == value
          index = i;
          return
      end
  end
  index = -1;
  return
end
```

To generate code, enter:

```
codegen test1 -config:lib -report -args {1}
```

Duplicate Functions Generated for Different Input Values

If your MATLAB code calls a function and passes multiple different constant inputs, the code generator can create specializations of the function for each different constant. In this case, use coder.ignoreConst to indicate to the code generator not to treat the value as an immutable constant. For example:

```
function [out3, out4] = test2(in)
  c = ['a', 'b', 'c'];
  if in > 0
      c(2)='d';
  end
  out3 = indexOf(c, coder.ignoreConst('a'));
  out4 = indexOf(c, coder.ignoreConst('b'));
end


function index = indexOf(array, value)
  coder.inline('never');
  for i = 1:numel(array)
      if array(i) == value
          index = i;
          return
      end
  end
  index = -1;
  return
end
```

To generate code, enter:

```
codegen test2 -config:lib -report -args {1}
```

Duplicate Functions Generated for Different Number of Outputs

If your MATLAB code calls a function and accepts a different number of outputs at different call sites, the code generator can produce specializations for each call. For example:

```
[a b] = foo();
c = foo();
```

To make each call return the same number of outputs and avoid duplicate functions, use the ~ symbol:

```
[a b] = foo();
[c, ~] = foo();
```

See Also

coder.ignoreConst | coder.ignoreSize | coder.varsize

More About

- “Code Generation Reports” on page 28-7
- “Force Code Generator to Use Run-Time Recursion” on page 20-17

21

Fixed-Point Conversion

- “Detect Dead and Constant-Folded Code” on page 21-2
- “Convert MATLAB Code to Fixed-Point C Code” on page 21-5
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 21-6
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 21-17
- “Specify Type Proposal Options” on page 21-29
- “Detect Overflows” on page 21-32
- “Replace the exp Function with a Lookup Table” on page 21-40
- “Replace a Custom Function with a Lookup Table” on page 21-47
- “Enable Plotting Using the Simulation Data Inspector” on page 21-53
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 21-54
- “View and Modify Variable Information” on page 21-64
- “Automated Fixed-Point Conversion” on page 21-67
- “Convert Fixed-Point Conversion Project to MATLAB Scripts” on page 21-85
- “Generated Fixed-Point Code” on page 21-87
- “Fixed-Point Code for MATLAB Classes” on page 21-92
- “Automated Fixed-Point Conversion Best Practices” on page 21-94
- “Replacing Functions Using Lookup Table Approximations” on page 21-100
- “MATLAB Language Features Supported for Automated Fixed-Point Conversion” on page 21-101
- “Inspecting Data Using the Simulation Data Inspector” on page 21-103
- “Custom Plot Functions” on page 21-105
- “Data Type Issues in Generated Code” on page 21-106

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 21-6 and “Propose Fixed-Point Data Types Based on Derived Ranges” on page 21-17.
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 21-83.
11. Verify the behavior of the fixed-point MATLAB code. See “Testing Numerics” on page 21-84.
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.

See Also

Related Examples

- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 21-6
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 21-17

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\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

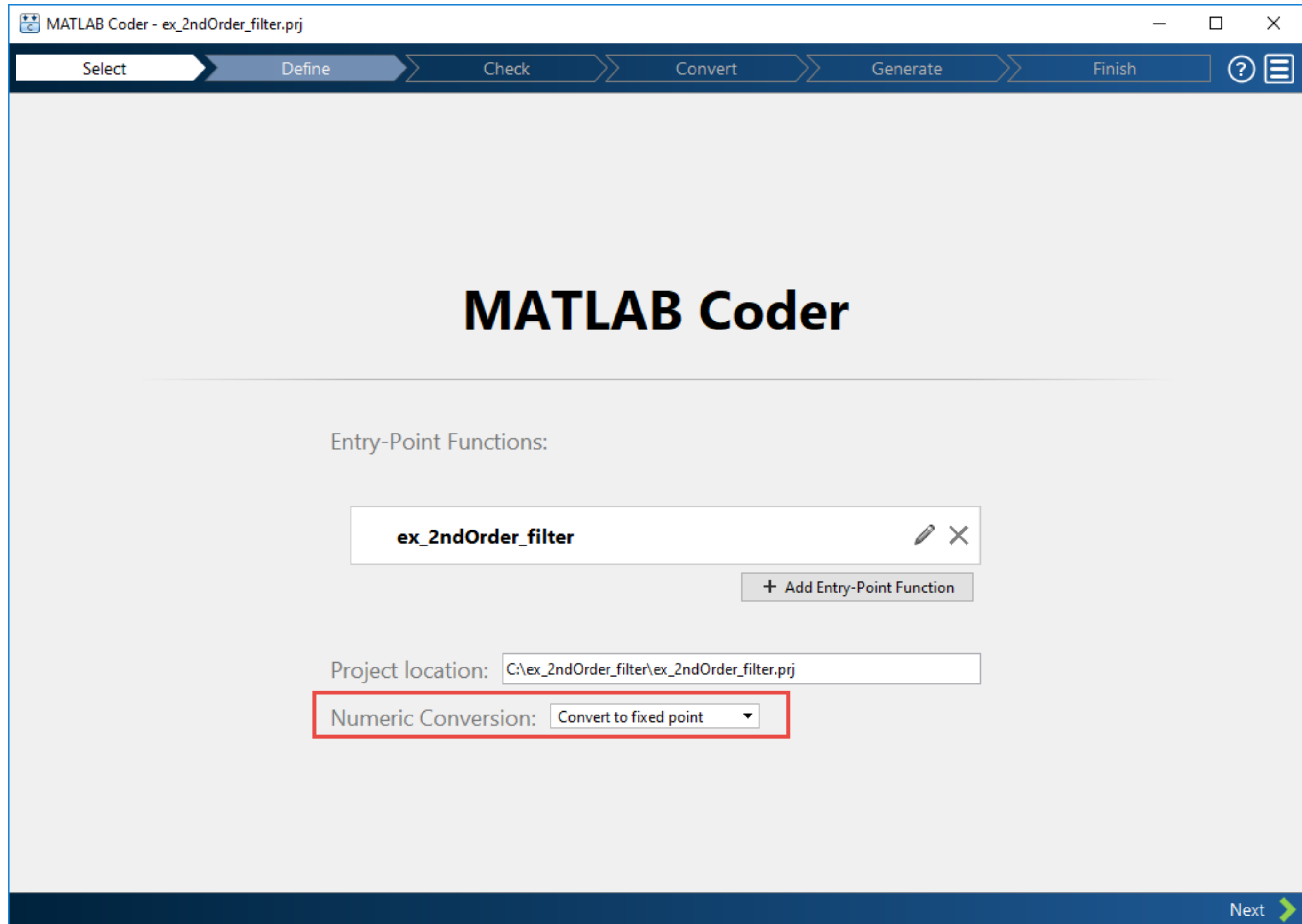
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

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`.



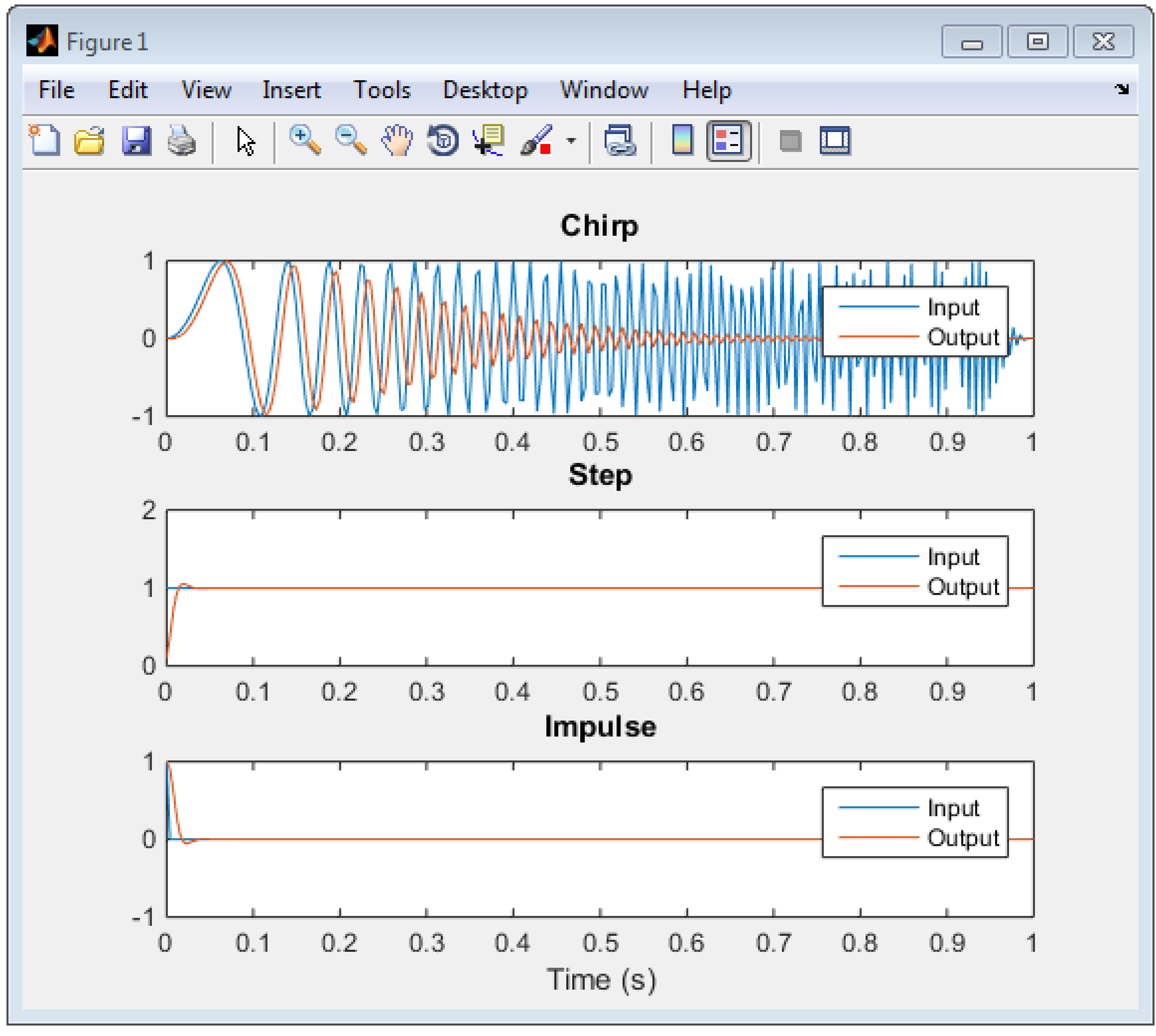
2 Click **Next** to go to the **Define Input Types** step.

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`.

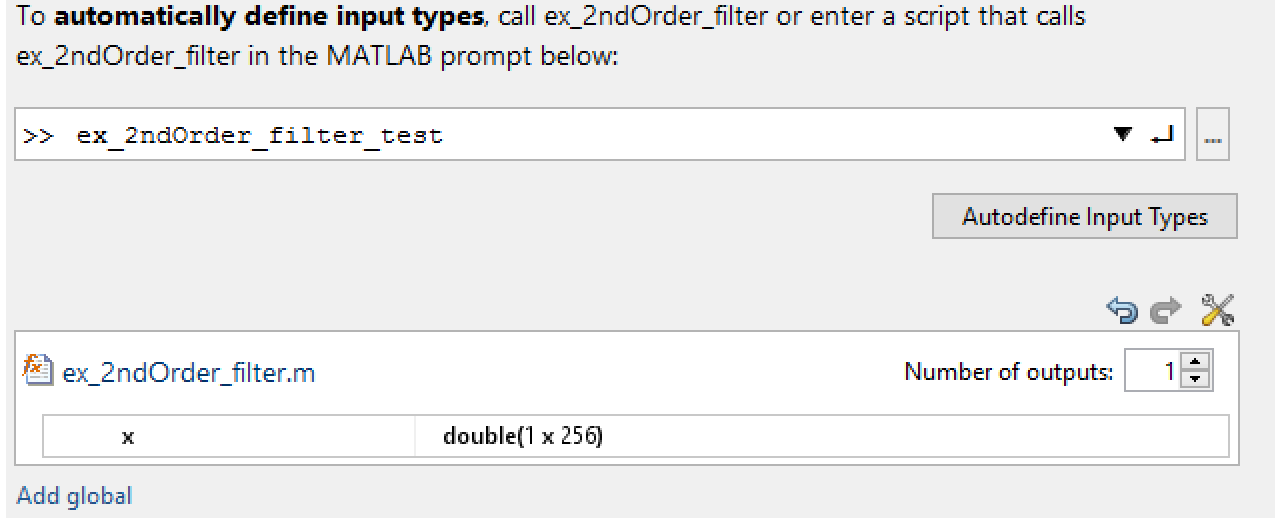
Define Input Types

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)`.



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

Check for Run-Time Issues

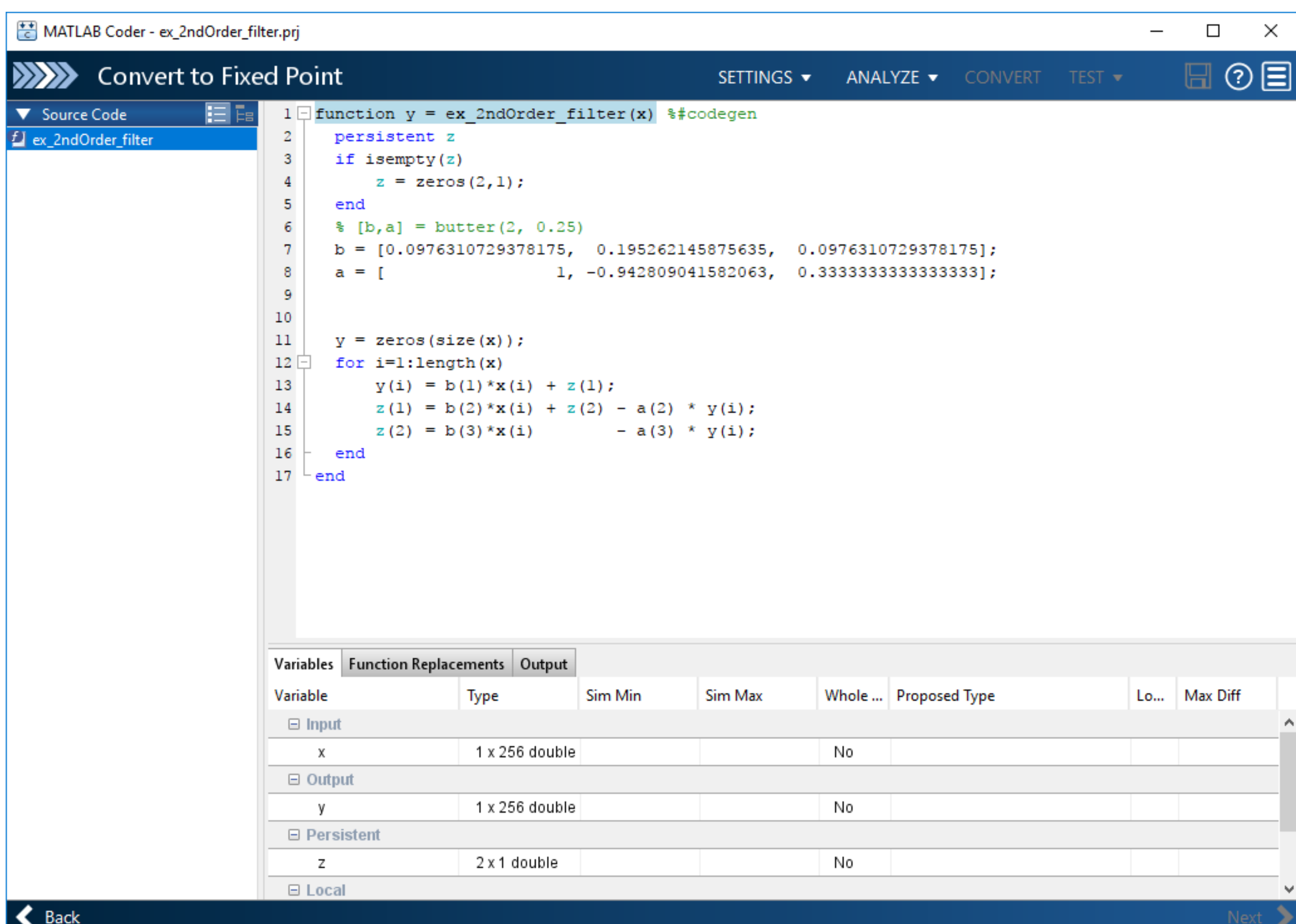
The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `ex_2ndOrder_filter_test` replacing calls to `ex_2ndOrder_filter` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the 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 21-64.

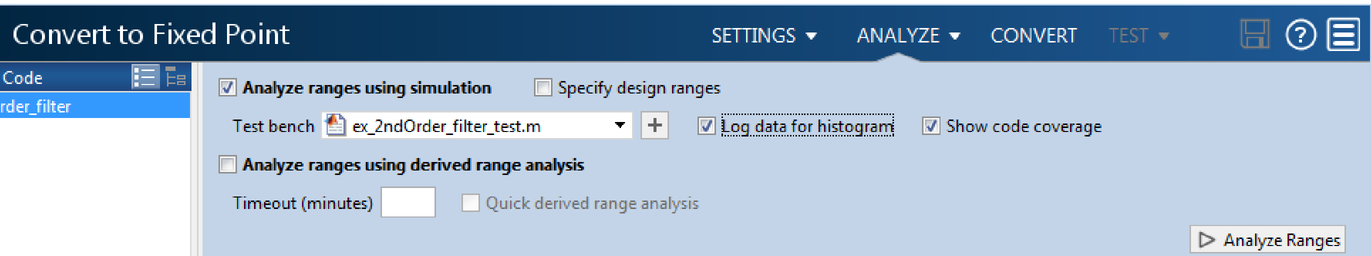


On the **Function Replacements** tab, the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 21-70.

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.

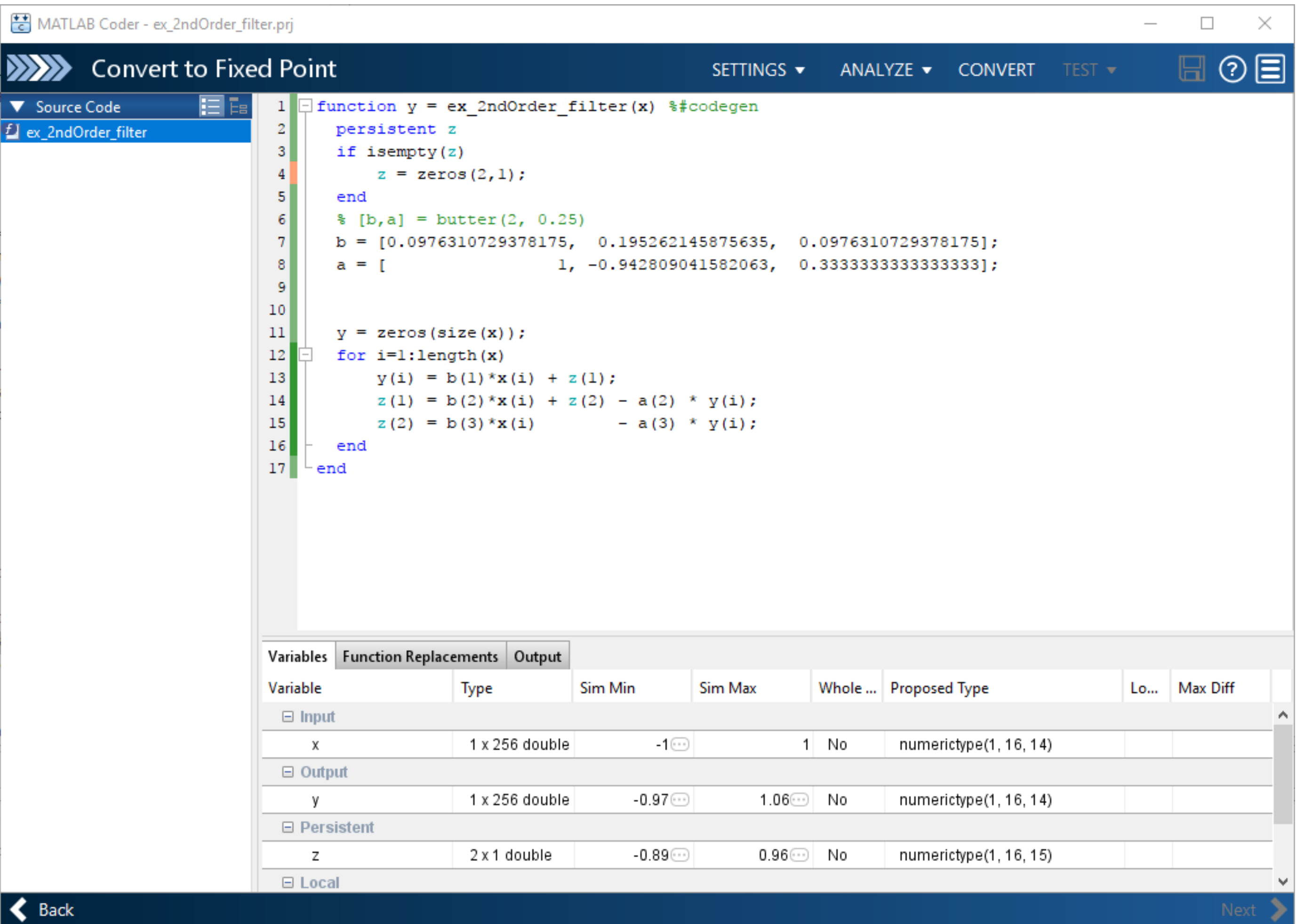
3 Select **Log data for histogram**.

   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.



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.



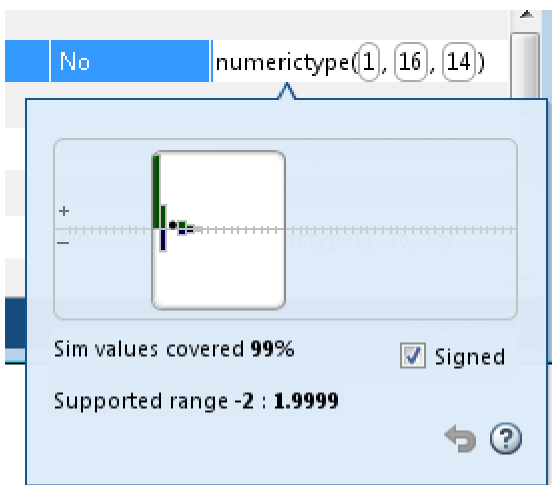
If a value has `...` next to it, the value is rounded. Pause over the `...` to view the actual value.

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.



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 21-81.

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 **Output** tab. See "Validating Types" on page 21-83.

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

```
6  function y = ex_2ndOrder_filter_fixpt(x) %#codegen
7    fm = get_fimath();
8
9    persistent z
10   if isempty(z)
11       z = fi(zeros(2,1), 1, 16, 15, fm);
12   end
13   % [b,a] = butter(2, 0.25)
14   b = fi([0.0976310729378175,  0.195262145875635,  0.0976310729378175], 0, 16, 18, fm);
15   a = fi([                 1, -0.942809041582063,  0.3333333333333333], 1, 16, 14, fm);
16
17
18   y = fi(zeros(size(x)), 1, 16, 14, fm);
19   for i=1:length(x)
20       y(i) = b(1)*x(i) + z(1);
21       z(1) = fi_signed(b(2)*x(i) + z(2)) - a(2) * y(i);
22       z(2) = fi_signed(b(3)*x(i))        - a(3) * y(i);
23   end
24 end
25
26
27
28 function y = fi_signed(a)
```

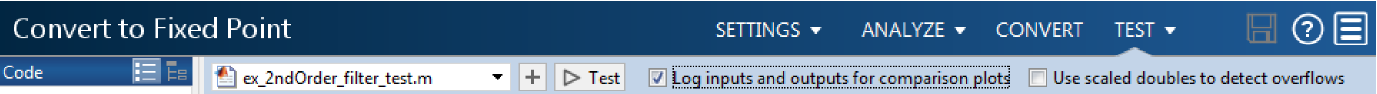
Output Files
ex_2ndOrder_filter_fixpt.m
ex_2ndOrder_filter_wrapper_fixp
ex_2ndOrder_filter_report.html
ex_2ndOrder_filter_fixpt_args.ma
ex_2ndOrder_filter_wrapper_fixp
report.mldatx

Variables | Function Replacements | Output

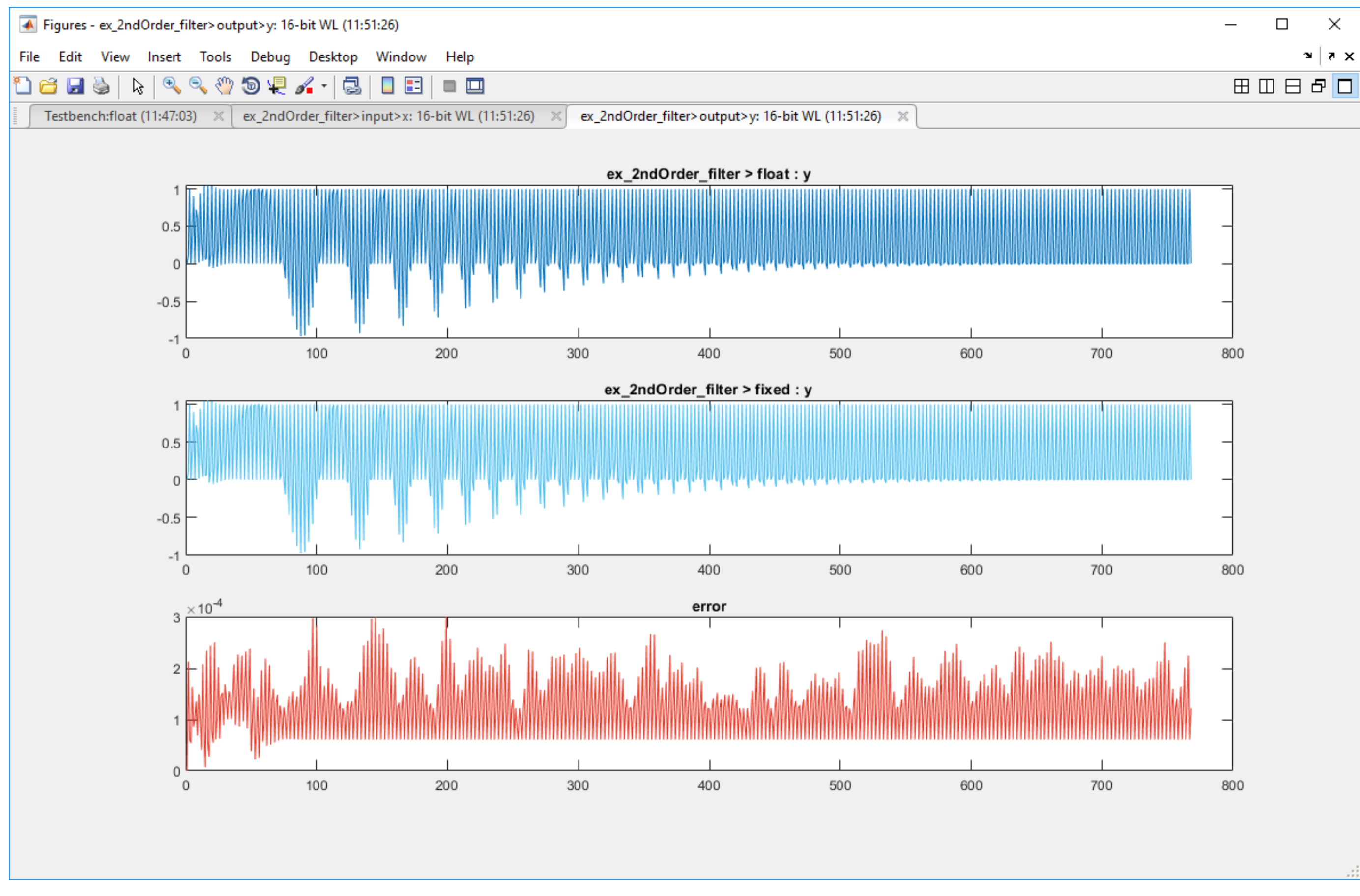
Variable	Type	Size	Signed	Word Length	Fraction Length
⊟ Input					
x	embedded.fi	1 x 256	Yes	16	14
⊟ Output					
y	embedded.fi	1 x 256	Yes	16	14
⊟ Persistent					
z	embedded.fi	2 x 1	Yes	16	15
⊟ Local					

Back   Validation succeeded   Next

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



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



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

If the difference is not acceptable, modify the fixed-point data types or your original algorithm. For more information, see “Testing Numerics” on page 21-84.

9 On the **Verification Output** tab, the app provides a link to a report that shows 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	0.0976295471191406	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.

Note Derived range analysis is not supported for non-scalar variables.

Prerequisites

This example requires the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	`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

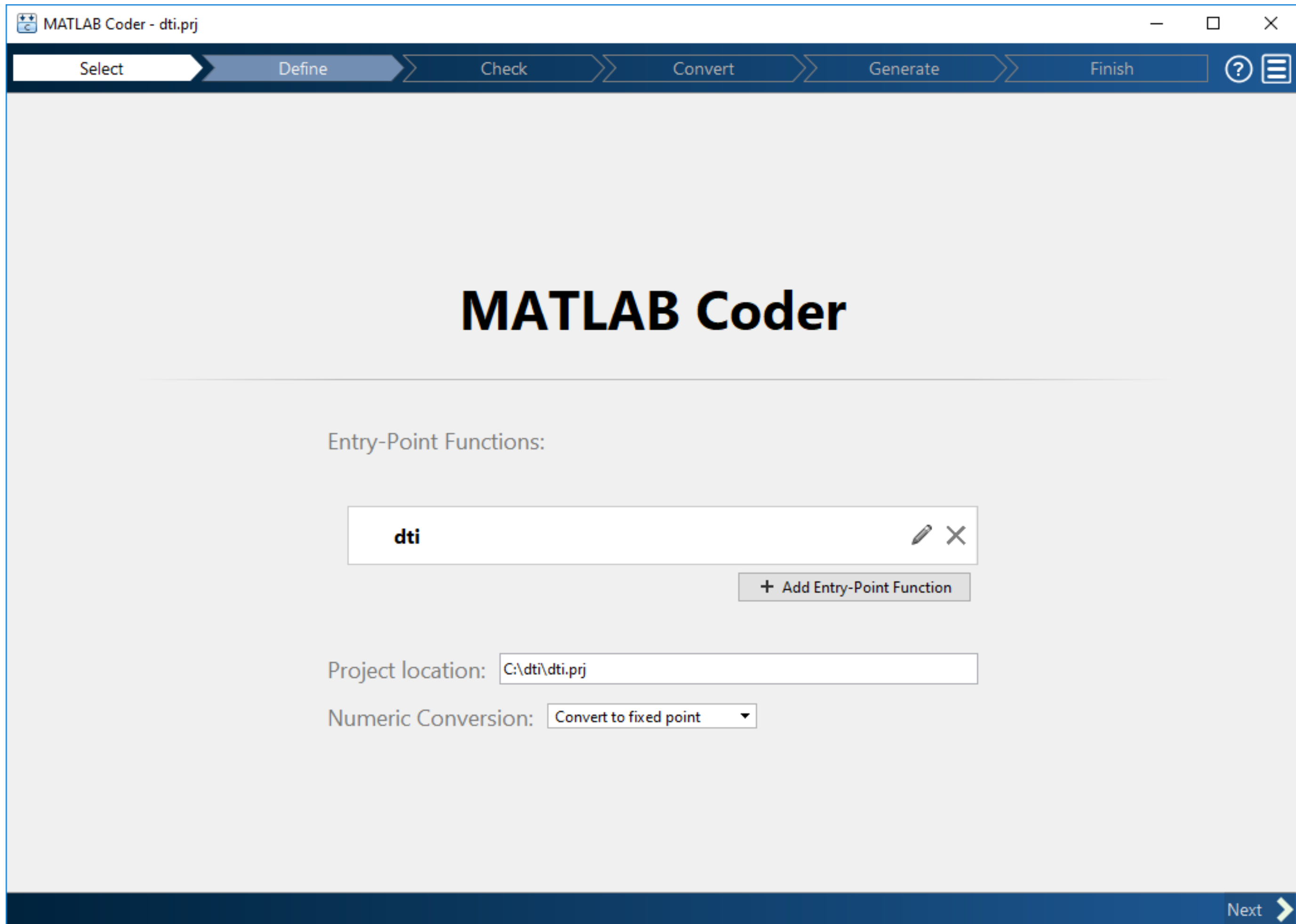
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

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`.



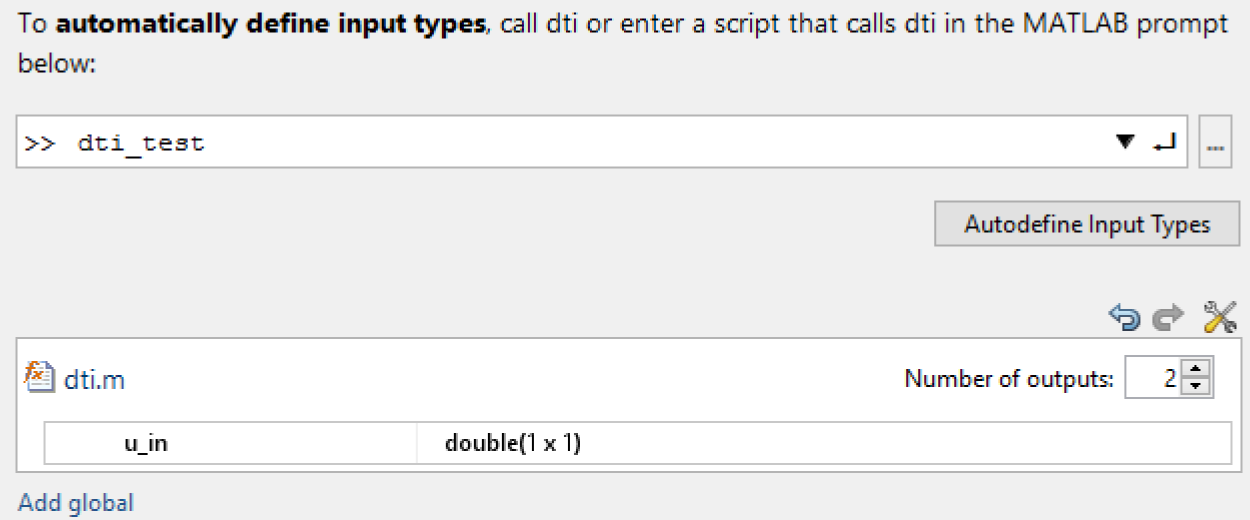
2 Click **Next** to go to the **Define Input Types** step.

   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)`.



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 21-64.

MATLAB Coder - dti.prj

Convert to Fixed Point — SETTINGS ▾ ANALYZE ▾ CONVERT TEST ▾

Source Code: dti

```
1  function [y, clip_status] = dti(u_in) %#codegen
2  % Discrete Time Integrator in MATLAB
3  %
4  % Forward Euler method, also known as Forward Rectangular, or left-hand
5  % approximation.  The resulting expression for the output of the block at
6  % step 'n' is y(n) = y(n-1) + K * u(n-1)
7  %
8  init_val = 1;
9  gain_val = 1;
10 limit_upper = 500;
11 limit_lower = -500;
12
13 % variable to hold state between consecutive calls to this block
14 persistent u_state;
15 if isempty(u_state)
16     u_state = init_val+1;
17 end
18
19 % Compute Output
20 if (u_state > limit_upper)
21     y = limit_upper;
22     clip_status = -2;
23 elseif (u_state >= limit_upper)
24     y = limit_upper;
```

Variables | Function Replacements | Output

Variable	Type	Sim Min	Sim Max	Whole ...	Proposed Type	Lo...	Max Diff
⊟ Input							
u_in	double			No			
⊟ Output							
y	double			No			
clip_status	double			No			
⊟ Persistent							
u_state	double			No			

Back Next

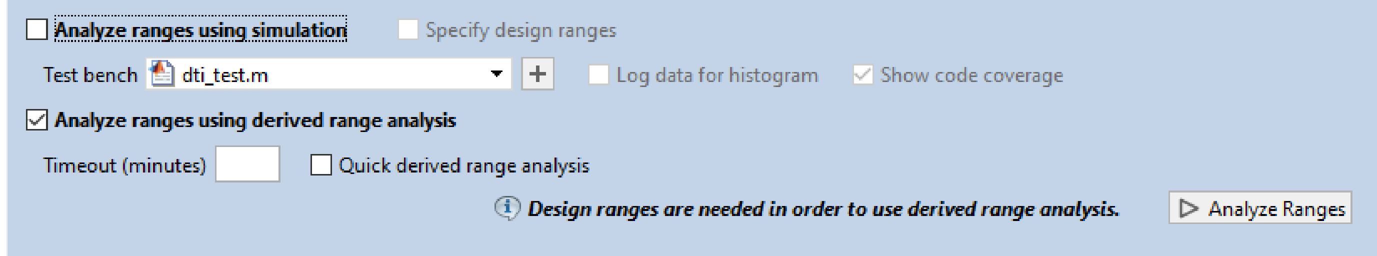
If functions are not supported for fixed-point conversion, the app displays them on the **Function Replacements** tab.

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.



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`.

```
1  function [y, clip_status] = dti(u_in) %#codegen
2  % Discrete Time Integrator in MATLAB
3  %
4  % Forward Euler method, also known as Forward Rectangular, or left-hand
5  % approximation.  The resulting expression for the output of the block at
6  % step 'n' is y(n) = y(n-1) + K * u(n-1)
7  %
8  init_val = 1;
9  gain_val = 1;
10 limit_upper = 500;
11 limit_lower = -500;
12
13 % variable to hold state between consecutive calls to this block
14 persistent u_state;
15 if isempty(u_state)
16     u_state = init_val+1;
```

Variables | Function Replacements | Output

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 21-71.

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.

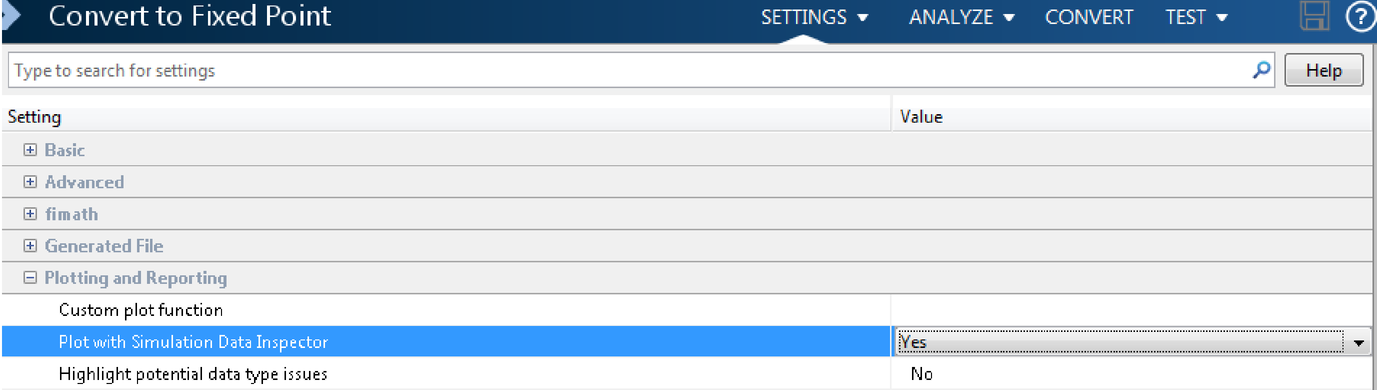
- `dti_fixpt_args.mat` — MAT-file containing a structure for the input arguments, a structure for the output arguments and the name of the fixed-point file.

If errors or warnings occur during validation, they show on the **Output** tab. See “Validating Types” on page 21-83.

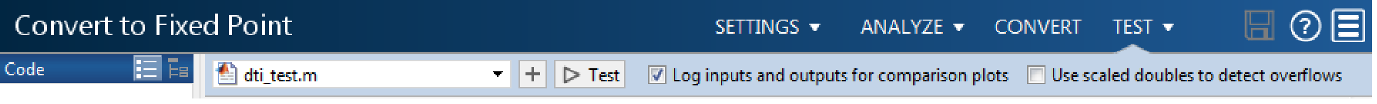
6. In the **Output Files** list, select `dti_fixpt.m`. The app displays the generated fixed-point code.
7. Use the Simulation Data Inspector to plot the floating-point and fixed-point results.

   a. Click the **Settings** arrow.

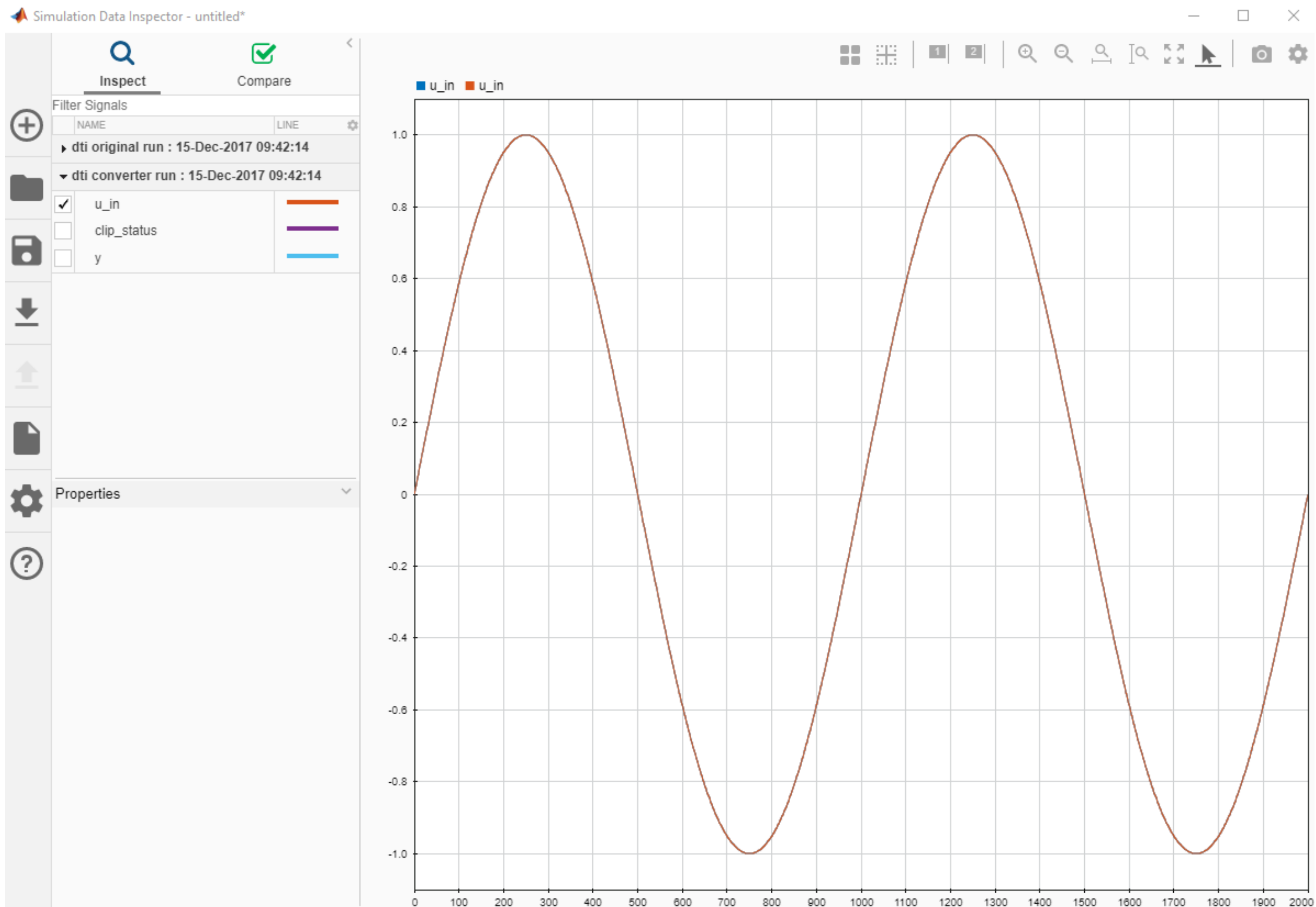
   b. Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to `Yes`.



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

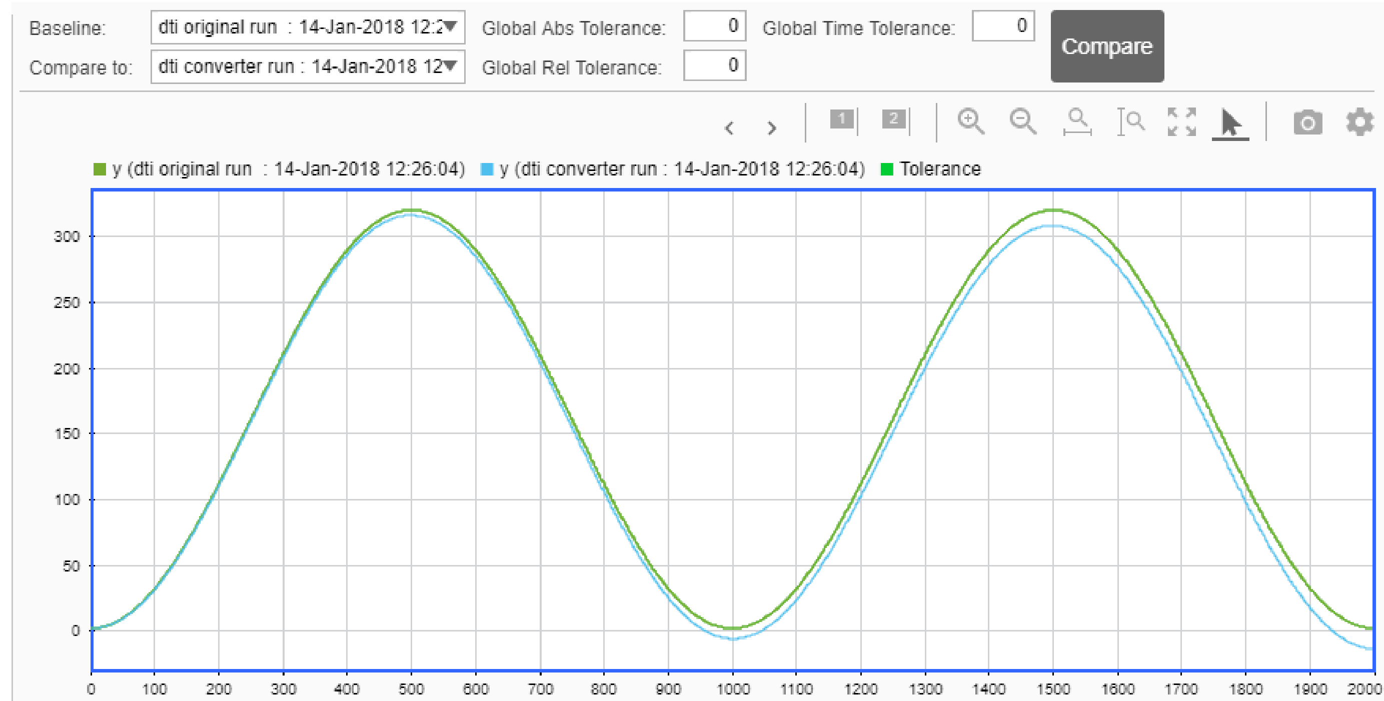


The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Optionally, you can add test files and select to run more than one test file to test numerics. The software runs both a floating-point and a fixed-point simulation and then calculates the errors for the output variable `y`. Because you selected to log inputs and outputs for comparison plots and to use the Simulation Data Inspector for these plots, the Simulation Data Inspector opens.



d You can use the Simulation Data Inspector to view floating-point and fixed-point run information and compare results. For example, to compare the floating-point and fixed-point values for the output `y`, select `y`. Click **Compare**. Set **Baseline** to the original run and **Compare to** to the converter run. 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.



8 On the **Verification Output** tab, the app provides a link to the Fixed_Point Report.

Variables | Function Replacements | Output | Verification Output

```
-------------- Output variable : clip_status --------------
Generating comparison plot...

-------------- Output variable : y --------------
Generating comparison plot...

### Generating Fixed-point Types Report for 'dti_fixpt' dti_fixpt_report.html
### Elapsed Time:             25.0139 sec(s)
```

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

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.
	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.

Advanced Type Proposal Settings	Values	Description
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.
Safety margin for sim min/max (%)	0 (default)	Specify safety factor for simulation minimum and maximum values. The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of `55` specifies that you want a range at least 55 percent larger. A value of `-15` specifies that a range up to 15 percent smaller is acceptable.
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 bias, and then rerun the verification.
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	
	Wrap (default)	
Product mode	FullPrecision (default)	
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	
	KeepLSB	

fimath Settings	Values	Description
	KeepMSB	For more information on `fimath` properties, see “fimath Object Properties” (Fixed-Point Designer).
	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-precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.
	Yes	

Detect Overflows

This example shows how to detect overflows using the MATLAB Coder app. At the numerical testing stage in the conversion process, you choose to simulate the fixed-point code using scaled doubles. The app then reports which expressions in the generated code produce values that overflow the fixed-point data type.

Prerequisites

This example requires the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `overflow.m` and `overflow_test.m` files to your local working folder.

Type	Name	Description
Function code	`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

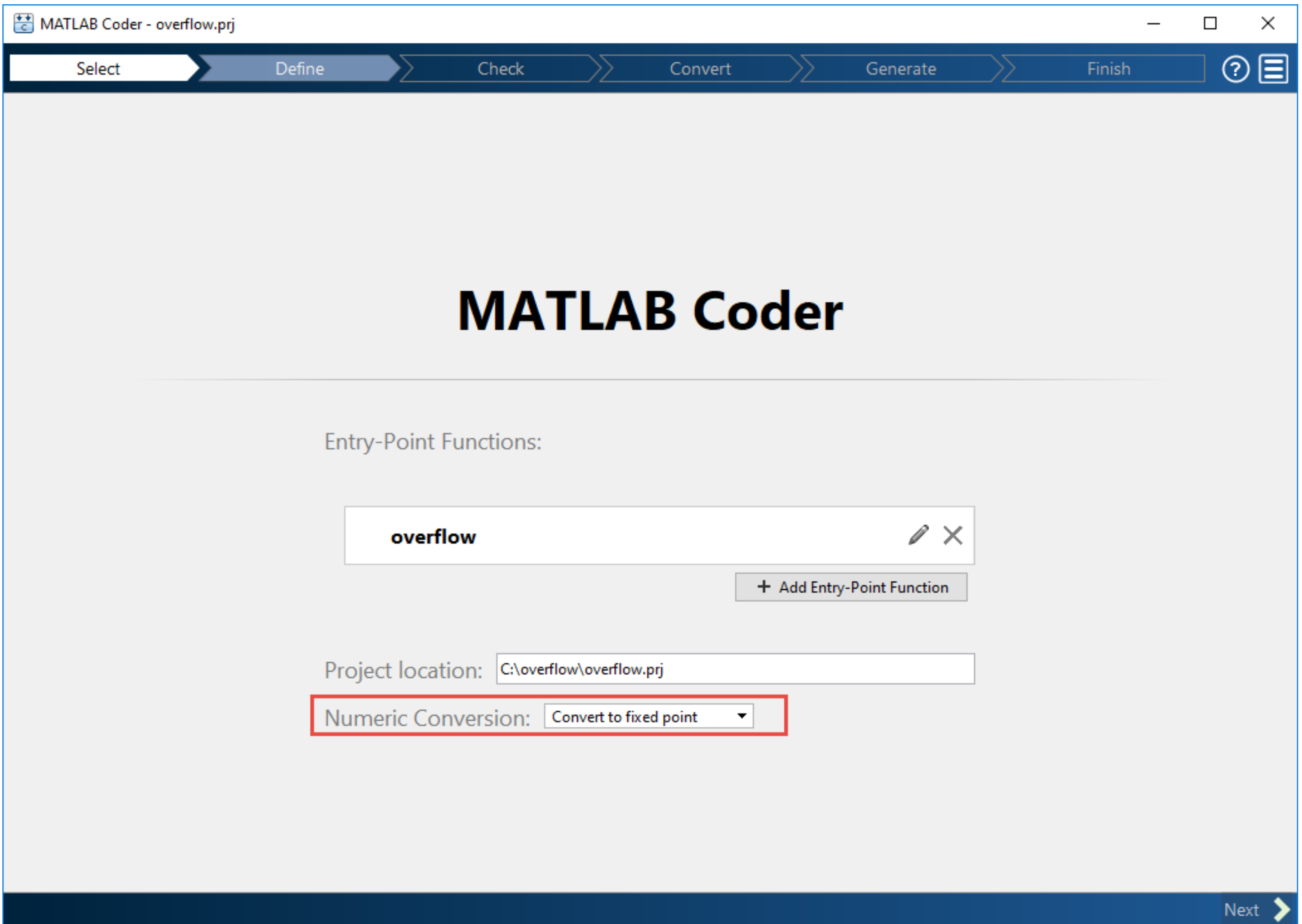
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

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`.



2 Click **Next** to go to the **Define Input Types** step.

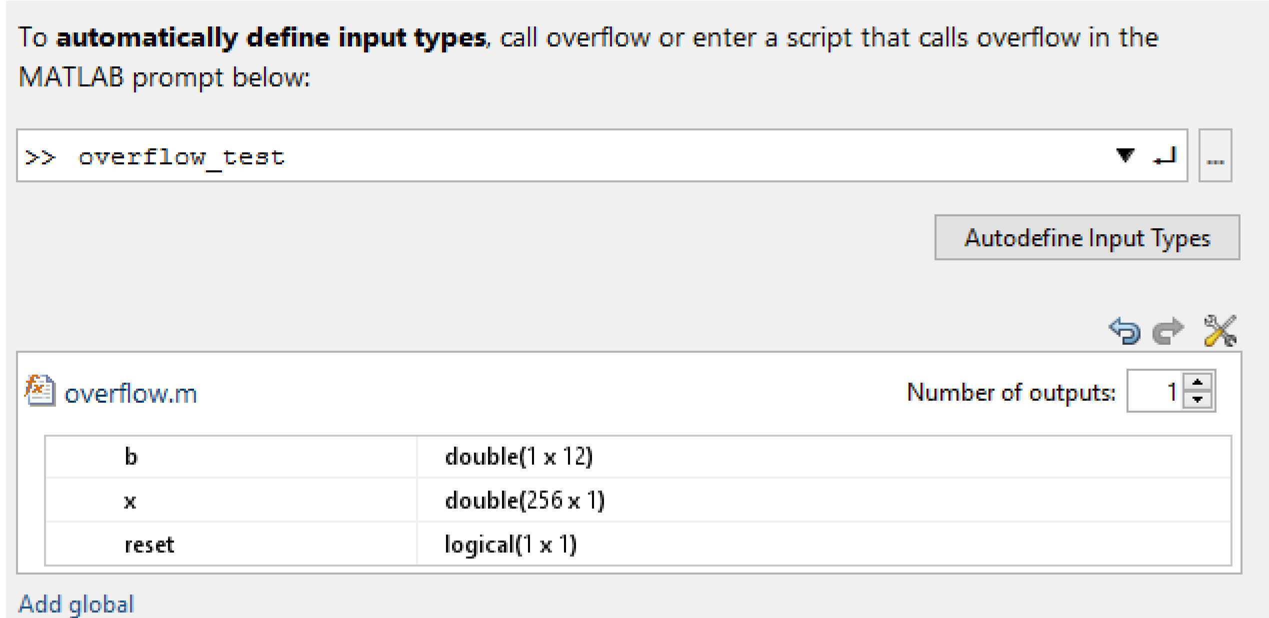
The app screens `overflow.m` for code violations and code generation readiness issues. The app does not find issues in `overflow.m`.

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)`.



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

Check for Run-Time Issues

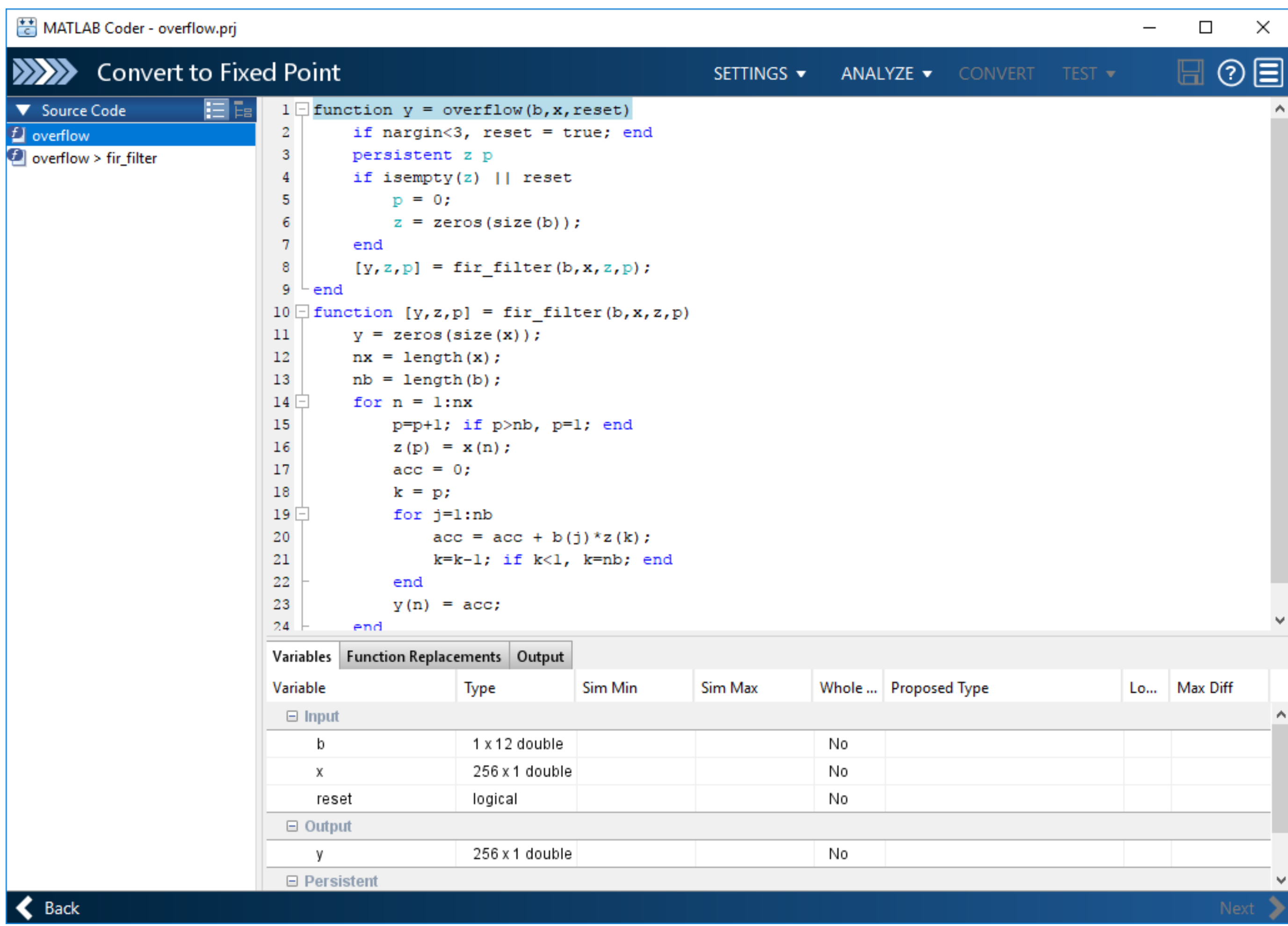
The **Check for Run-Time Issues** step generates instrumented MEX. It runs the test file `overflow_test` replacing calls to `overflow` with calls to the generated MEX function. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code in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

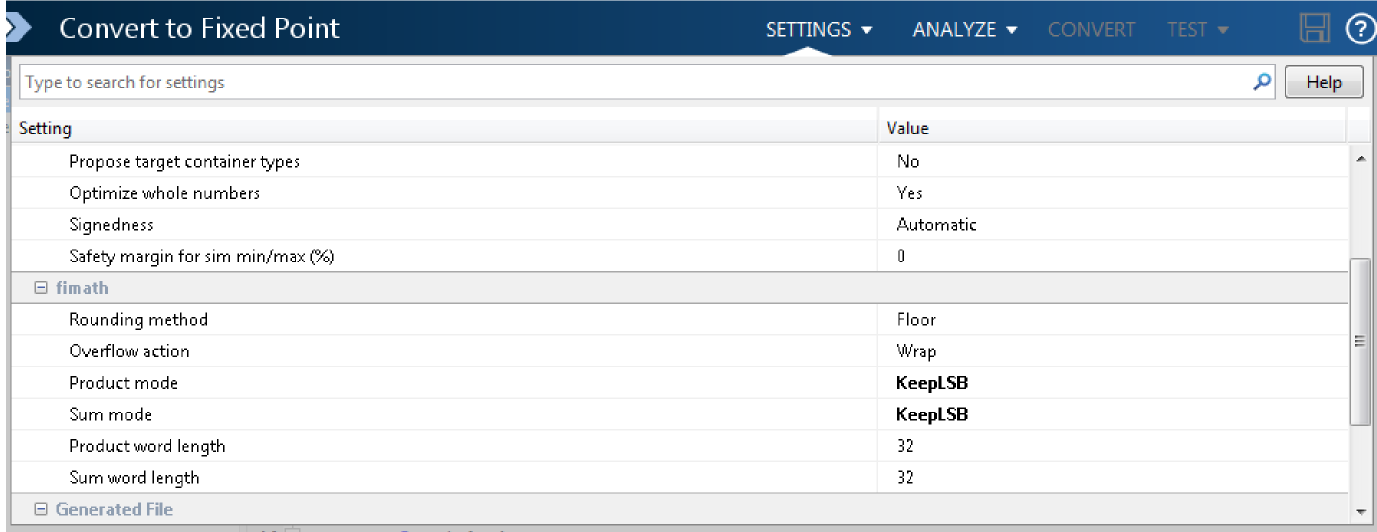
Convert to Fixed Point

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 21-64.



On the **Function Replacements** tab the app displays functions that are not supported for fixed-point conversion. See “Running a Simulation” on page 21-70.

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.



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.

Variables | Function Replacements | Output

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 21-83.

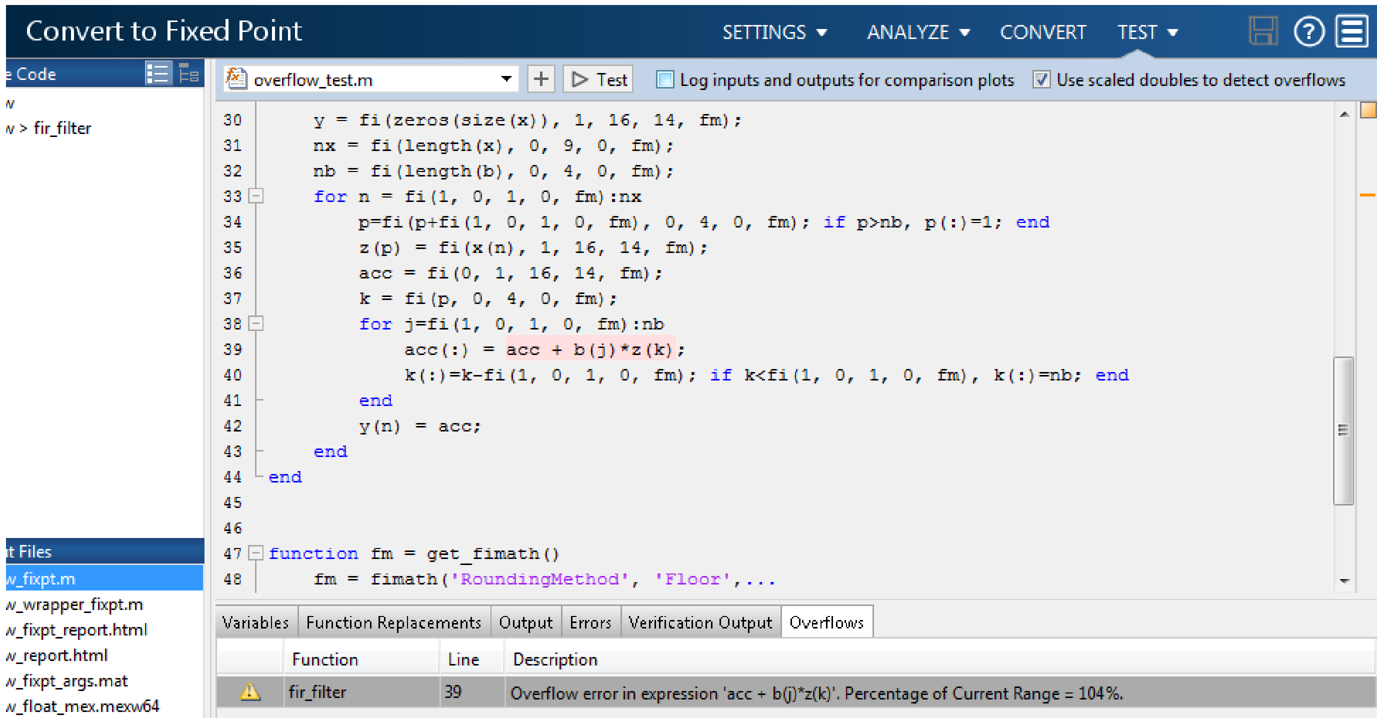
Test Numerics and Check for Overflows

1 Click the **Test** arrow ▼. Verify that the test file is `overflow_test.m`. Select **Use scaled doubles to detect overflows**, and then click **Test**.

The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to detect overflows, it also runs the simulation using scaled double versions

of the proposed fixed-point types. Scaled doubles store their data in double-precision floating-point, so they carry out arithmetic in full range. Because they retain their fixed-point settings, they can report when a computation goes out of the range of the fixed-point type.

The simulation runs. The app detects an overflow. The app reports the overflow on the **Overflow** tab. To highlight the expression that overflowed, click the overflow.



2 Determine whether it was the sum or the multiplication that overflowed.

In the **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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create Algorithm and Test Files

1. Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

2. Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
end
plot( x, y );
```

Open the MATLAB Coder App

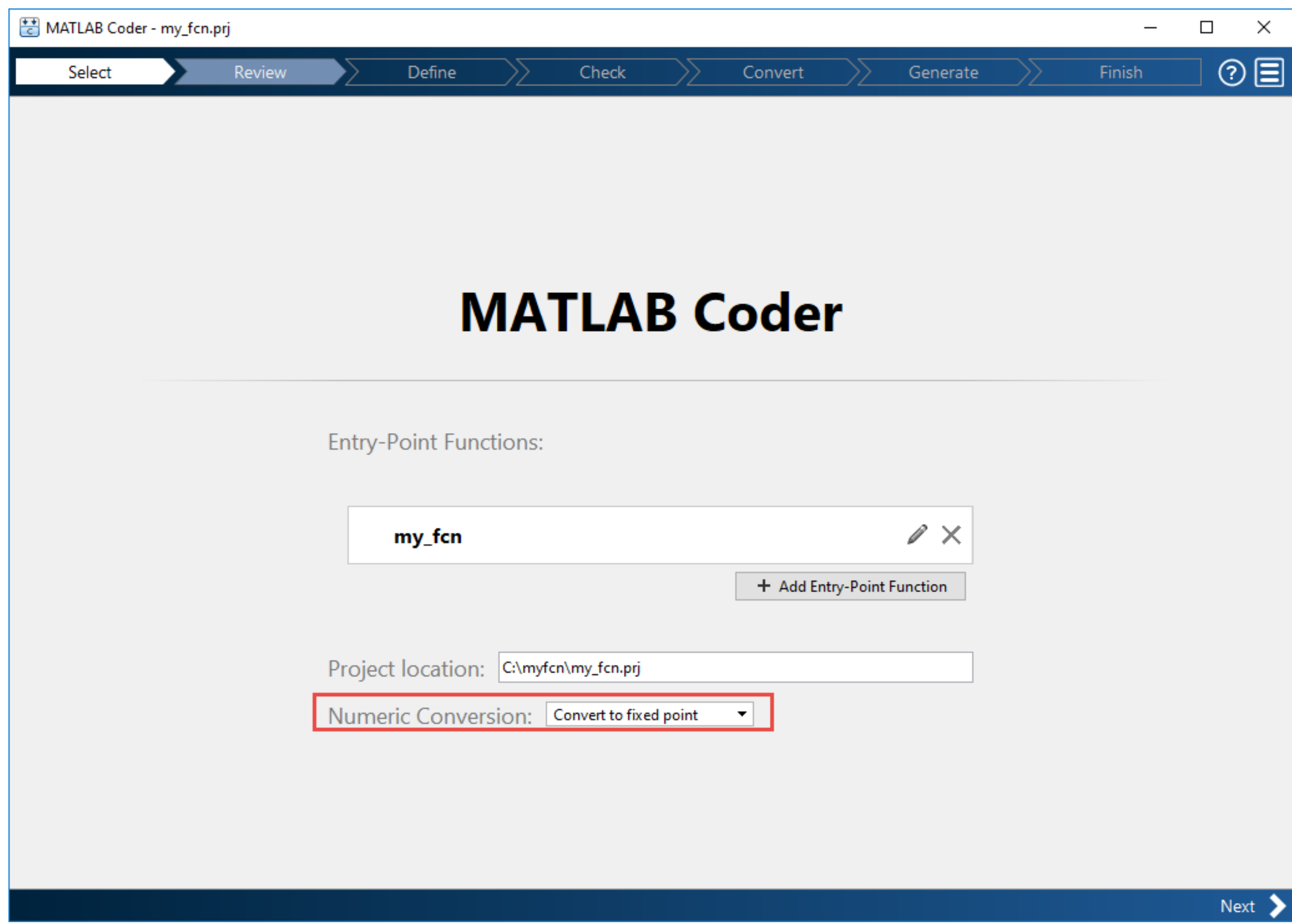
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

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`.

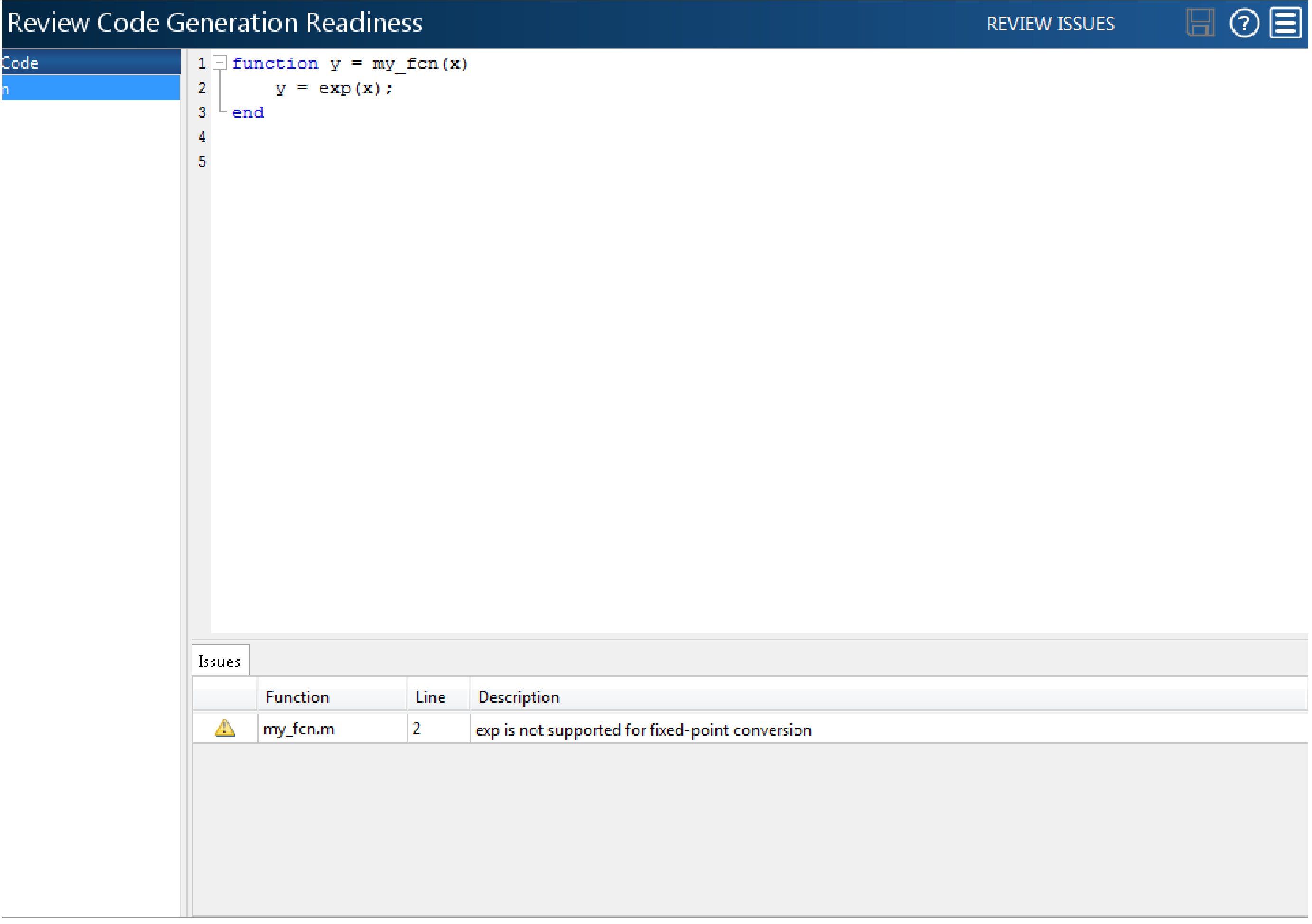


2 Click **Next** to go to the **Define Input Types** step.

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.



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**.

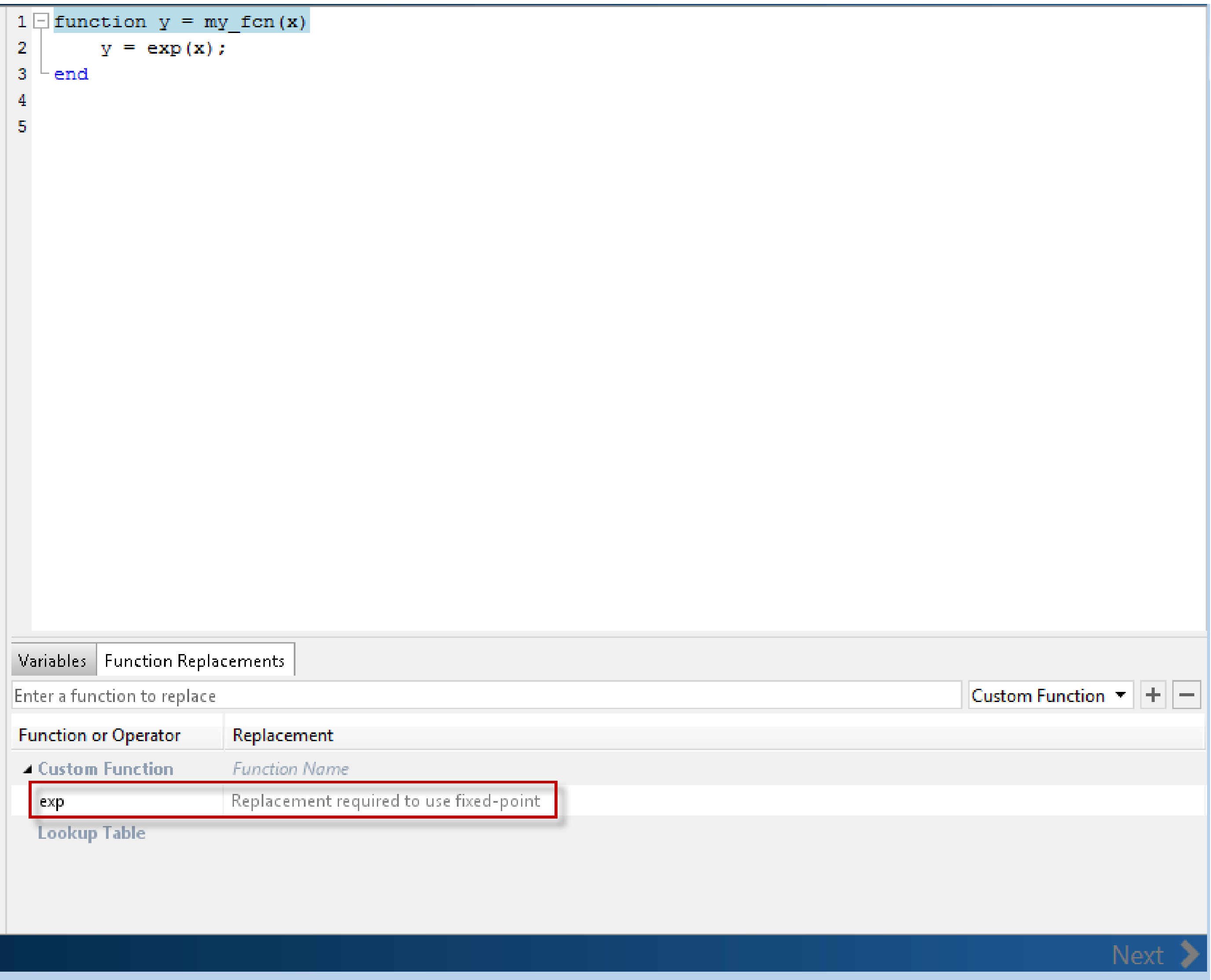
   The app does not detect issues.

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

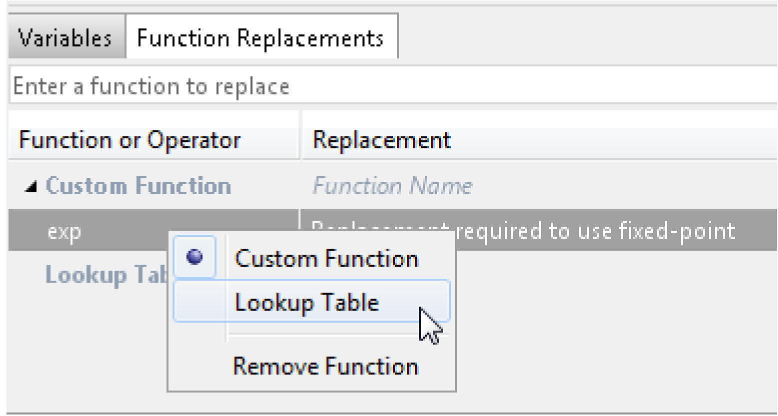
Replace exp Function with Lookup Table

1 Select the **Function Replacements** tab.

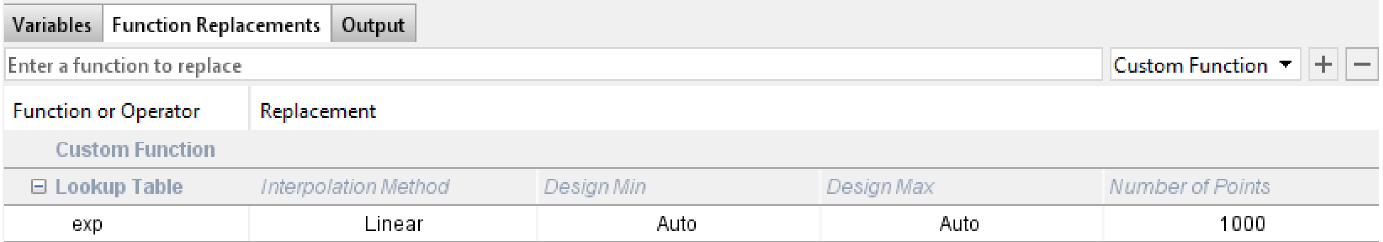
The app indicates that you must replace the `exp` function.



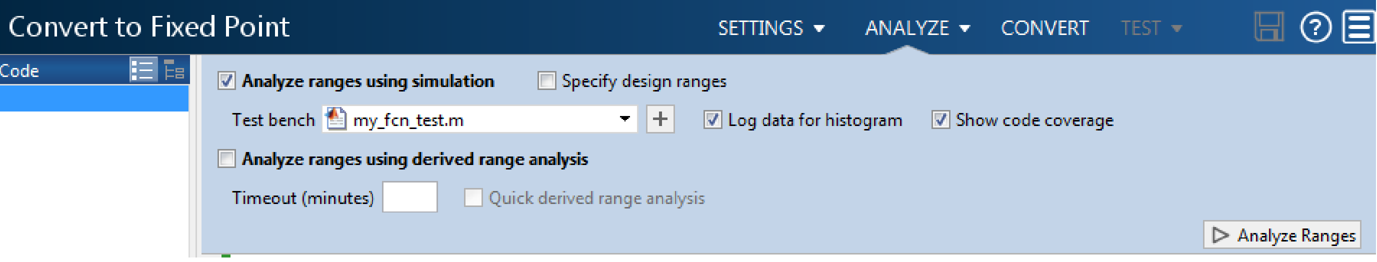
2 On the **Function Replacements** tab, right-click the `exp` function and select `Lookup Table`.



The app moves the `exp` function to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. **Design Min** and **Design Max** are set to `Auto` which means that the app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.



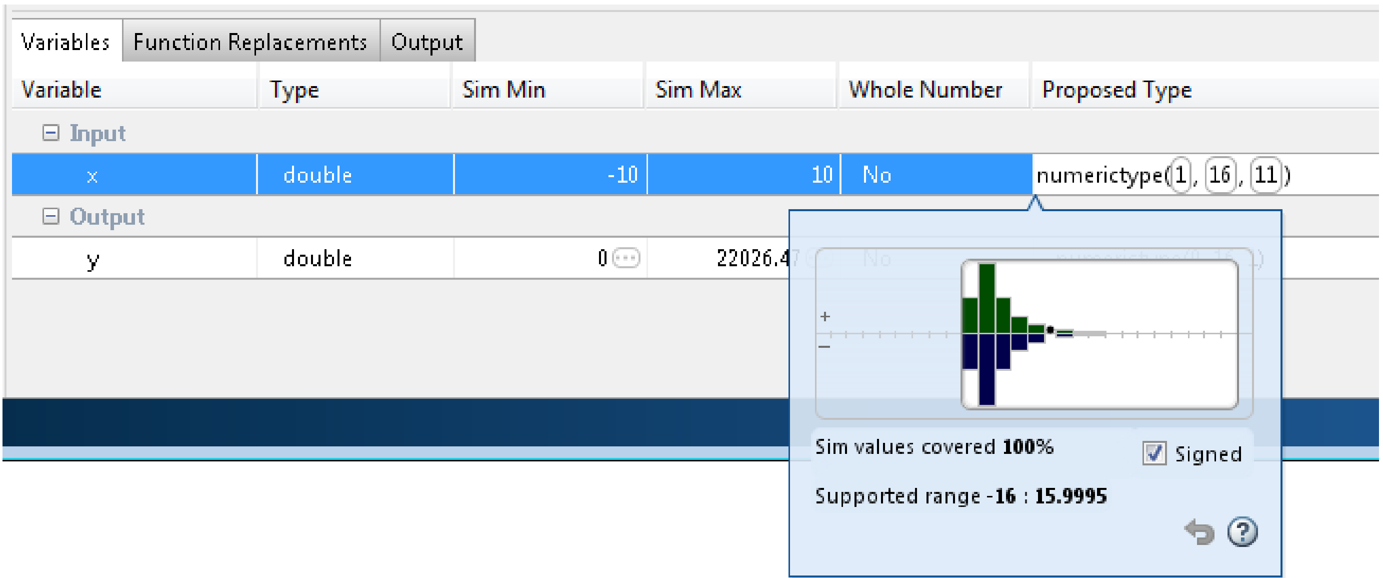
3 Click the **Analyze** arrow, select **Log data for histogram**, and verify that the test file is `my_fcn_test`.



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.



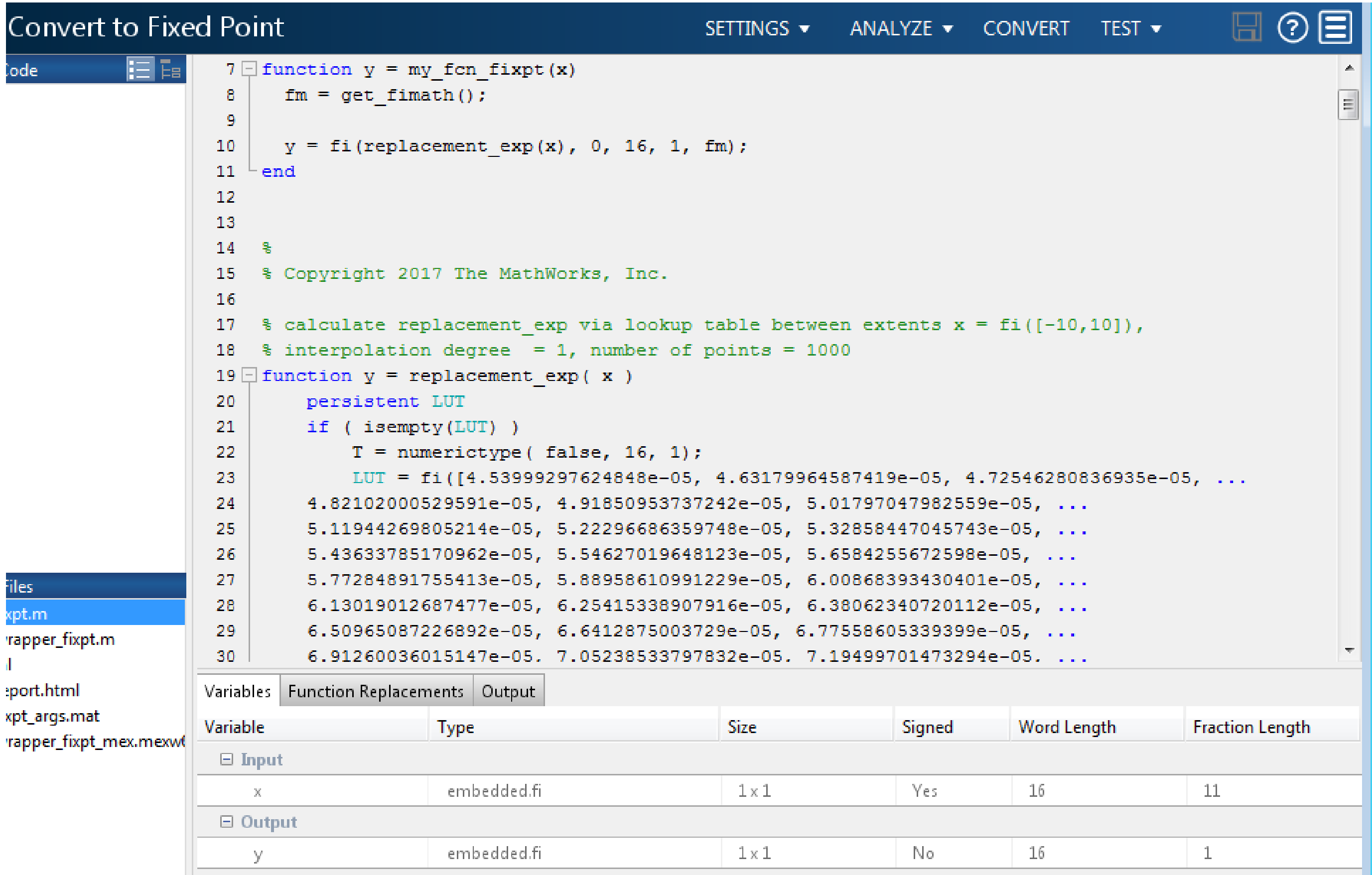
Convert to Fixed Point

1 Click **Convert**.

   The app validates the proposed types, and generates a fixed-point version of the entry-point function, `my_fcn_fixpt.m`.

2 In the Output Files list, select `my_fcn_fixpt.m`.

   The conversion process generates a lookup table approximation, `replacement_exp`, for the `exp` function.



The 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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create Algorithm and Test Files

In a local, writable folder:

1. Create a MATLAB function, `custom_fcn.m` which is the function that you want to replace.

   ```
   function y = custom_fcn(x)
       y = 1./(1+exp(-x));
   end
   ```
2. Create a wrapper function, `call_custom_fcn.m`, that calls `custom_fcn.m`.

   ```
   function y = call_custom_fcn(x)
       y = custom_fcn(x);
   end
   ```
3. Create a test file, `custom_test.m`, that uses `call_custom_fcn`.

   ```
   close all
   clear all

   x = linspace(-10,10,1e3);
   for itr = 1e3:-1:1
      y(itr) = call_custom_fcn( x(itr) );
   end
   plot( x, y );
   ```

Open the MATLAB Coder App

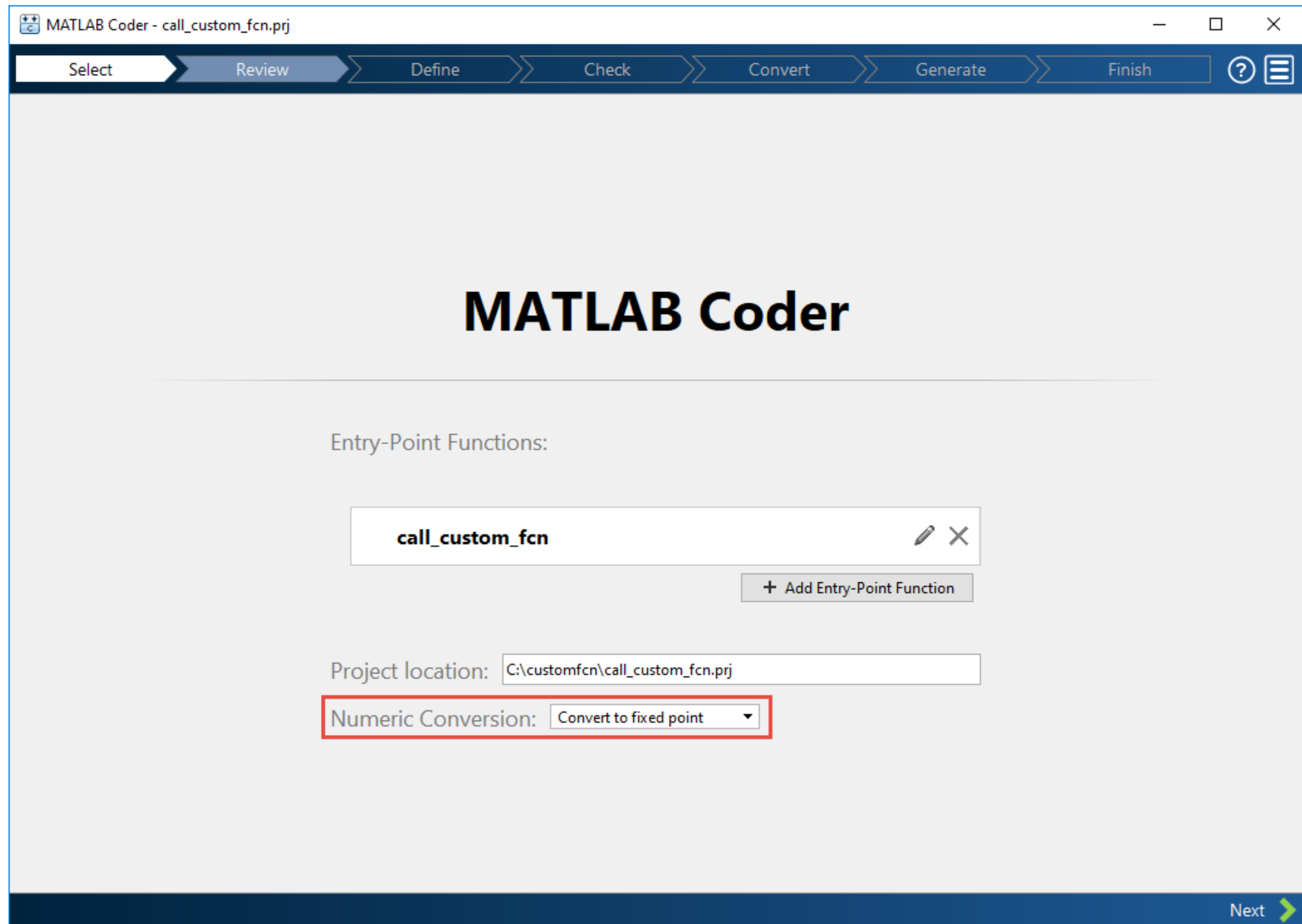
1. Navigate to the work folder that contains the file for this example.
2. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.

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`.

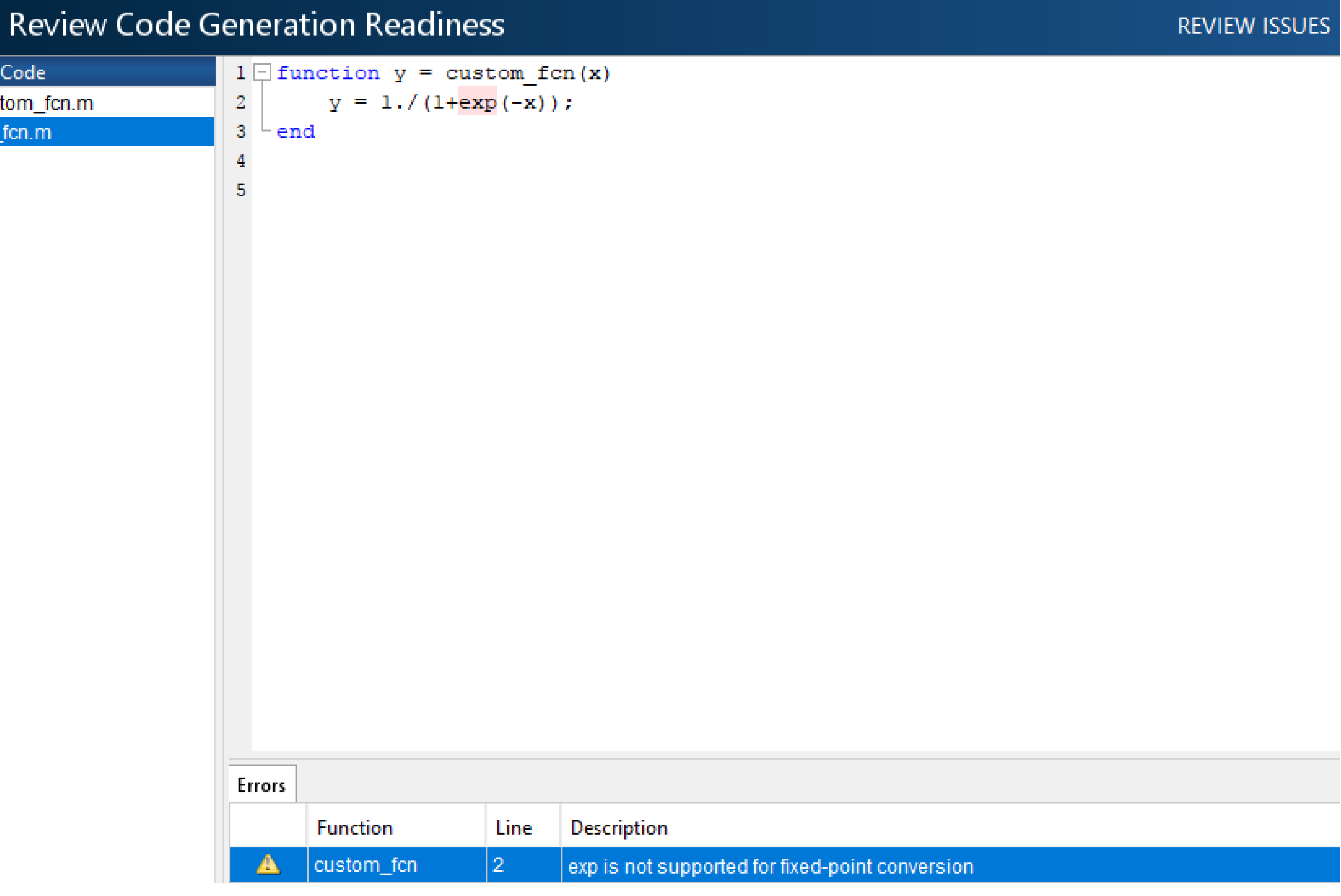


2 Click **Next** to go to the **Define Input Types** step.

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`.



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**.

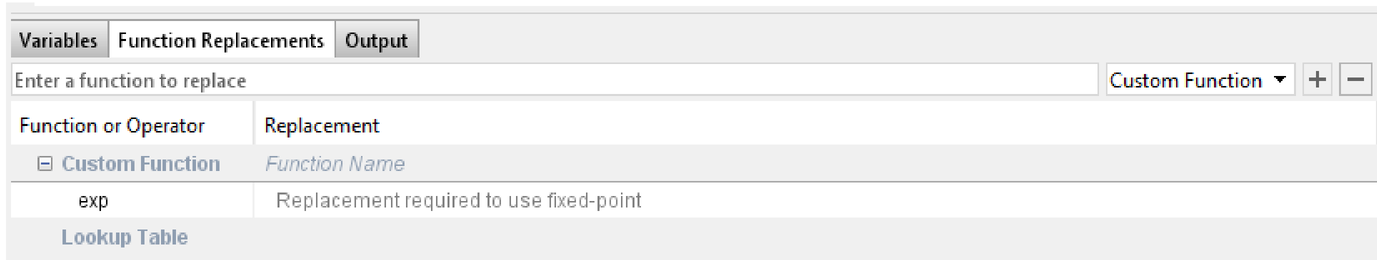
   The app does not detect issues.

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

Replace custom_fcn with Lookup Table

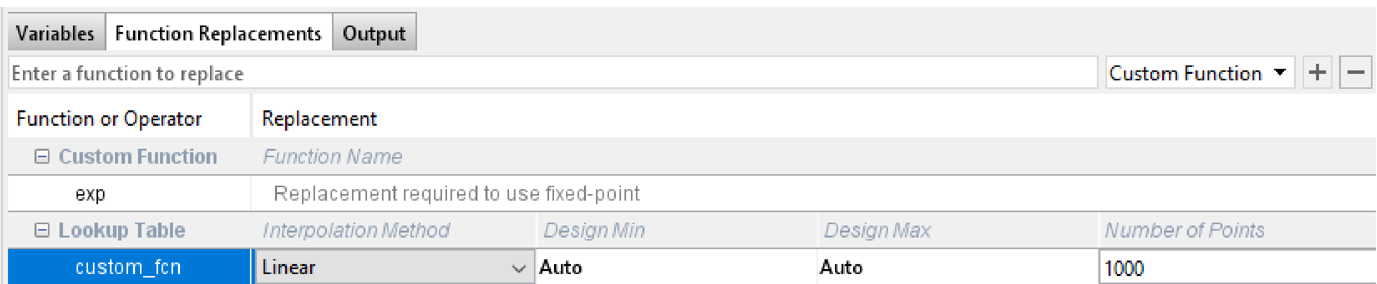
1. Select the **Function Replacements** tab.

   The app indicates that you must replace the `exp` function.

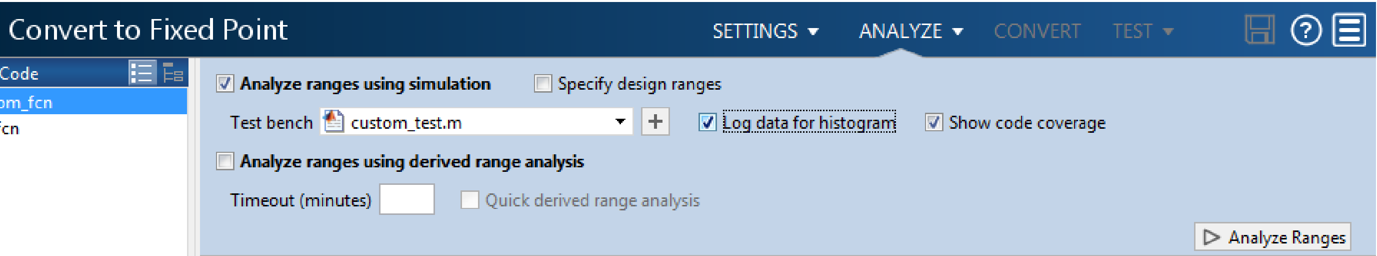


2. Enter the name of the function to replace, `custom_fcn`, select `Lookup Table`, and then click +.

   The app adds `custom_fcn` to the list of functions that it will replace with a Lookup Table. By default, the lookup table uses linear interpolation and 1000 points. The app sets **Design Min** and **Design Max** to `Auto` which means that app uses the design minimum and maximum values that it detects by either running a simulation or computing derived ranges.



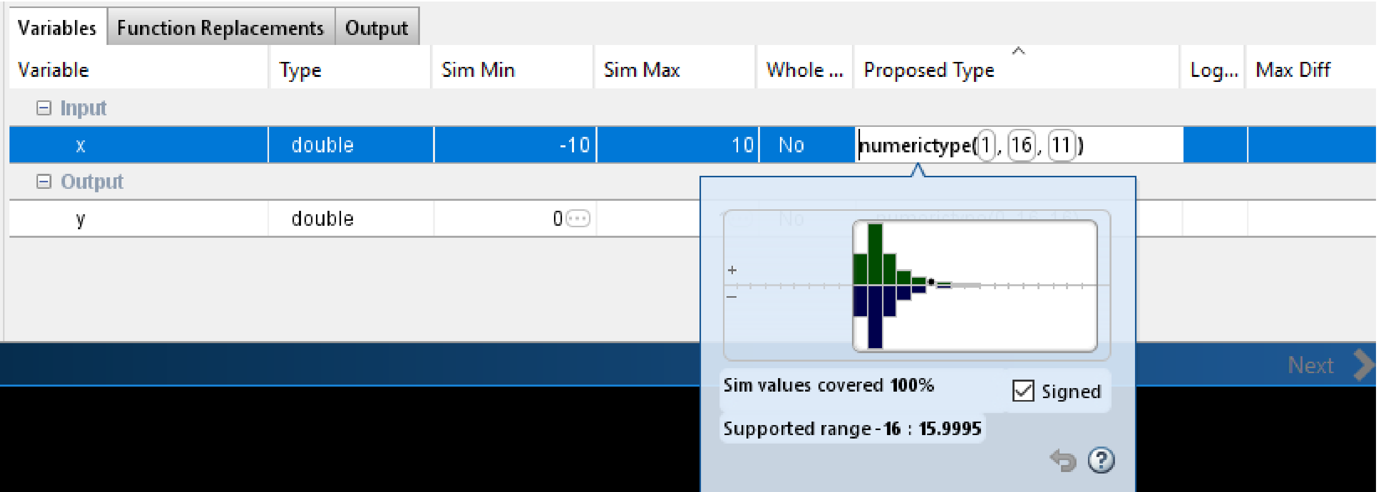
3. Click the **Analyze** arrow, select **Log data for histogram**, and verify that the test file is `call_custom_test`.



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.



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`.

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

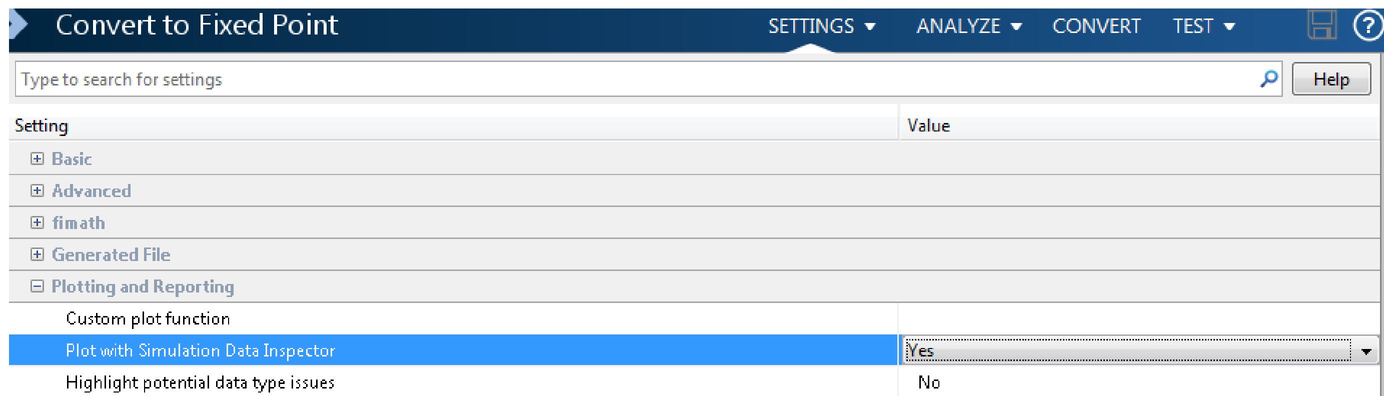
of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

Enable Plotting Using the Simulation Data Inspector

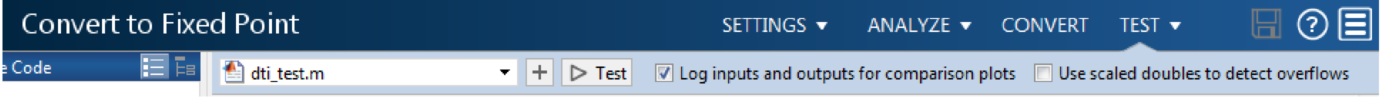
You can use the Simulation Data Inspector (Simulink) with the MATLAB Coder app to inspect and compare floating-point and fixed-point logged input and output data.

1 On the **Convert to Fixed Point** page,

   Click the **Settings** arrow.
2 Expand the **Plotting and Reporting** settings and set **Plot with Simulation Data Inspector** to `Yes`.



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



For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges” on page 21-17“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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	`myFilter.m`	Entry-point MATLAB function
Test file	`myFilterTest.m`	MATLAB script that tests `myFilter.m`
Plotting function	`plotDiff.m`	Custom plot function
MAT-file	`filterData.mat`	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
```

```
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
 y = myFilter(d(idx));
end
```

The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%           i) name
%           ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after Fixed-Point conve
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % 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.

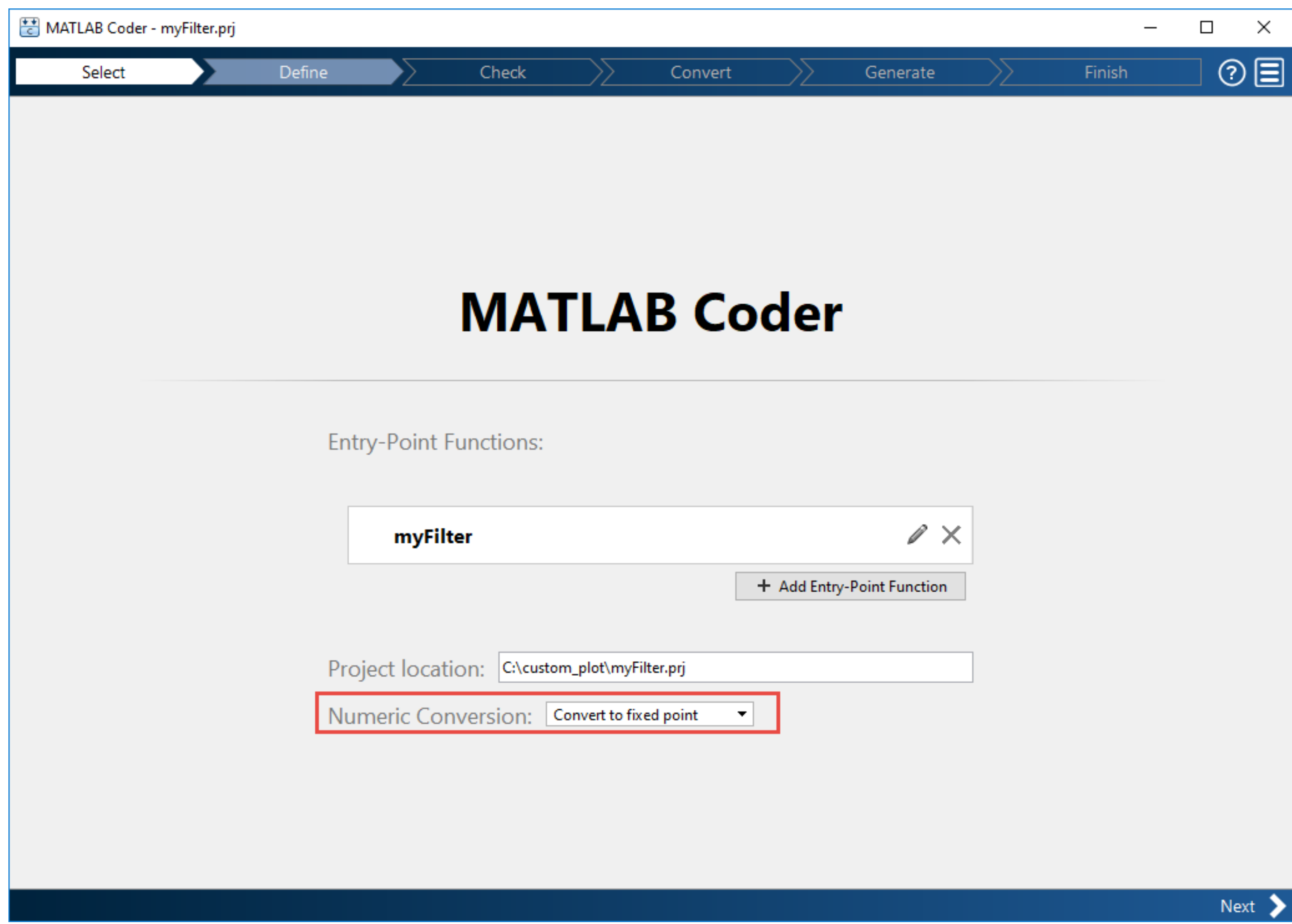
Select Source Files

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`.



2 Click **Next** to go to the **Define Input Types** step.

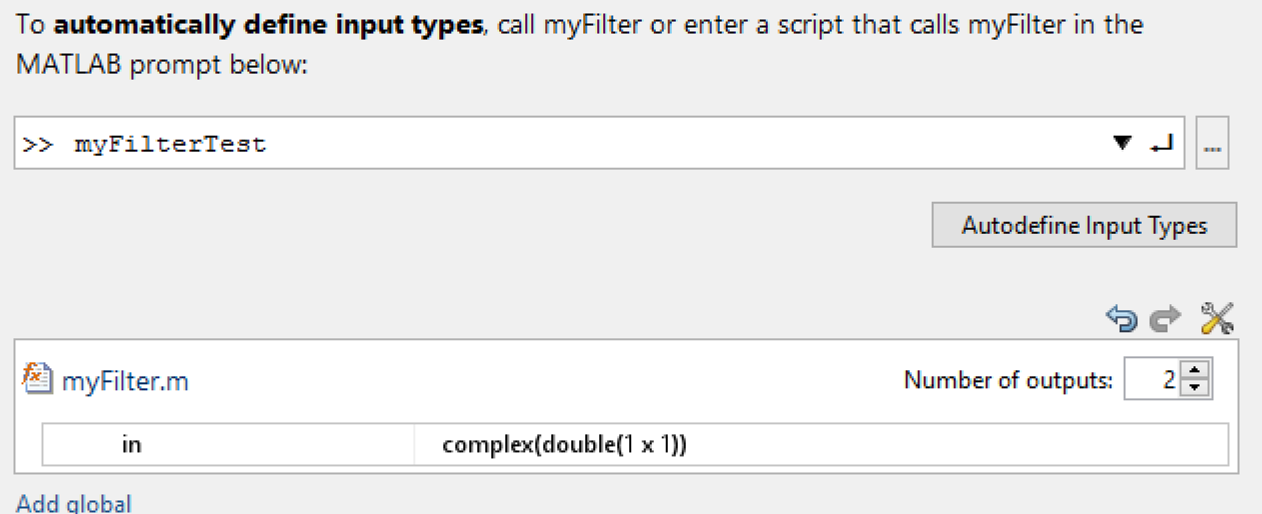
The app screens `myFilter.m` for code violations and code generation readiness issues. The app does not find issues in `myFilter.m`.

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))`.



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

Check for Run-Time Issues

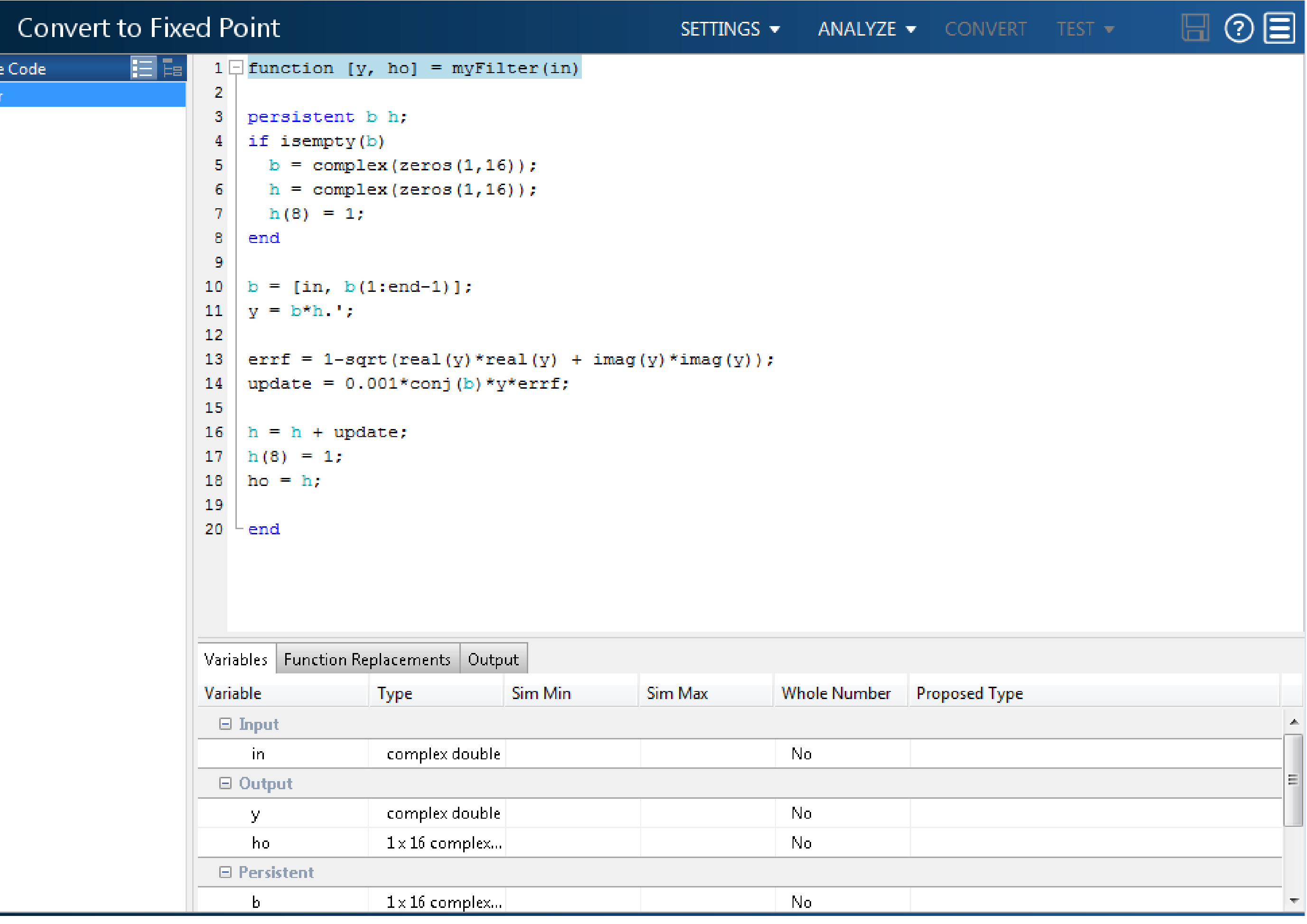
The **Check for Run-Time Issues** step generates instrumented MEX. `myFilter`. It runs the test file `myFilterTest` replacing calls to `myFilter` with calls to the generated MEX. If the app finds issues, it provides warning and error messages. You can click a message to highlight the problematic code 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 21-64"View and Modify Variable Information" (Fixed-Point Designer).



2 To open the settings dialog box, click the **Settings** arrow .

a Verify that **Default word length** is set to `16`.

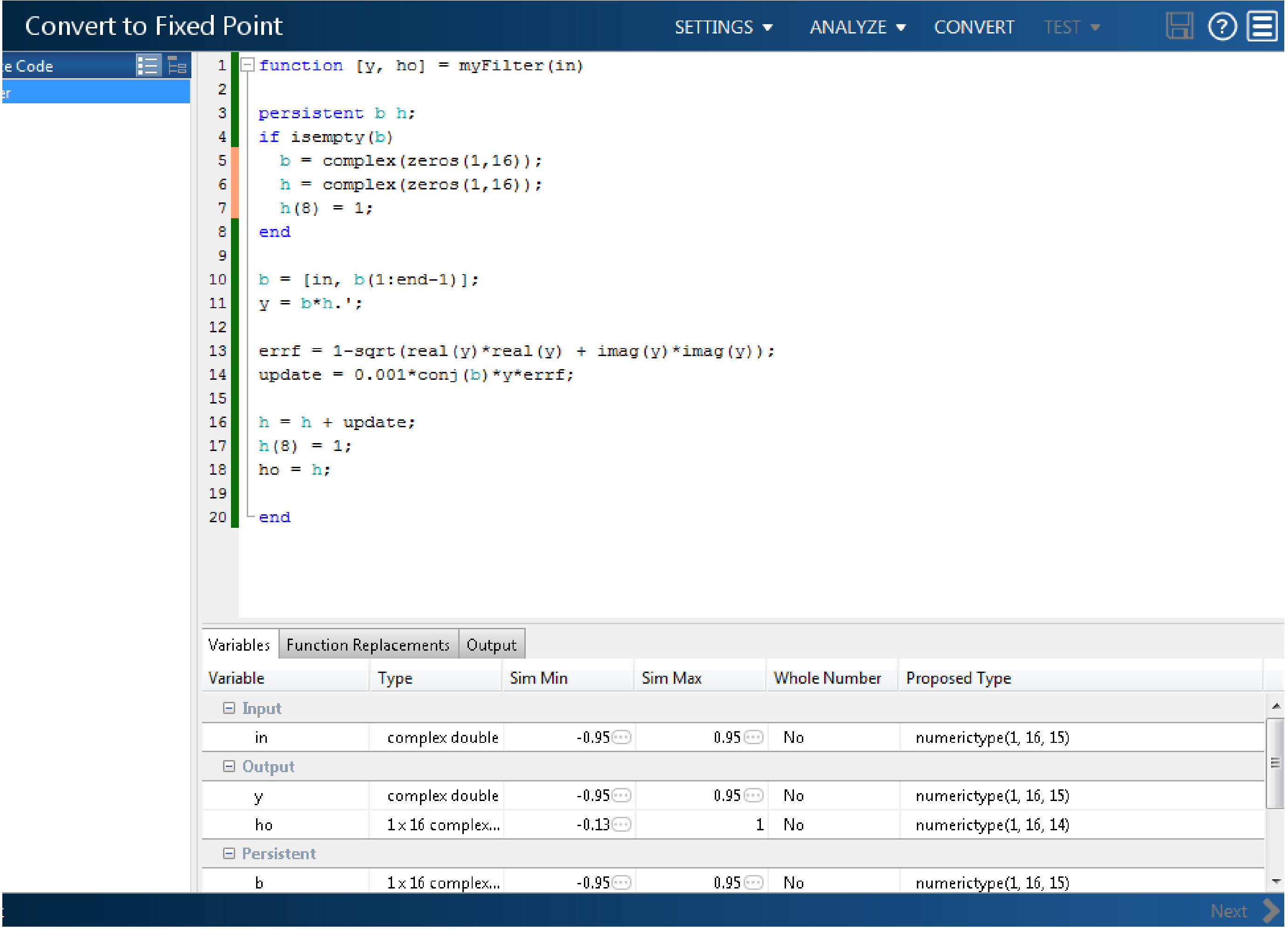
b Under **Advanced**, set **Signedness** to `Signed`

c Under **Plotting and Reporting**, set **Custom plot function** to `plotDiff`.

3 Click the **Analyze** arrow . Verify that the test file is `myFilterTest`.

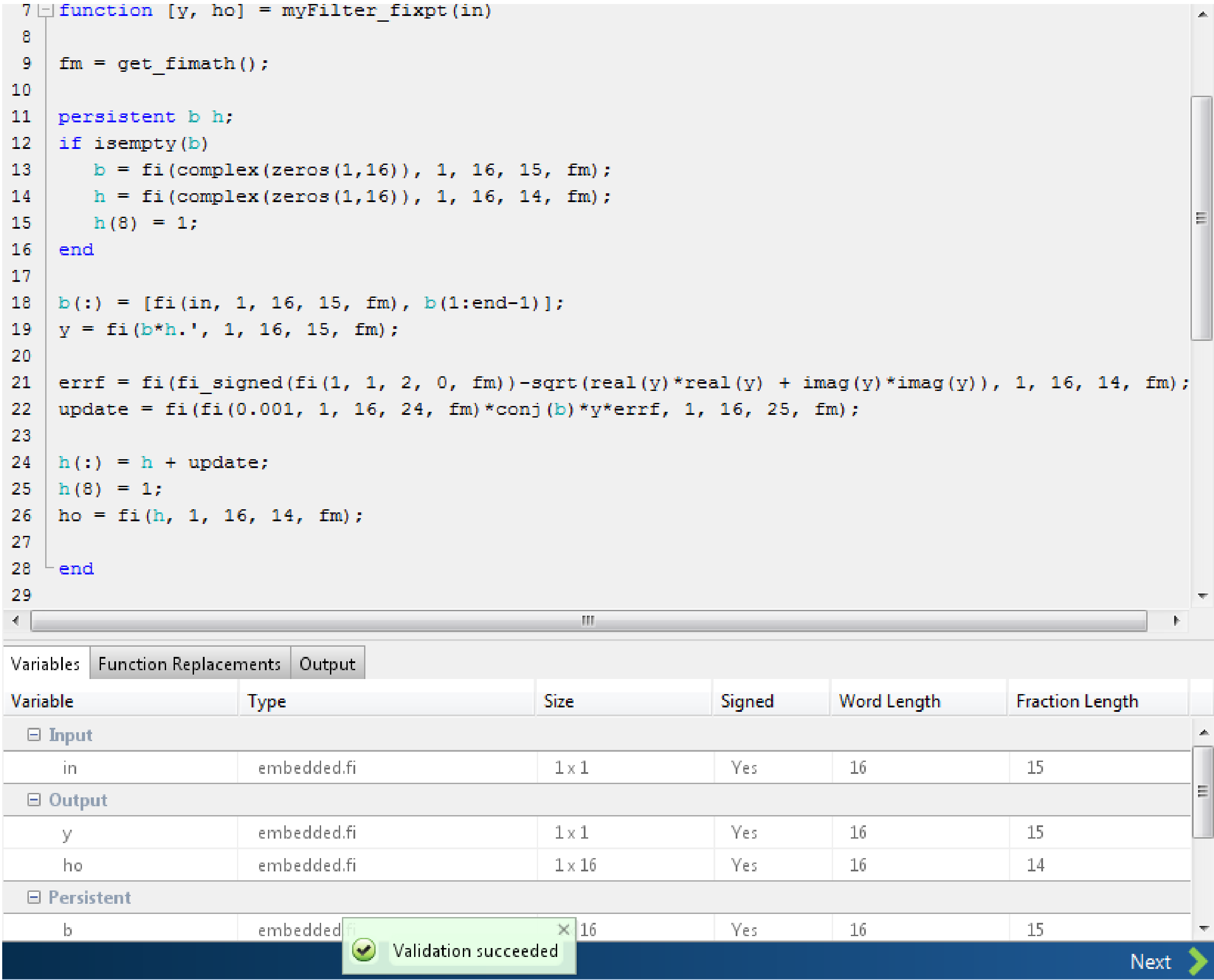
4 Click **Analyze**.

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.



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

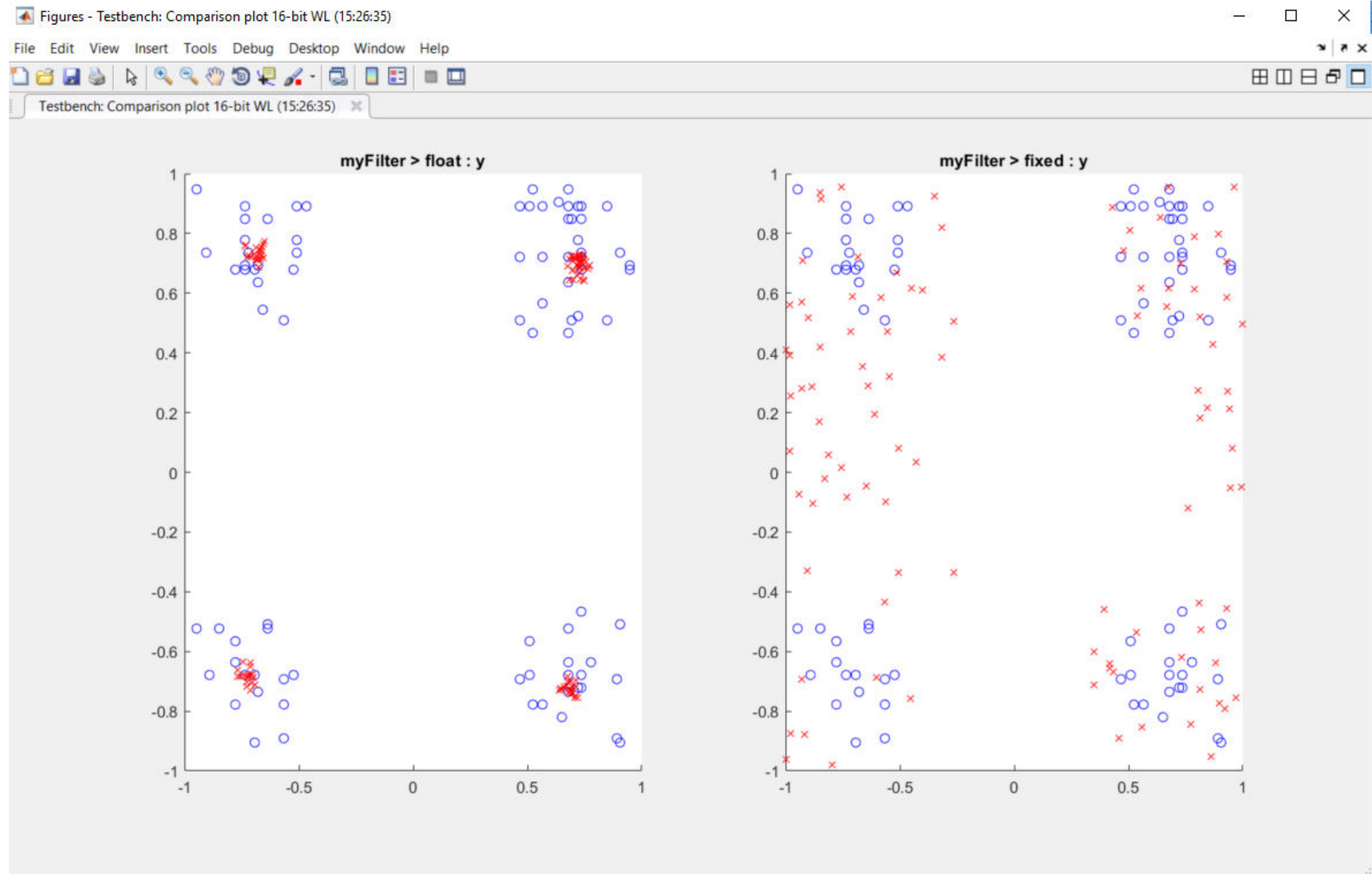
The software validates the proposed types and generates a fixed-point version of the entry-point function.



Test Numerics and View Comparison Plots

1. Click **Test** arrow, select **Log inputs and outputs for comparison plots**, and then click **Test**.

   The app runs the test file that you used to define input types to test the fixed-point MATLAB code. Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the app uses this function to generate the comparison plot. The plot shows that the fixed-point results do not closely match the floating-point results.

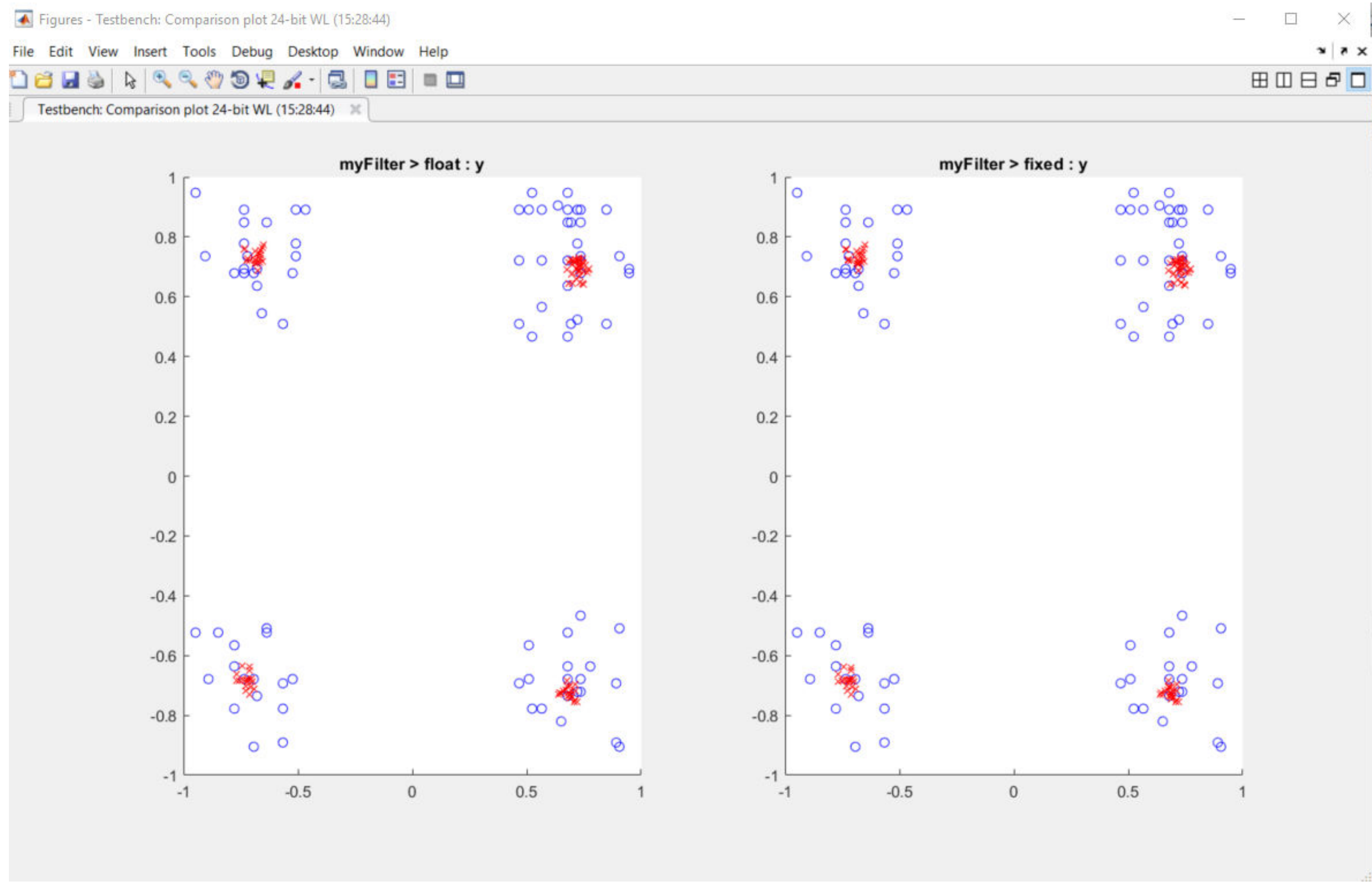


2 In the settings, increase the **DefaultWordLength** to 24 and then convert to fixed point again.

The app converts `myFilter.m` to fixed point and proposes fixed-point data types using the new default word length.

3 Run the test numerics step again.

The increased word length improves the results. This time, the plot shows that the fixed-point results match the floating-point results.


Figures - Testbench: Comparison plot 24-bit WL (15:28:44)
File Edit View Insert Tools Debug Desktop Window Help
Testbench: Comparison plot 24-bit WL (15:28:44)
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
-0.5
0.5

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 for the function that you selected in the **Source Code** pane, use the **Variables** tab or pause over a variable in the code window. For more information, see "Viewing Variables" on page 21-79.

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`.

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 21-65.

  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 21-65.

  Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.
- **Whole Number**

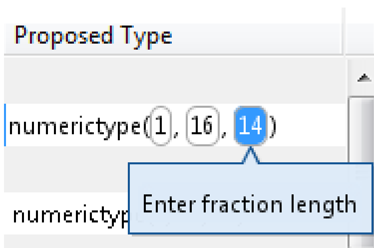
  The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

  Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.

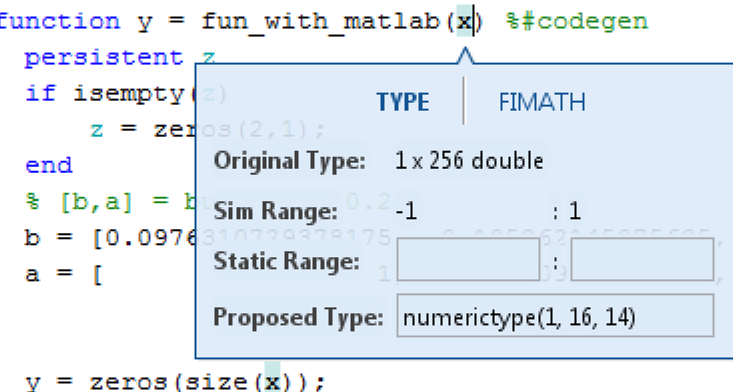
- **Proposed Type**

  You can modify the signedness, word length, and fraction length settings individually:

  - On the **Variables** tab, modify the value in the **ProposedType** field.

  - In the code window, select a variable, and then modify the **Proposed Type** field.

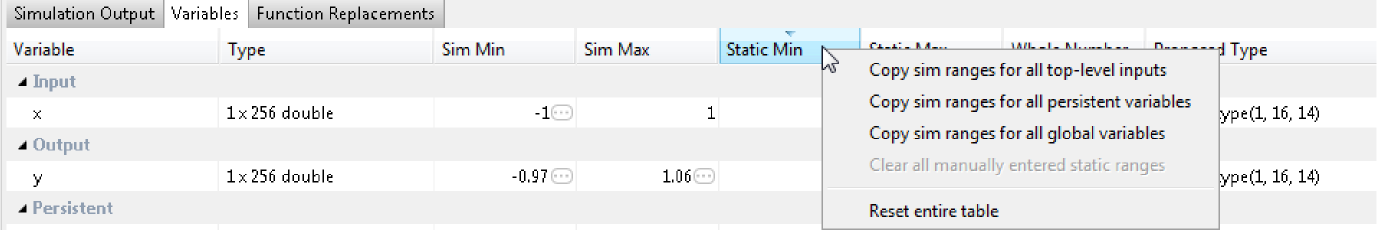
  If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see “Log Data for Histogram” on page 21-81.

Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select `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.



To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select `Copy sim range`.
- Simulation ranges for top-level inputs, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all top-level inputs`.
- Simulation ranges for persistent variables, right-click the Static Min or Static Max column, and then select `Copy sim ranges for all persistent variables`.

Automated Fixed-Point Conversion

In this section...

“Automated Fixed-Point Conversion Capabilities” on page 21-67
“Code Coverage” on page 21-67
“Proposing Data Types” on page 21-70
“Locking Proposed Data Types” on page 21-71
“Viewing Functions” on page 21-72
“Viewing Variables” on page 21-79
“Log Data for Histogram” on page 21-81
“Function Replacements” on page 21-83
“Validating Types” on page 21-83
“Testing Numerics” on page 21-84
“Detecting Overflows” on page 21-84

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 21-71.

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.

```
11      persistent current_state
12      if isempty( current_state )
13          current_state = S1;
14      end
15
16      % switch to new state based on the value state register
17      switch uint8( current_state )
18          case S1
19              % value of output 'Z' depends both on state and inputs
20              if (A)
21                  Z = true;
22                  current_state( 1 ) = S1;
23              else
24                  Z = false;
25                  current_state( 1 ) = S2;
26              end
27          case S2
28              if (A)
29                  Z = false;
30                  current_state( 1 ) = S1;
31              else
32                  Z = true;
33                  current_state( 1 ) = S2;
34              end
35          case S3
36              if (A)
37                  Z = false;
38                  current_state( 1 ) = S2;
39              else
40                  Z = true;
41                  current_state( 1 ) = S3;
42              end
```

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.

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.

```
11      persistent current_state
12      if isempty( current_state )
13          current_state = S1;                                              1 calls
14      end                                                                 51 calls
15
16      % switch to new state based on the value state register
17      switch uint8( current_state )
18          case S1
19              % value of output 'Z' depends both on state and inputs
20              if (A)
21                  Z = true;                                               37 calls
22                  current_state( 1 ) = S1;
23              else                                                         7 calls
24                  Z = false;
25                  current_state( 1 ) = S2;
26              end
27          case S2                                                         51 calls
28              if (A)
29                  Z = false;                                               7 calls
30                  current_state( 1 ) = S1;
31              else                                                         0 calls
32                  Z = true;
33                  current_state( 1 ) = S2;
34              end
35          case S3                                                         51 calls
36              if (A)
37                  Z = false;                                               0 calls
38                  current_state( 1 ) = S2;
39              else
40                  Z = true;
41                  current_state( 1 ) = S3;
42              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 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 21-71.

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.

Derived range analysis is not supported for non-scalar variables.

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 21-71.

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 21-83.

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 21-81.

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. The app can compute derived ranges for scalar variables only.

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 21-71.

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`.

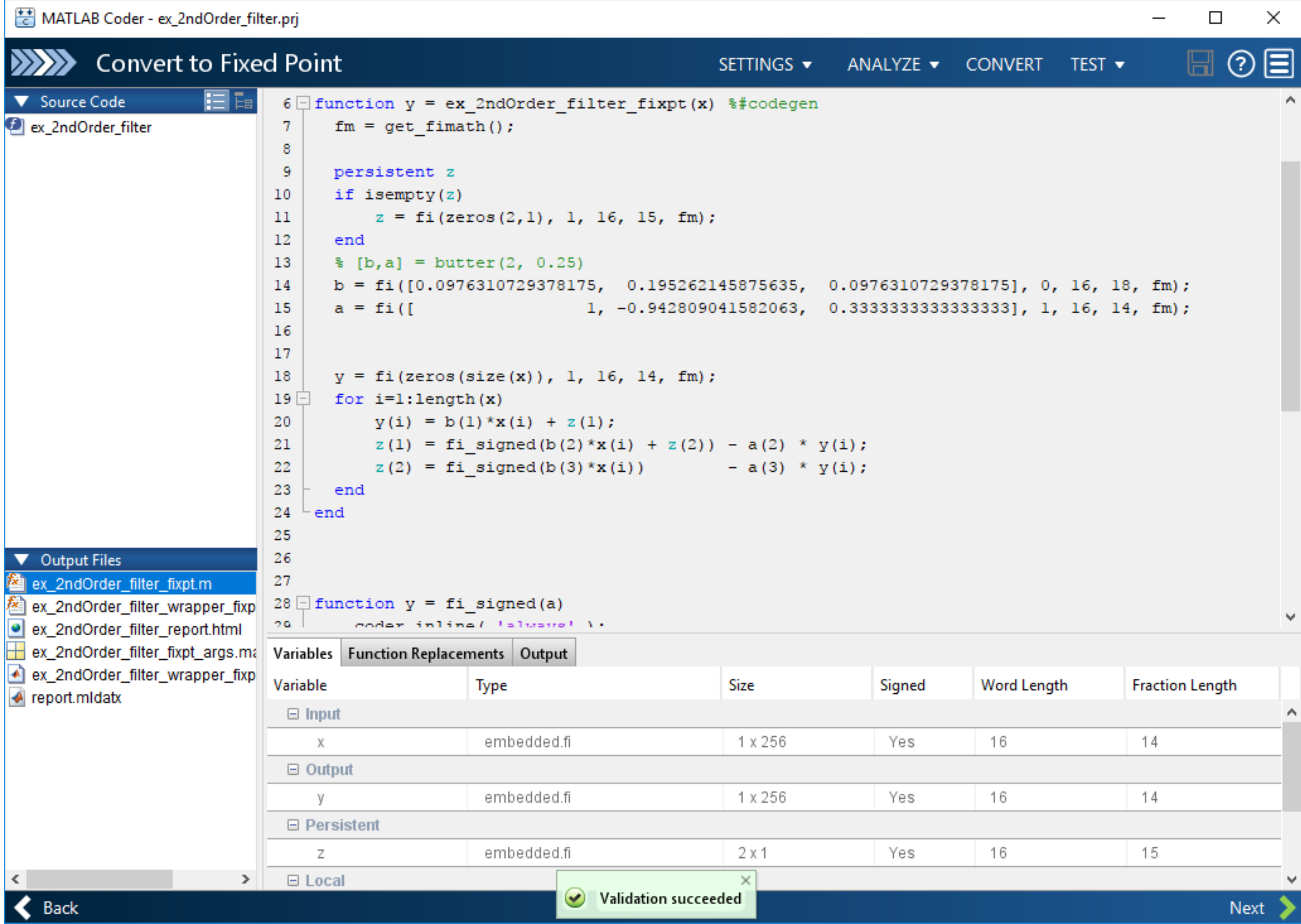
The app displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- 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.

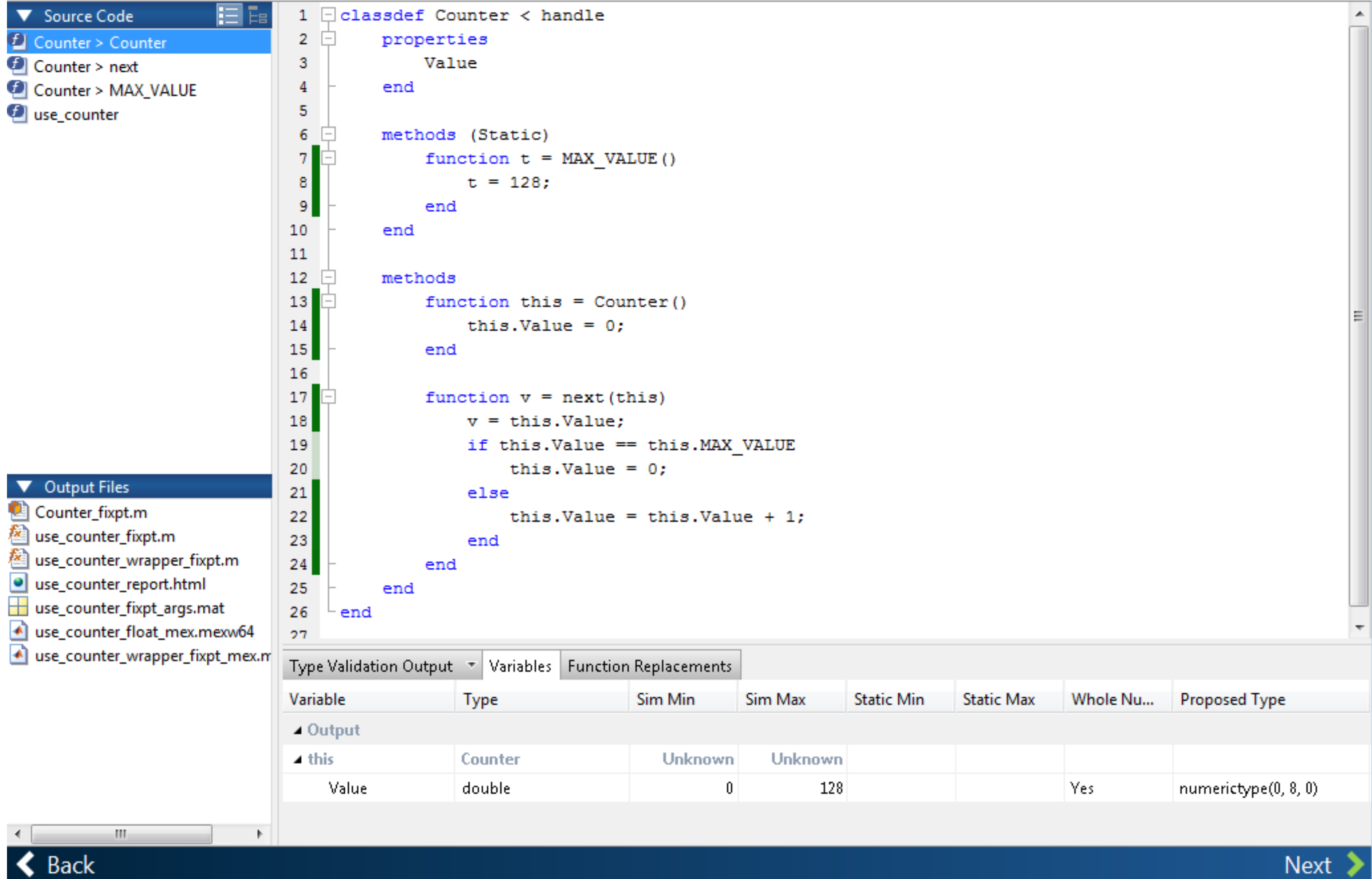
After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the app retains the original function name in the fixed-point file name and appends the fixed-point suffix. For example, here the fixed-point version of `ex_2ndOrder_filter.m` is `ex_2ndOrder_filter_fixpt.m`.



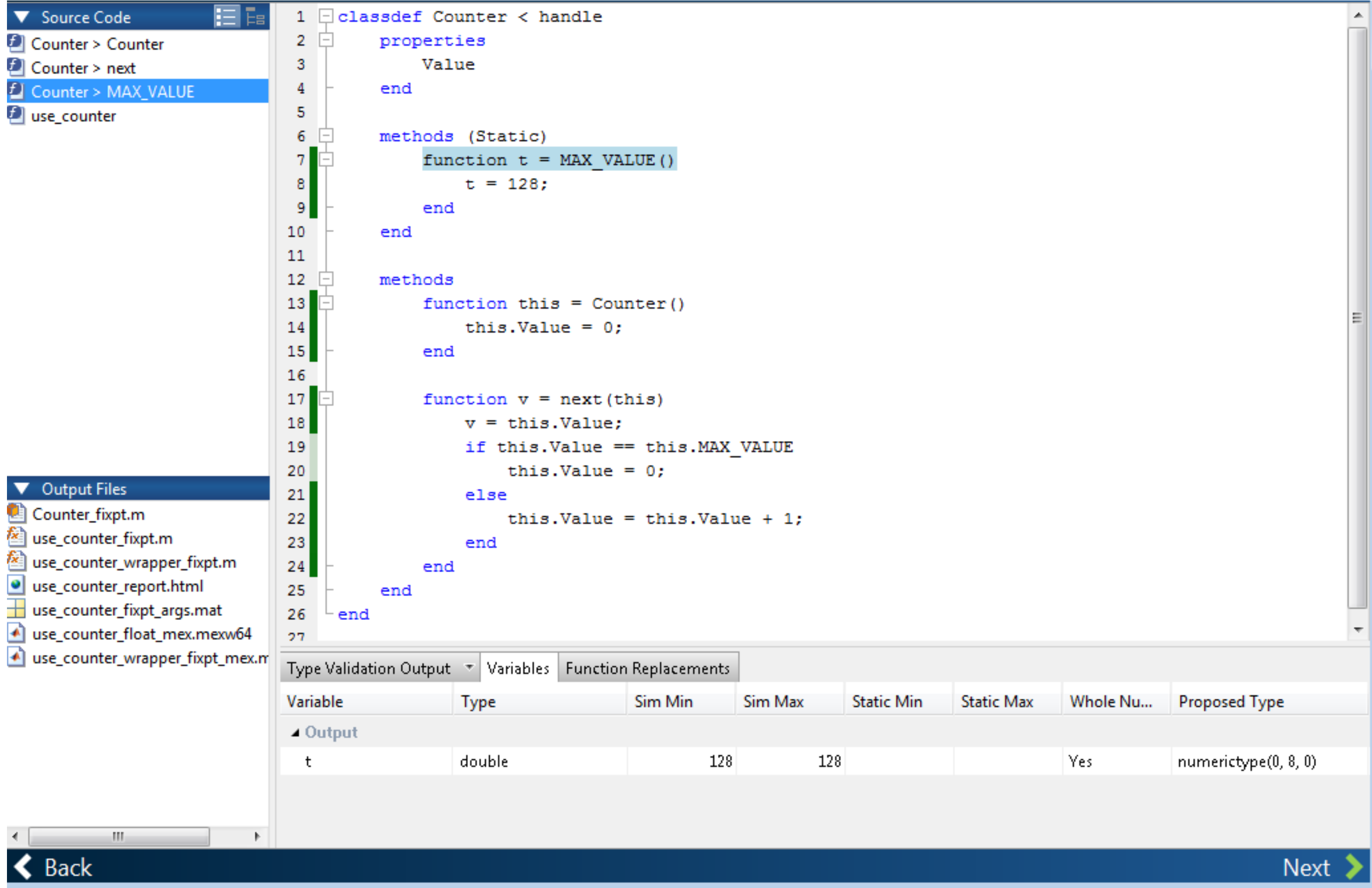
Classes

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.



If you select a method, the app displays only the variables that the method uses.



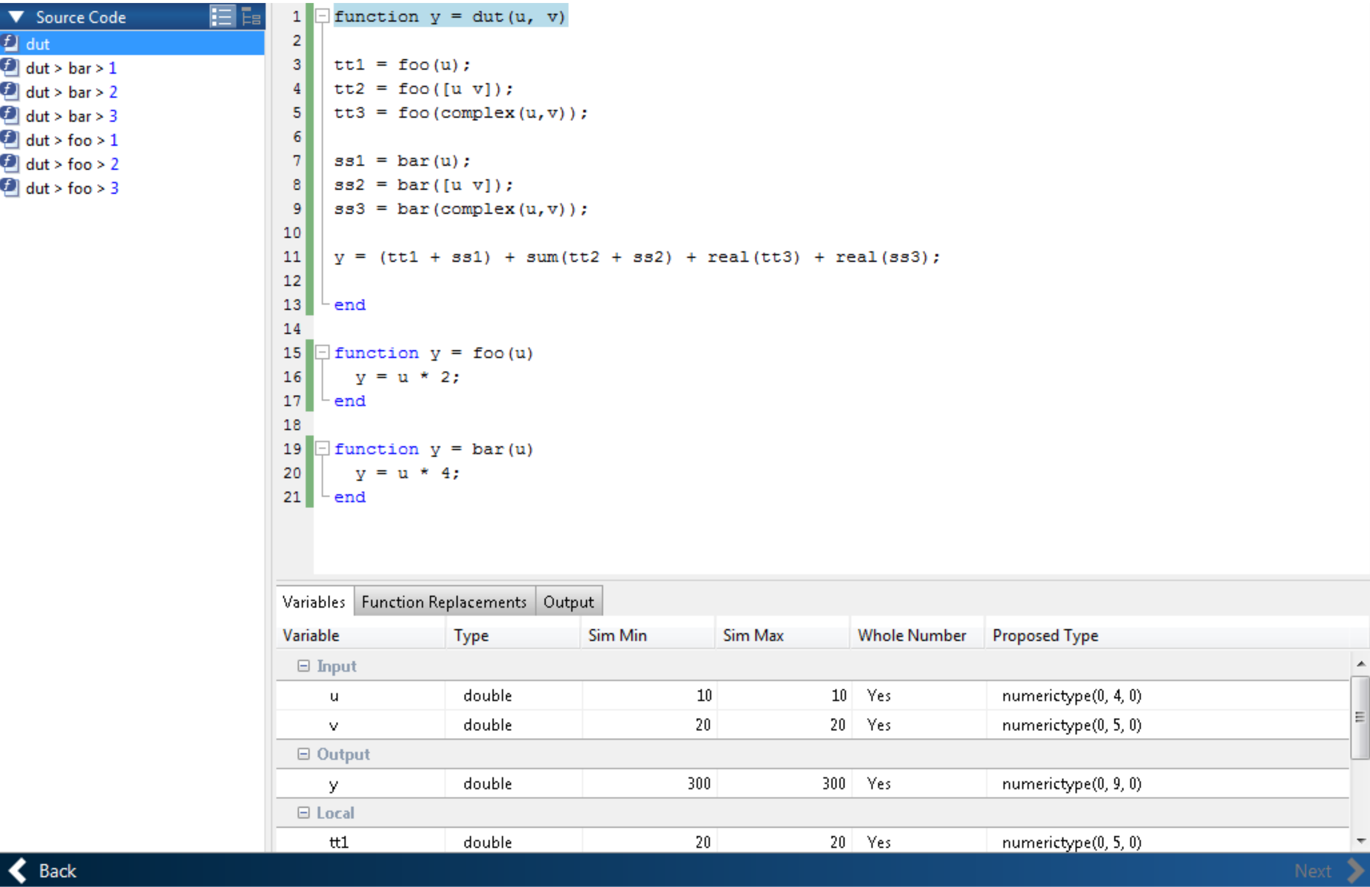
Specializations

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.

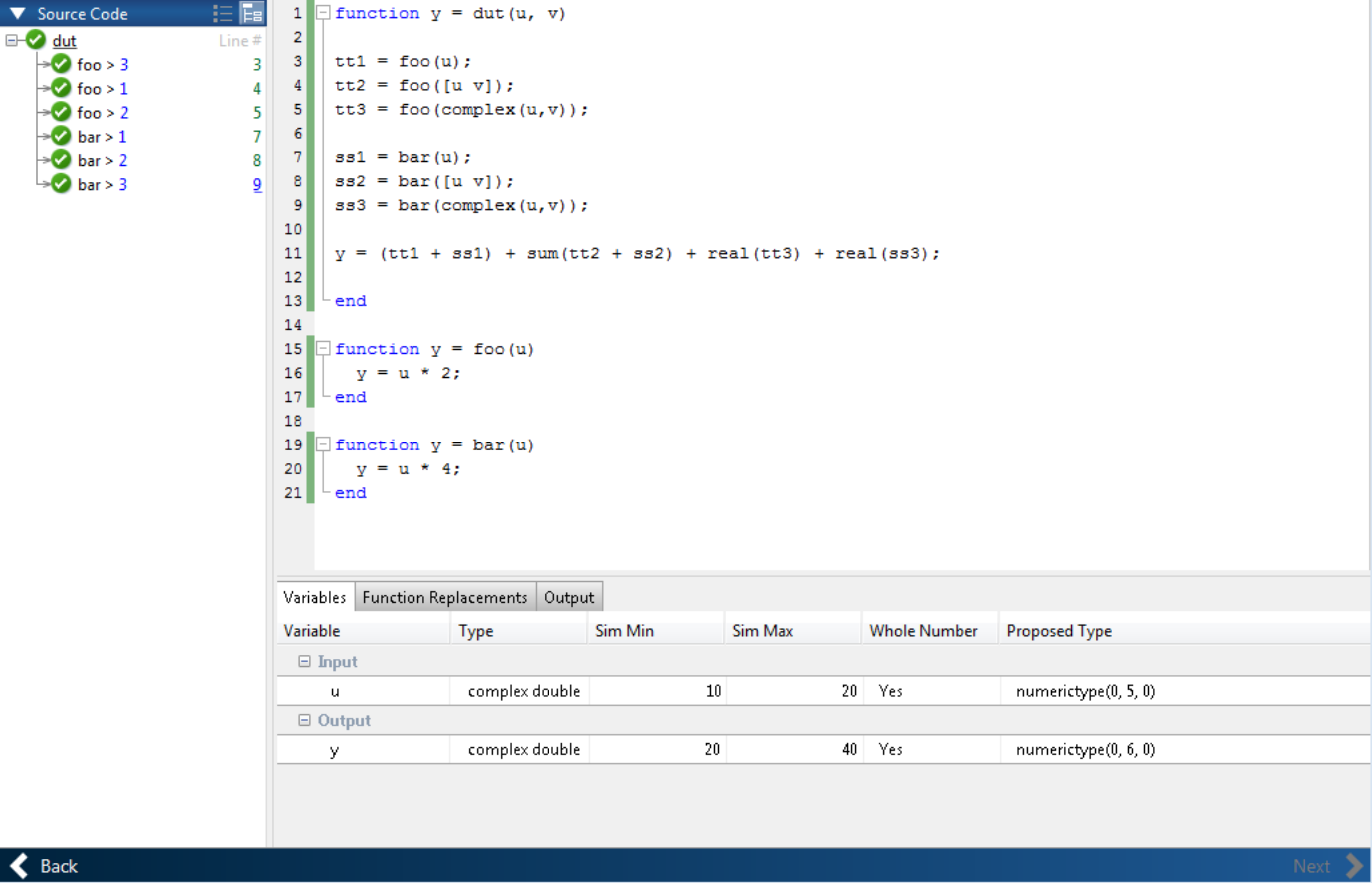
```
function y = dut(u, v)

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
```

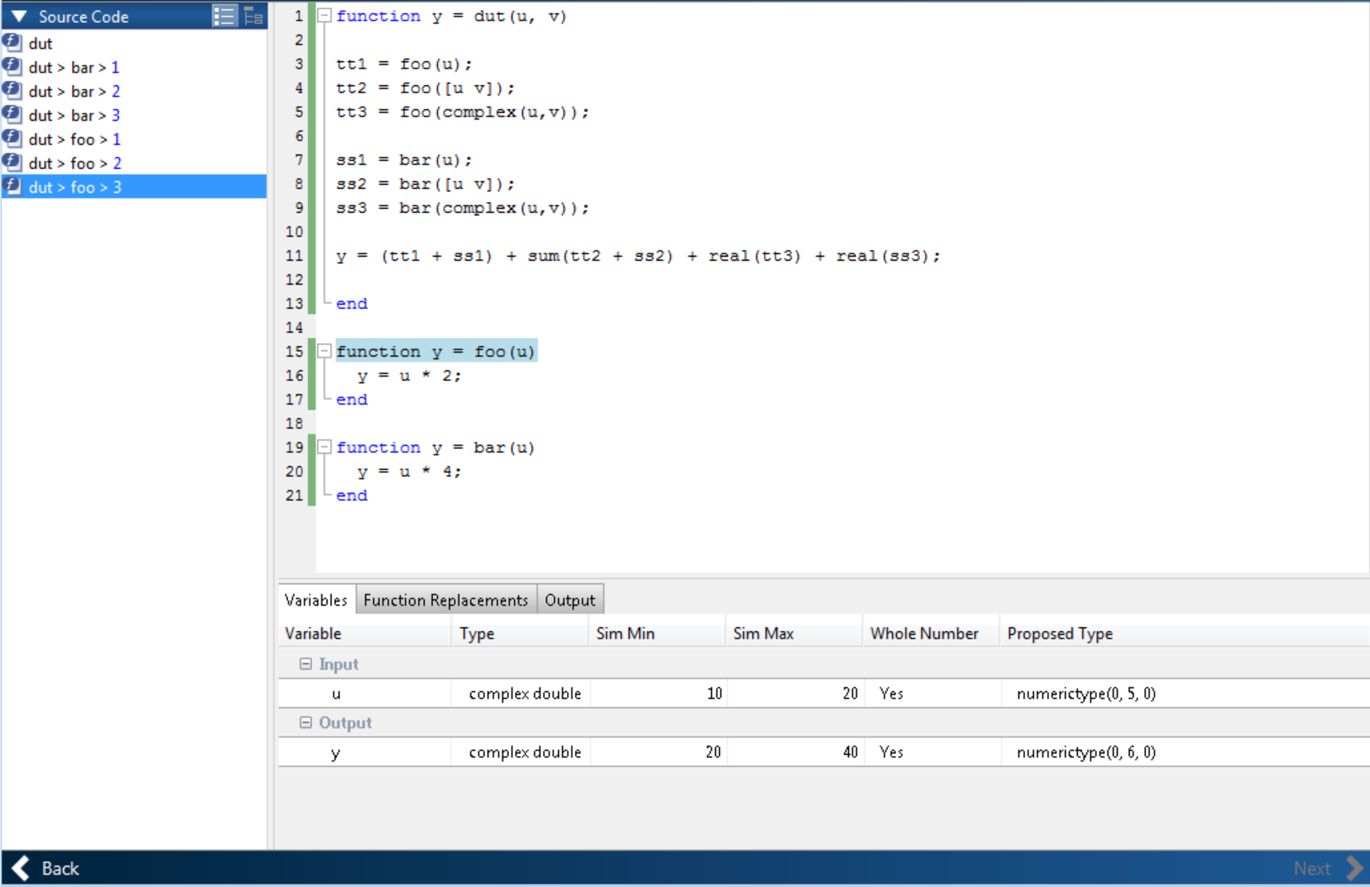
If you select the top-level function, the app displays all the variables on the **Variables** tab.



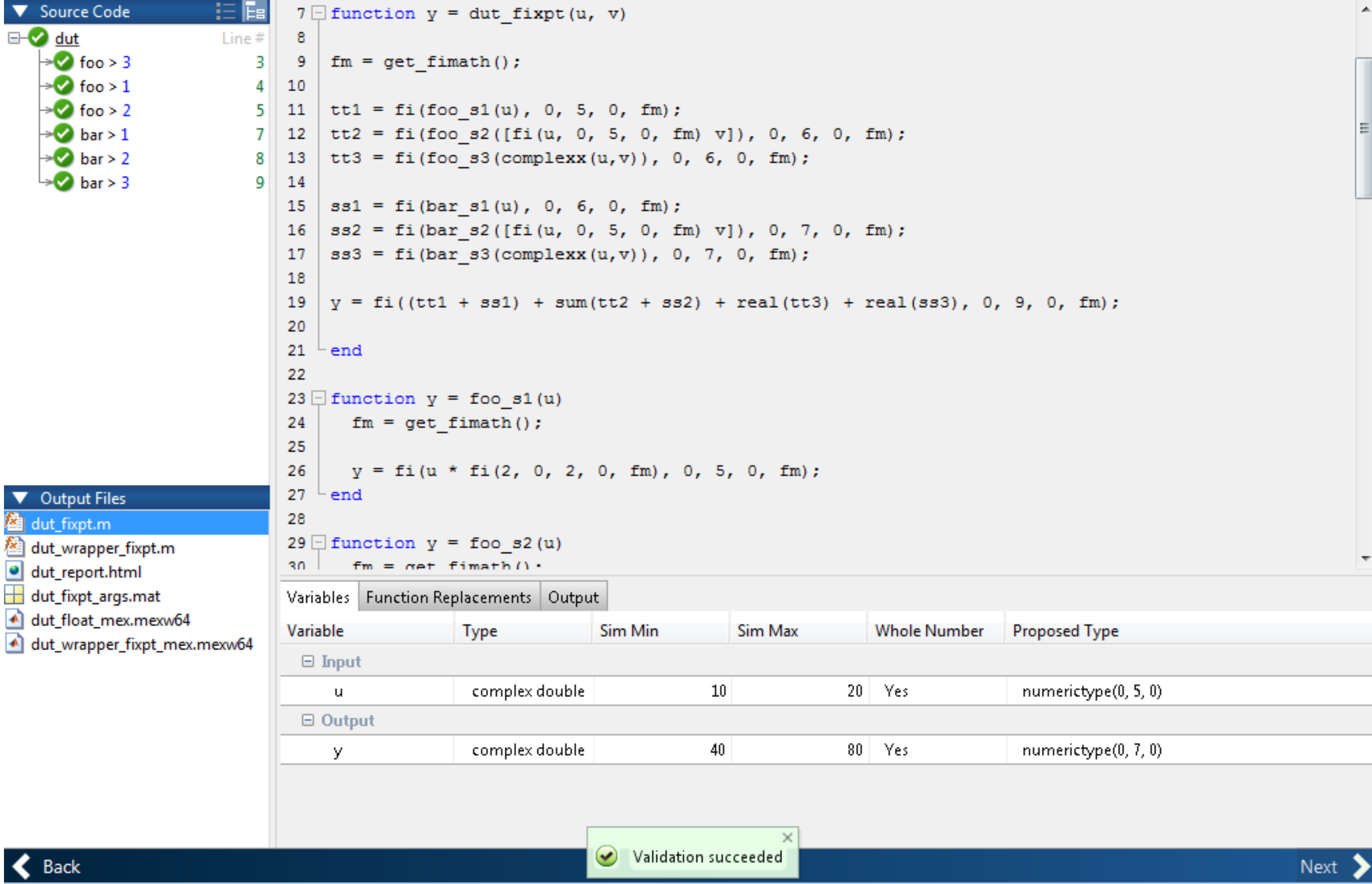
If you select the tree view, the app also displays the line numbers for the call to each specialization.



If you select a specialization, the app displays only the variables that the specialization uses.



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`.



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

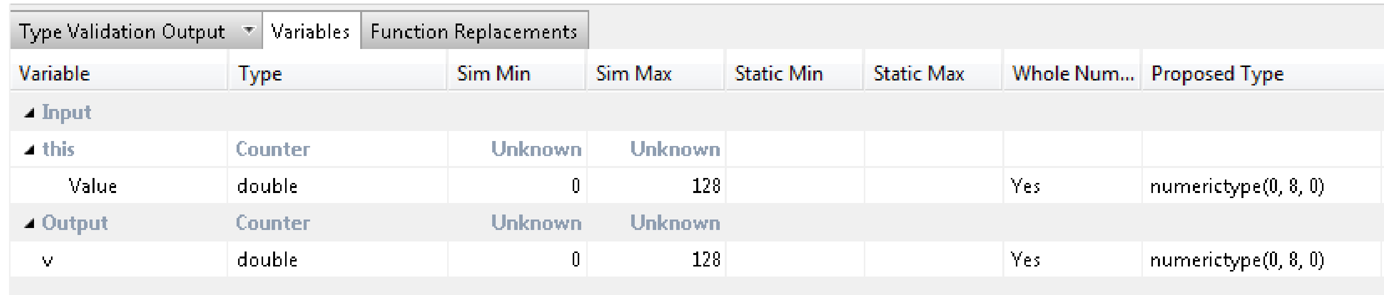
The app displays:

- 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.

```
1   classdef Counter < handle
2       properties
3           Value
4       end
5
6       methods (Static)
7           function t = MAX_VALUE()
8               t = 128;
9           end
10      end
11
12      methods
13          function this = Counter()
14              this.Value = 0;
15          end
16
17          function v = next(this)
18              v = this.Value;
19              if this.Value == this.MAX_VALUE
20                  this.Value = 0;
21              else
22                  this.Value = this.Value + 1;
23              end
24          end
25      end
26  end
```


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.

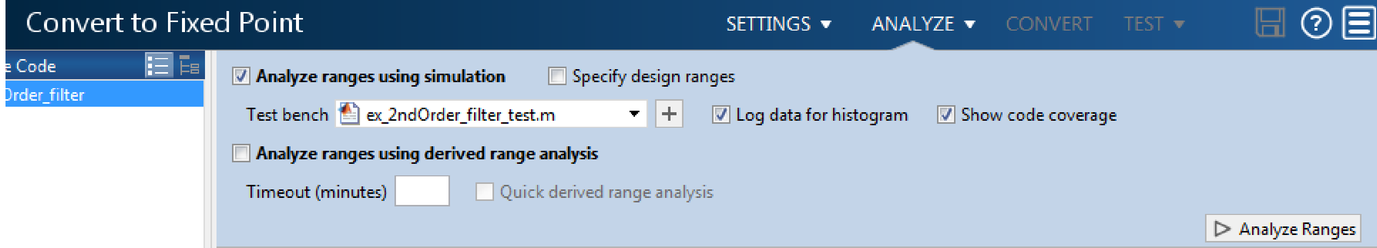


Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Num...	Proposed Type
Input							
this	Counter	Unknown	Unknown				
Value	double	0	128			Yes	numerictype(0, 8, 0)
Output	Counter	Unknown	Unknown				
v	double	0	128			Yes	numerictype(0, 8, 0)

Log Data for Histogram

To log data for histograms:

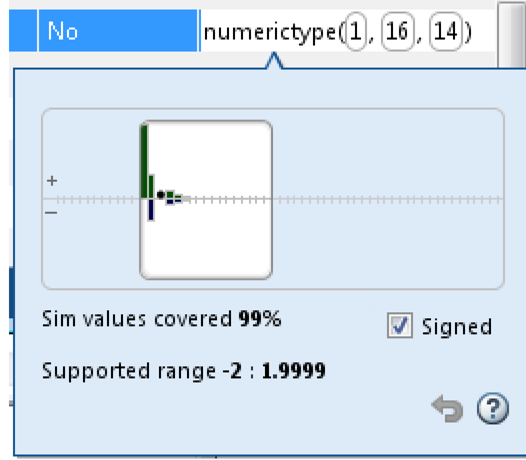
- On the **Convert to Fixed Point** page, click the **Analyze** arrow ▼.
- Select **Log data for histogram**.



- Click **Analyze Ranges**.

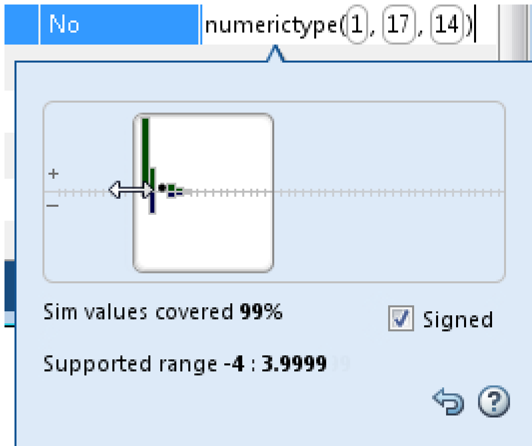
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

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)`.



You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.



- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click .

Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the app lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.

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 24-30.

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

Variables | Function Replacements | Overflows

	Function	Line	Description
⚠	overflow_fixpt	7	Overflow error in expression 'x'.
⚠	overflow_fixpt	7	Overflow error in expression 'y'.
⚠	overflow_fixpt	10	Overflow error in expression 'z'.
⚠	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'x'.
⚠	overflow_fixpt	10	Overflow error in expression 'x*y'.
⚠	overflow_fixpt	10	Overflow error in expression 'y'.
⚠	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the app also provides overflow information for these expressions.

See Also

"Detect Overflows" on page 21-32

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 21-6, 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

`codegen` | `coder` | `coder.FixPtConfig`

Related Examples

- “Convert MATLAB Code to Fixed-Point C Code” on page 21-5
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 21-6
- “Convert MATLAB Coder Project to MATLAB Script” on page 27-35

Generated Fixed-Point Code

In this section...
“Location of Generated Fixed-Point Files” on page 21-87
“Minimizing fi-casts to Improve Code Readability” on page 21-87
“Avoiding Overflows in the Generated Fixed-Point Code” on page 21-88
“Controlling Bit Growth” on page 21-88
“Avoiding Loss of Range or Precision” on page 21-89
“Handling Non-Constant mpower Exponents” on page 21-90

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.
`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" on page 21-92
"Unsupported Constructs" on page 21-92
"Coding Style Best Practices" on page 21-92

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 21-80.
- 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" on page 21-94
"Prepare Your Algorithm for Code Acceleration or Code Generation" on page 21-95
"Check for Fixed-Point Support for Functions Used in Your Algorithm" on page 21-95
"Manage Data Types and Control Bit Growth" on page 21-96
"Convert to Fixed Point" on page 21-96
"Use the Histogram to Fine-Tune Data Type Settings" on page 21-97
"Optimize Your Algorithm" on page 21-97
"Avoid Explicit Double and Single Casts" on page 21-99

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.

<table>
<tr><th>Original code</th><th>Best Practice</th><th>Modified code</th></tr>
<tr><td>

```
% 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);
plot(y-yExpected)
title('Error')
```

</td><td>

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. Put the algorithm in its own function.

</td><td>

Test file

```
% TEST INPUT
x = randn(100,1);

% ALGORITHM
y = cumulative_sum(x);

% VERIFY RESULTS
yExpected = cumsum(x);
plot(y-yExpected)
title('Error')
```

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
```

</td></tr>
</table>

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 21-67.

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” 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 25-5. 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 25-7.

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.

Variables
Function Replacements
Enter a function to replace
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 21-100.

Manage Data Types and Control Bit Growth

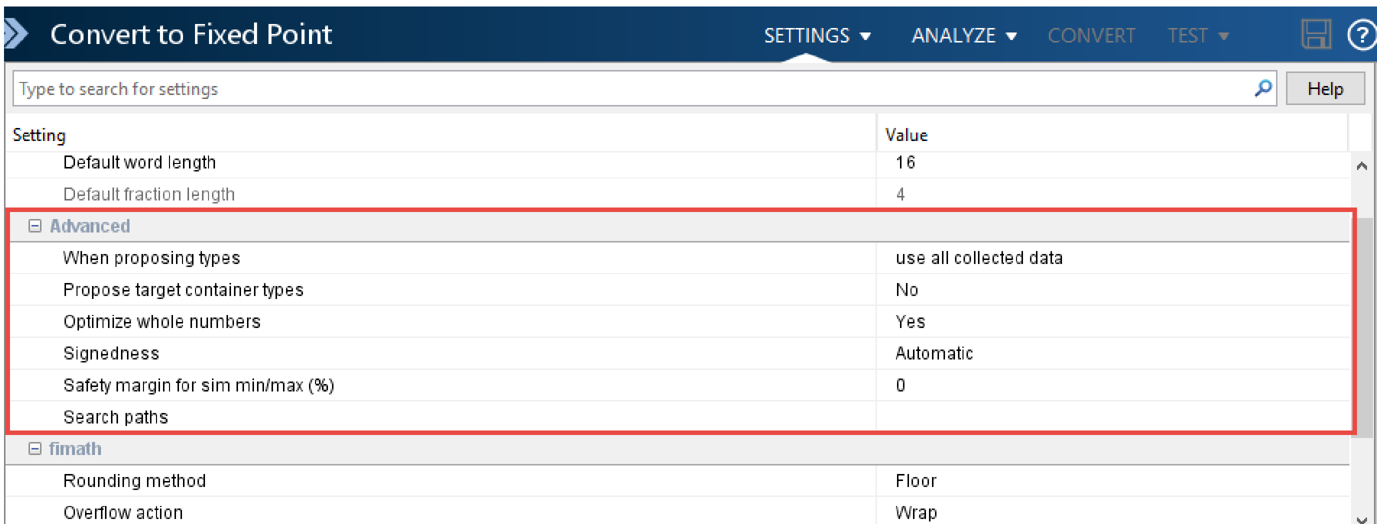
The automated fixed-point conversion process automatically manages data types and controls bit growth. It controls bit growth by using subscripted assignments, that is, assignments that use the colon (:) operator, in the generated code. When you use subscripted assignments, MATLAB overwrites the value of the left-hand side argument but retains the existing data type and array size. In addition to preventing bit growth, subscripted assignment reduces the number of casts in the generated fixed-point code and makes the code more readable.

Convert to Fixed Point

What Are Your Goals for Converting to Fixed Point?

Before you start the conversion, consider your goals for converting to fixed point. Are you implementing your algorithm in C or HDL? What are your target constraints? The answers to these questions determine many fixed-point properties such as the available word length, fraction length, and math modes, as well as available math libraries.

To set up these properties, use the **Advanced** settings.



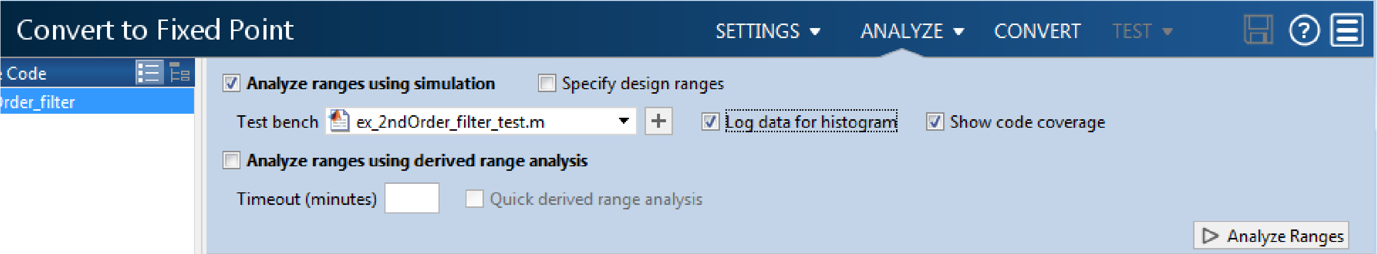
For more information, see "Specify Type Proposal Options" on page 21-29.

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 21-70 and "Log Data for Histogram" on page 21-81 .

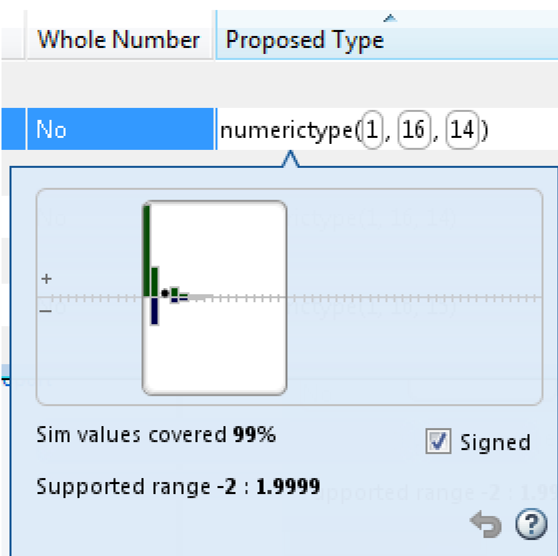
Use the Histogram to Fine-Tune Data Type Settings

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`.



After simulation and static analysis:

- To view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.



  You can view the effect of changing the proposed data types by dragging the edges of the bounding box in the histogram window to change the proposed data type and selecting or clearing the **Signed** option.
- If the values overflow and the range cannot fit the proposed type, the table shows proposed types in red.

When the tool applies data types, it generates an html report that provides overflow information and highlights overflows in red. Review the proposed data types.

Optimize Your Algorithm

Use fimath to Get Optimal Types for C or HDL

`fimath` properties define the rules for performing arithmetic operations on `fi` objects, including math, rounding, and overflow properties. You can use the `fimath` `ProductMode` and `SumMode`

properties to retain optimal data types for C or HDL. HDL can have arbitrary word length types in the generated HDL code whereas C requires container types (`uint8`, `uint16`, `uint32`). Use the **Advanced** settings, see “Specify Type Proposal Options” on page 21-29.

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`.

<table>
<tr><th>MATLAB Code</th><th>Best Practice</th><th>Generated C Code</th></tr>
<tr>
<td rowspan="2">Code being compiled
<pre>
function y = adder(a,b)
  y = a + b;
end
</pre>
Note In the app, set Default word length to <code>16</code>.</td>
<td>Issue

With the default word length set to 16 and the default <code>fimath</code> settings, additional code is generated to implement saturation overflow, nearest rounding, and full-precision arithmetic.</td>
<td><pre>
int adder(short a, short b)
{
  int y;
  int i;
  int i1;
  int i2;
  int i3;
  i = a;
  i1 = b;
  if ((i & 65536) != 0) {
    i2 = i | -65536;
  } else {
    i2 = i & 65535;
  }

  if ((i1 & 65536) != 0) {
    i3 = i1 | -65536;
  } else {
    i3 = i1 & 65535;
  }

  i = i2 + i3;
  if ((i & 65536) != 0) {
    y = i | -65536;
  } else {
    y = i & 65535;
  }

  return y;
}
</pre></td>
</tr>
<tr>
<td>Fix

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 Settings. Select fimath and then set:
<table><tr><td>Rounding method</td><td>Floor</td></tr></table></td>
<td><pre>
int adder(short a, short b)
{
  return a + b;
}
</pre></td>
</tr>
</table>

MATLAB Code	Best Practice		Generated C Code
	Overflow action	Wrap	
	Product mode	KeepLSB	
	Sum mode	KeepLSB	
	Product word length	32	
	Sum word length	32	

HDL

For HDL code generation, set:

- `ProductMode` and `SumMode` to `FullPrecision`
- `Overflow action` to `Wrap`
- `Rounding method` to `Floor`

Replace Built-in Functions with More Efficient Fixed-Point Implementations

Some MATLAB built-in functions can be made more efficient for fixed-point implementation. For example, you can replace a built-in function with a Lookup table implementation, or a CORDIC implementation, which requires only iterative shift-add operations. For more information, see “Function Replacements” on page 21-83.

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 21-83.

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 21-40 and “Replace a Custom Function with a Lookup Table” on page 21-47.

To use lookup table approximations in the programmatic workflow, see `coder.approximation`, “Replace the exp Function with a Lookup Table” on page 22-19, and “Replace a Custom Function with a Lookup Table” on page 22-21.

MATLAB Language Features Supported for Automated Fixed-Point Conversion

In this section...

"MATLAB Language Features Supported for Automated Fixed-Point Conversion" on page 21-101
"MATLAB Language Features Not Supported for Automated Fixed-Point Conversion" on page 21-102

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 27-98). 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 20-2)
- 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.BiquadFilter`
  - `dsp.FIRDecimator`
  - `dsp.FIRInterpolator`
  - `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
  - `dsp.FIRRateConverter`
  - `dsp.VariableFractionalDelay`

MATLAB Language Features Not Supported for Automated Fixed-Point Conversion

Fixed-Point Designer does not support the following features in automated fixed-point conversion:

- Anonymous functions
- Cell arrays
- String scalars
- Objects of value classes as entry-point function inputs or outputs
- Function handles
- Java
- Nested functions
- Recursion
- Sparse matrices
- `try`/`catch` statements
- `varargin`, `varargout`, or generation of fewer input or output arguments than an entry-point function defines
- Dot indexing properties of fixed-point data types.

  Avoid using properties of fixed-point types in the code being converted by the Fixed-Point Converter app, and in MATLAB Function blocks being converted by the Fixed-Point Tool.

Inspecting Data Using the Simulation Data Inspector

In this section...

"What Is the Simulation Data Inspector?" on page 21-103
"Import Logged Data" on page 21-103
"Export Logged Data" on page 21-103
"Group Signals" on page 21-103
"Run Options" on page 21-103
"Create Report" on page 21-104
"Comparison Options" on page 21-104
"Enabling Plotting Using the Simulation Data Inspector" on page 21-104
"Save and Load Simulation Data Inspector Sessions" on page 21-104

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 21-53.

To enable the Simulation Data Inspector in the programmatic workflow, see “Enable Plotting Using the Simulation Data Inspector” on page 22-23.

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 22-24.

Data Type Issues in Generated Code

Within the fixed-point conversion 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 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” on page 22-2
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 22-4
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 22-9
- “Detect Overflows” on page 22-16
- “Replace the exp Function with a Lookup Table” on page 22-19
- “Replace a Custom Function with a Lookup Table” on page 22-21
- “Enable Plotting Using the Simulation Data Inspector” on page 22-23
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 22-24

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, select the function in the **MATLAB Source** pane. To view the fixed-point C code, select the file in the **Generated Code** pane.

See Also

`coder.FixptConfig`

Related Examples

- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 22-4
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 22-9
- “Enable Plotting Using the Simulation Data Inspector” on page 22-23

More About

- “Automated Fixed-Point Conversion” on page 21-67

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

1. Create a local working folder, for example, `c:\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*

Simulation Coverage	Code
100%	`function y = ex_2ndOrder_filter(x) %#codegen` `persistent z`

Once	`if isempty(z)`
	`z = zeros(2,1);`
	`end`

100%	`% [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	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

`codegen` | `coder.FixptConfig`

Related Examples

- “Convert MATLAB Code to Fixed-Point C Code” on page 21-5
- “Propose Fixed-Point Data Types Based on Derived Ranges” on page 22-9

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.

Note Derived range analysis is not supported for non-scalar variables.

Prerequisites

To complete this example, you must install the following products:

- MATLAB
- MATLAB Coder
- Fixed-Point Designer
- C compiler

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `dti.m` and `dti_test.m` files to your local working folder.

Type	Name	Description
Function code	`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
%
```

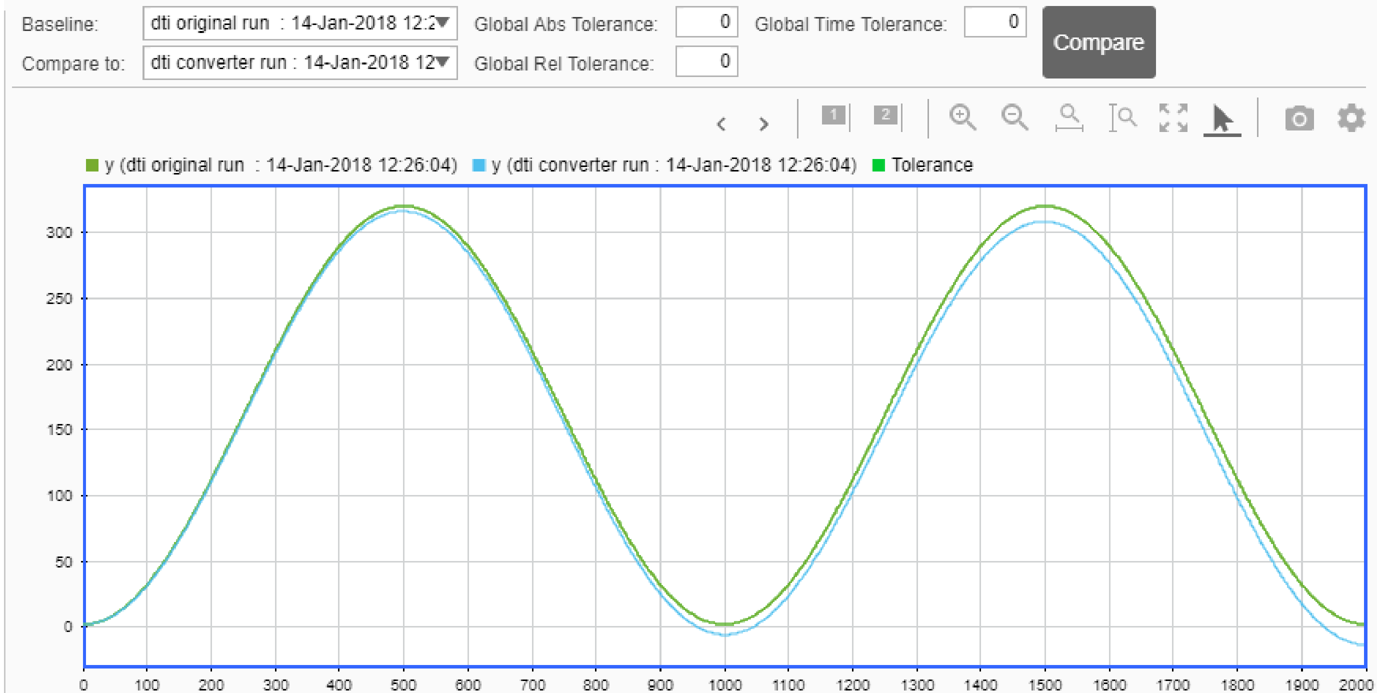
```
% Forward Euler method, also known as Forward Rectangular, or left-hand
% approximation.  The resulting expression for the output of the block at
% step 'n' is y(n) = y(n-1) + K * u(n-1)
%
fm = get_fimath();

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.



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

`codegen` | `coder.FixptConfig`

Related Examples

- “Convert MATLAB Code to Fixed-Point C Code” on page 21-5
- “Propose Fixed-Point Data Types Based on Simulation Ranges” on page 22-4

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create Algorithm and Test Files

1. Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

   ```
   function y = my_fcn(x)
     y = exp(x);
   end
   ```
2. Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

   ```
   close all

   x = linspace(-10,10,1e3);
   for itr = 1e3:-1:1
      y(itr) = my_fcn( x(itr) );
   end
   plot( x, y );
   ```

Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'my_fcn_test';
fixptcfg.TestNumerics = true;
fixptcfg.DefaultWordLength = 16;
fixptcfg.addApproximation(q);
```

Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
  y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
  y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
   y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

Create a function replacement configuration object to approximate `custom_fcn`. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...
                        'CandidateFunction',@custom_fcn, 'NumberOfPoints',50);
```

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'custom_test';
fixptcfg.TestNumerics = true;
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg call_custom_fcn
```

`codegen` generates fixed-point MATLAB code in `call_custom_fcn_fixpt.m`.

To view the generated fixed-point code, click the link to `call_custom_fcn_fixpt`.

The generated code contains a lookup table approximation, `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 (Simulink) 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 22-9.

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a New Folder and Copy Relevant Files

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

   ```
   cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))
   ```
3. Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	`myFilter.m`	Entry-point MATLAB function
Test file	`myFilterTest.m`	MATLAB script that tests `myFilter.m`
Plotting function	`plotDiff.m`	Custom plot function
MAT-file	`filterData.mat`	Data to filter.

The myFilter Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
  b = complex(zeros(1,16));
```

```
  h = complex(zeros(1,16));
  h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end
```

The myFilterTest File

```
% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end
```

The plotDiff Function

```
% varInfo - structure with information about the variable. It has the following fields
%            i) name
%            ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after Fixed-Point conve
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % 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;
```

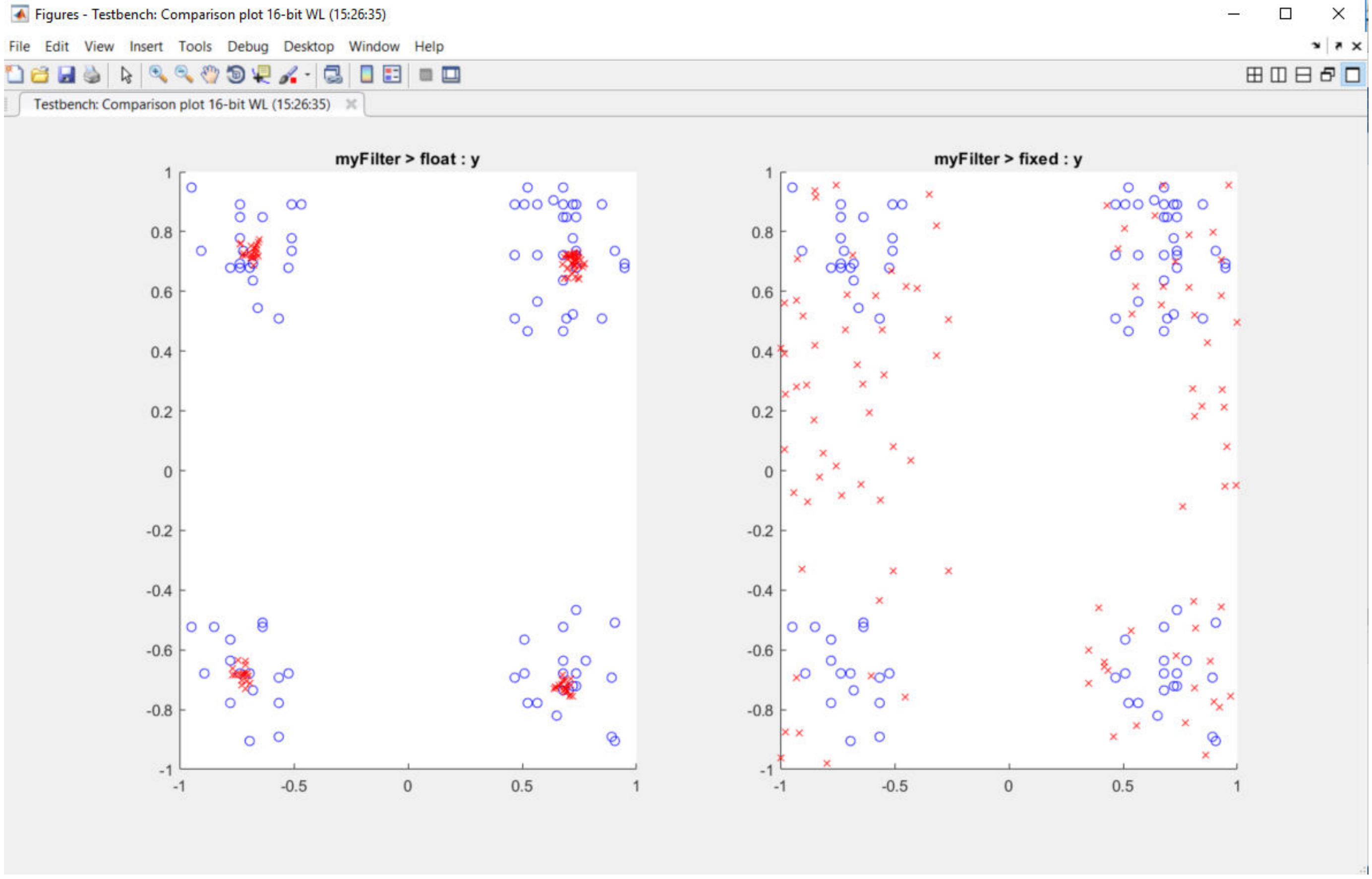
Convert to Fixed Point

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.

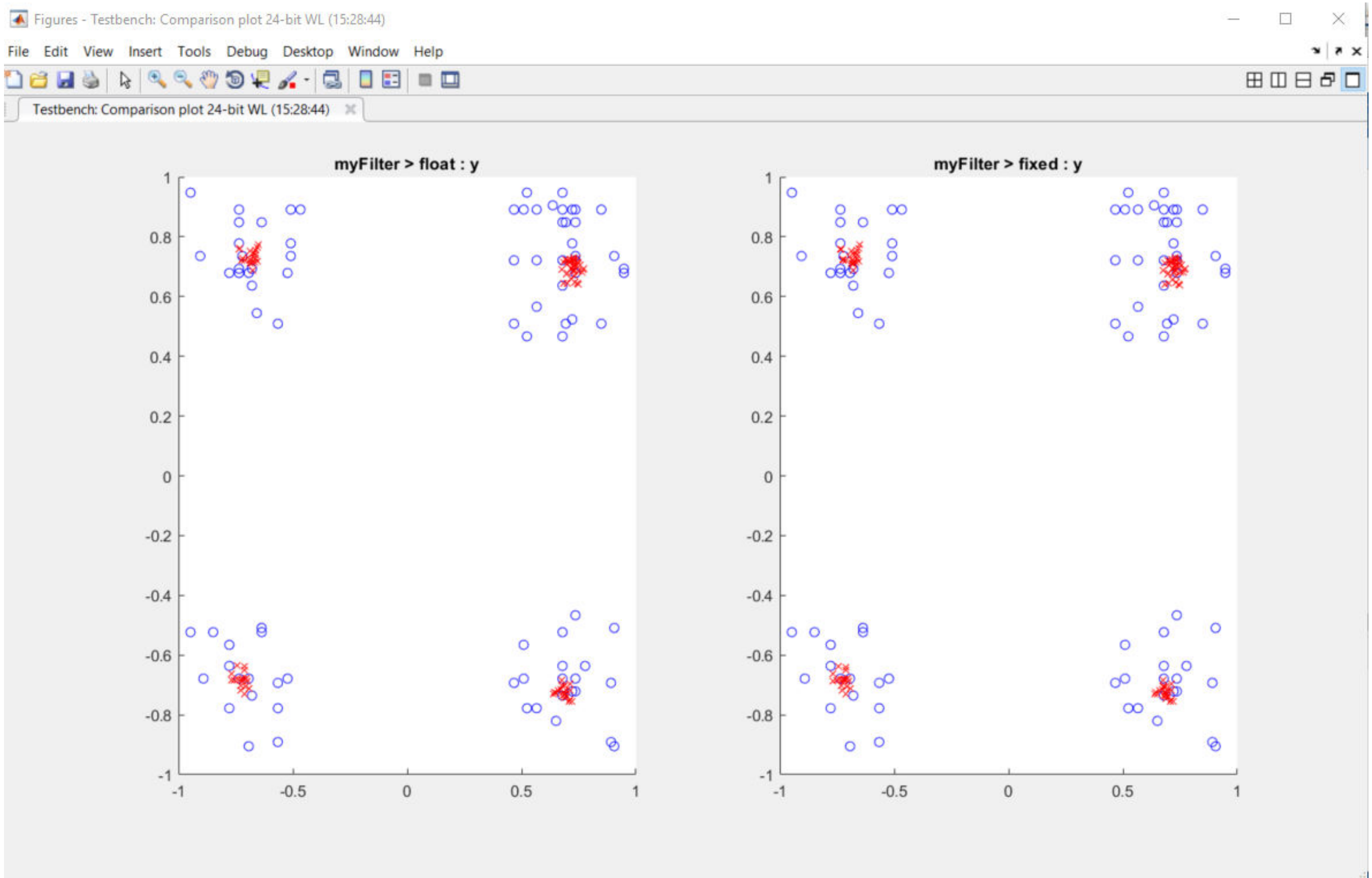


The plot shows that the fixed-point results do not closely match the floating-point results.

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.


Figures - Testbench: Comparison plot 24-bit WL (15:28:44)
File Edit View Insert Tools Debug Desktop Window Help
Testbench: Comparison plot 24-bit WL (15:28:44)
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
-0.5
0.5

Single-Precision Conversion

- “Generate Single-Precision C Code at the Command Line” on page 23-2
- “Generate Single-Precision C Code Using the MATLAB Coder App” on page 23-6
- “Generate Single-Precision MATLAB Code” on page 23-11
- “Choose a Single-Precision Conversion Workflow” on page 23-18
- “Single-Precision Conversion Best Practices” on page 23-19
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 23-22
- “Combining Integers and Double-Precision Numbers” on page 23-24
- “MATLAB Language Features Supported for Single-Precision Conversion” on page 23-25

Generate Single-Precision C Code at the Command Line

In this section...

"Prerequisites" on page 23-2
"Create a Folder and Copy Relevant Files" on page 23-2
"Determine the Type of the Input Argument" on page 23-4
"Generate and Run Single-Precision MEX to Verify Numerical Behavior" on page 23-4
"Generate Single-Precision C Code" on page 23-4
"View the Generated Single-Precision C Code" on page 23-4
"View Potential Data Type Issues" on page 23-5

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a Folder and Copy Relevant Files

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.

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

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.

In the **Generated Code** pane, click `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 **Code Insights** tab. Expand **Potential data type issues**. The absence of double-precision operations indicates 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 23-6
- “Generate Single-Precision MATLAB Code” on page 23-11

More About

- “Single-Precision Conversion Best Practices” on page 23-19
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 23-22

Generate Single-Precision C Code Using the MATLAB Coder App

In this section...

"Prerequisites" on page 23-6
"Create a Folder and Copy Relevant Files" on page 23-6
"Open the MATLAB Coder App" on page 23-8
"Select the Source Files" on page 23-8
"Enable Single-Precision Conversion" on page 23-8
"Define Input Types" on page 23-9
"Check for Run-Time Issues" on page 23-9
"Generate Single-Precision C Code" on page 23-10
"View the Generated C Code" on page 23-10
"View Potential Data Type Issues" on page 23-10

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a Folder and Copy Relevant Files

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`.

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. For other settings, use the default values.
4. 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 **Code Insights** tab. Expand **Potential data type issues**. The absence of double-precision operations indicates that no double-precision operations remain.

See Also

Related Examples

- “Generate Single-Precision C Code at the Command Line” on page 23-2

More About

- “Single-Precision Conversion Best Practices” on page 23-19
- “Warnings from Conversion to Single-Precision C/C++ Code” on page 23-22

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

  See `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a Folder and Copy Relevant Files

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	Code
100%	`function y = ex_2ndOrder_filter(x) %#codegen`
	`persistent z`
Once	`if isempty(z)`
	`z = zeros(2,1);`
	`end`
100%	`% [b,a] = butter(2, 0.25)`

```
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. In the **MATLAB Source** pane, click `ex_2ndOrder_filter_single`.

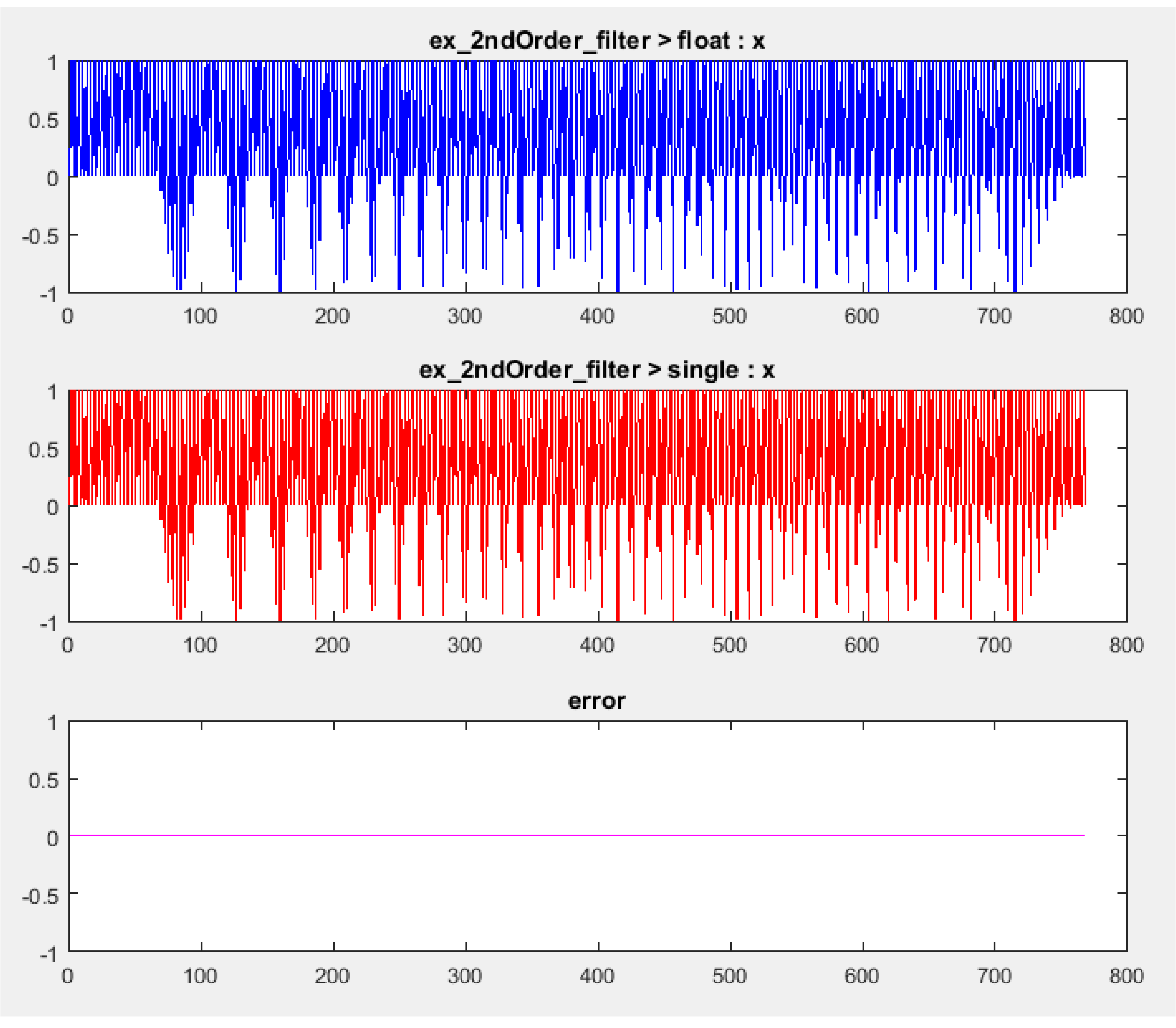
The code generation report displays the single-precision MATLAB code for `ex_2ndOrder_filter`.

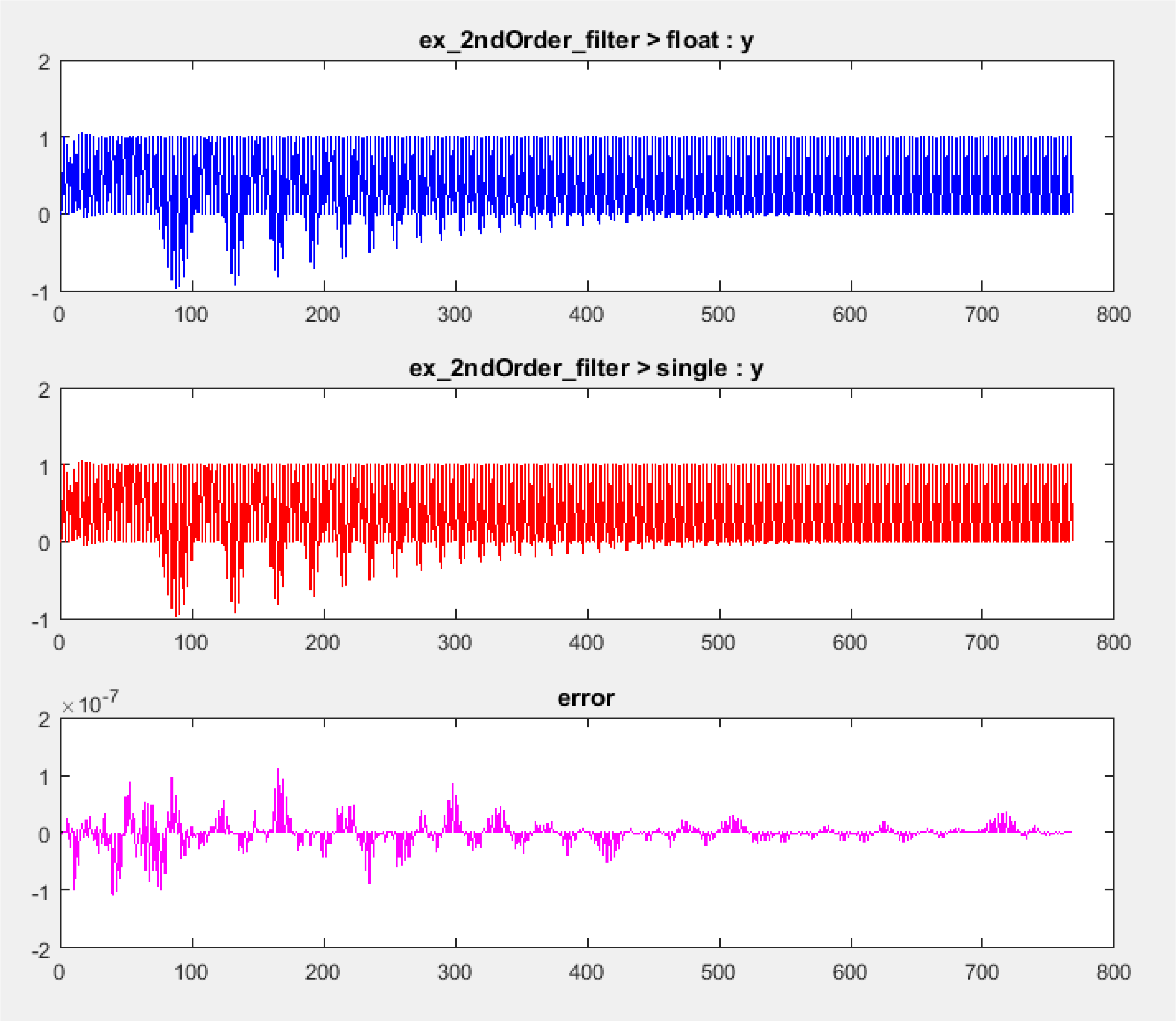
View Potential Data Type Issues

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 **Code Insights** tab. The absence of potential data type issues indicates that no double-precision operations remain.

Compare the Double-Precision and Single-Precision Variables

You can see the comparison plots for the input x and output y because you selected to log inputs and outputs for comparison plots .





Optionally Generate Single-Precision C Code

If you also want to generate single-precision C code, create a code configuration object for C code generation. Use this configuration object with the `-config` option of the `codegen` function. For example:

1 Create a code configuration object for generation of a C static library.

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

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.

In the **Generated Code** pane, click `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 **Code Insights** tab. Then, expand **Potential data type issues**. The absence of double-precision operations indicates that no double-precision operations remain.

See Also

`codegen` | `coder.SingleConfig` | `coder.config`

Related Examples

- "Generate Single-Precision C Code Using the MATLAB Coder App" on page 23-6
- "Generate Single-Precision C Code at the Command Line" on page 23-2

More About

- "Single-Precision Conversion Best Practices" on page 23-19
- "Warnings from Conversion to Single-Precision C/C++ Code" on page 23-22

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 23-2.
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 23-6.
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 23-11.
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 23-11.

Single-Precision Conversion Best Practices

In this section...

"Use Integers for Index Variables" on page 23-19
"Limit Use of assert Statements" on page 23-19
"Initialize MATLAB Class Properties in Constructor" on page 23-19
"Provide a Test File That Calls Your MATLAB Function" on page 23-19
"Prepare Your Code for Code Generation" on page 23-20
"Verify Double-Precision Code Before Single-Precision Conversion" on page 23-20
"Best Practices for Generation of Single-Precision C/C++ Code" on page 23-20
"Best Practices for Generation of Single-Precision MATLAB Code" on page 23-21

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 25-5. 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 25-7.

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 26-2.

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:

Use the C99 Standard Math Library

When you generate C/C++ libraries or executables, by default, the code generator uses the C99 (ISO) standard math library. If you generate single-precision C/C++ code using the C89/C90 (ANSI) 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 23-22.

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;
```

See Also

More About

- “Warnings from Conversion to Single-Precision C/C++ Code” on page 23-22

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` using the C89/C90 standard.

```
x = -pi:0.01:pi;
cfg = coder.config('lib');
cfg.TargetLangStandard = 'C89/C90 (ANSI)';
codegen -singleC -config cfg mysine -args {x} -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 **Code Insights** tab. Expand **Potential data type issues** and then expand **Double-precision operations**. The report indicates that `mysine` has a double-precision operation at line 2 `c = sin(a)`.

To address this warning, use the default standard math library, C99 (ISO).

- At the command line:

  ```
  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 **Code Insights** tab. Expand **Potential data type issues** and then expand **Double-precision operations**. The report indicates that `geterf` has a double-precision operation at line 2 `y = erf(x)` .

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 **Code Insights** tab. Expand **Potential data type issues** and then expand **Double-precision operations**. The report indicates that `mysum` has a double-precision operation at line 2 `y = sum(int32(x))`.

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.

See Also

More About

- “Single-Precision Conversion Best Practices” on page 23-19

Combining Integers and Double-Precision Numbers

MATLAB supports the combination of integers of the same class and scalar double-precision numbers. MATLAB does not support the combination of integers and single-precision numbers. If you use the MATLAB Coder app or `codegen` with the `-singleC` option to generate single-precision C/C++ code, your MATLAB code cannot combine integers and double-precision numbers. Converting an expression that combines integers and doubles results in an illegal MATLAB expression. To work around this limitation, cast the numbers so that the types of the numbers match. Either cast the integer numbers to double-precision or cast the double-precision numbers to the integer class.

For example, consider the function `dut` that returns the sum of `a` and `b`.

```
function c = dut(a,b)
c = a + b;
end
```

Generate single-precision code using `codegen` with the `-singleC` option. Specify that the first argument is double and the second argument is int32.

```
codegen -singleC -config:lib dut -args {0, int32(2)} -report
```

Code generation fails. The message suggests that you cast the operands so that they have the same types.

Rewrite the code so that it cast `a` to the type of `b`.

```
function c = dut(a,b)
c = int32(a) + b;
end
```

MATLAB Language Features Supported for Single-Precision Conversion

In this section...

"MATLAB Language Features Supported for Single-Precision Conversion" on page 23-25
"MATLAB Language Features Not Supported for Single-Precision Conversion" on page 23-26

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 27-98). Comparison plotting does not support variable-size data.
- Subscripting (see "Incompatibility with MATLAB in Matrix Indexing Operations for Code Generation" on page 6-19).
- Complex numbers (see "Code Generation for Complex Data" on page 5-3).
- Numeric classes (see "Supported Variable Types" on page 4-11).
- 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.

  Single-precision conversion using `codegen` with the `-singleC` option does not support classes when the properties have default values. Property values must be initialized in the constructor. Constant properties cannot be initialized to double precision data types.

- Function calls (see "Resolution of Function Calls for Code Generation" on page 20-2)
- `varargin` and `varargout` are supported when you generate single-precision C/C++ code by using the MATLAB Coder app or `codegen` with `-singleC`. They are not supported when you use `codegen` with `-double2single`.

  For functions that do not use `varargin` or `varargout`, you can control the number of input or output arguments in the generated entry-point function only if you generate single-precision C/C++ code by using the MATLAB Coder app or `codegen` with `-singleC`.

MATLAB Language Features Not Supported for Single-Precision Conversion

Single-precision conversion does not support the following features:

- Anonymous functions
- Cell arrays
- String scalars
- Objects of value classes as entry-point function inputs or outputs
- Function handles
- Java
- Nested functions
- Recursion
- Sparse matrices
- `try`/`catch` statements
- `varargin` and `varargout`, or generation of fewer input or output arguments than an entry-point function defines

Setting Up a MATLAB Coder Project

- “Set Up a MATLAB Coder Project” on page 24-2
- “Specify Properties of Entry-Point Function Inputs Using the App” on page 24-3
- “Automatically Define Input Types by Using the App” on page 24-4
- “Make Dimensions Variable-Size When They Meet Size Threshold” on page 24-5
- “Define Input Parameter by Example by Using the App” on page 24-6
- “Define or Edit Input Parameter Type by Using the App” on page 24-14
- “Define Constant Input Parameters Using the App” on page 24-23
- “Define Inputs Programmatically in the MATLAB File” on page 24-24
- “Add Global Variables by Using the App” on page 24-25
- “Specify Global Variable Type and Initial Value Using the App” on page 24-26
- “Undo and Redo Changes to Type Definitions in the App” on page 24-29
- “Code Generation Readiness Screening in the MATLAB Coder App” on page 24-30
- “Slow Operations in MATLAB Coder App” on page 24-31
- “Unable to Open a MATLAB Coder Project” on page 24-32

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 24-2 and “Open an Existing Project” on page 24-2.
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 24-3.
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 27-13.

   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.

The MATLAB Coder app is not supported in MATLAB Online™.

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 27-43

Specify an Input Definition Using the App

Specify an input definition using one of the following methods:

- Autodefine Input Types on page 24-4
- Define Type on page 24-14
- Define by Example on page 24-6
- Define Constant on page 24-23
- “Define Inputs Programmatically in the MATLAB File” on page 24-24

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.

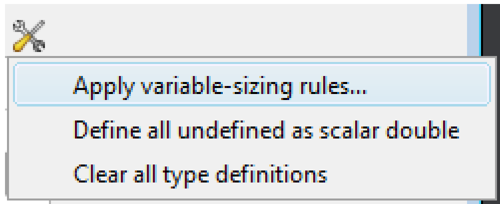
If your test file does not call an entry-point function with different size inputs, the resulting type dimensions are fixed-size. After you define the input types, you can specify and apply rules for making type dimensions variable-size when they meet a size threshold. See “Make Dimensions Variable-Size When They Meet Size Threshold” on page 24-5.

The MATLAB Coder app is not supported in MATLAB Online.

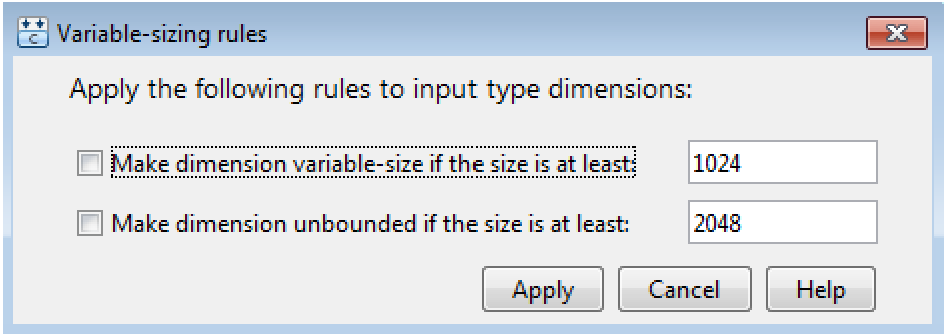
Make Dimensions Variable-Size When They Meet Size Threshold

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`.



2. In the **Variable-sizing rules** dialog box, select the rules that you want to apply.



   - 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.

The MATLAB Coder app is not supported in MATLAB Online.

See Also

More About

- “Specify Properties of Entry-Point Function Inputs” on page 27-43
- “Code Generation for Variable-Size Arrays” on page 6-2

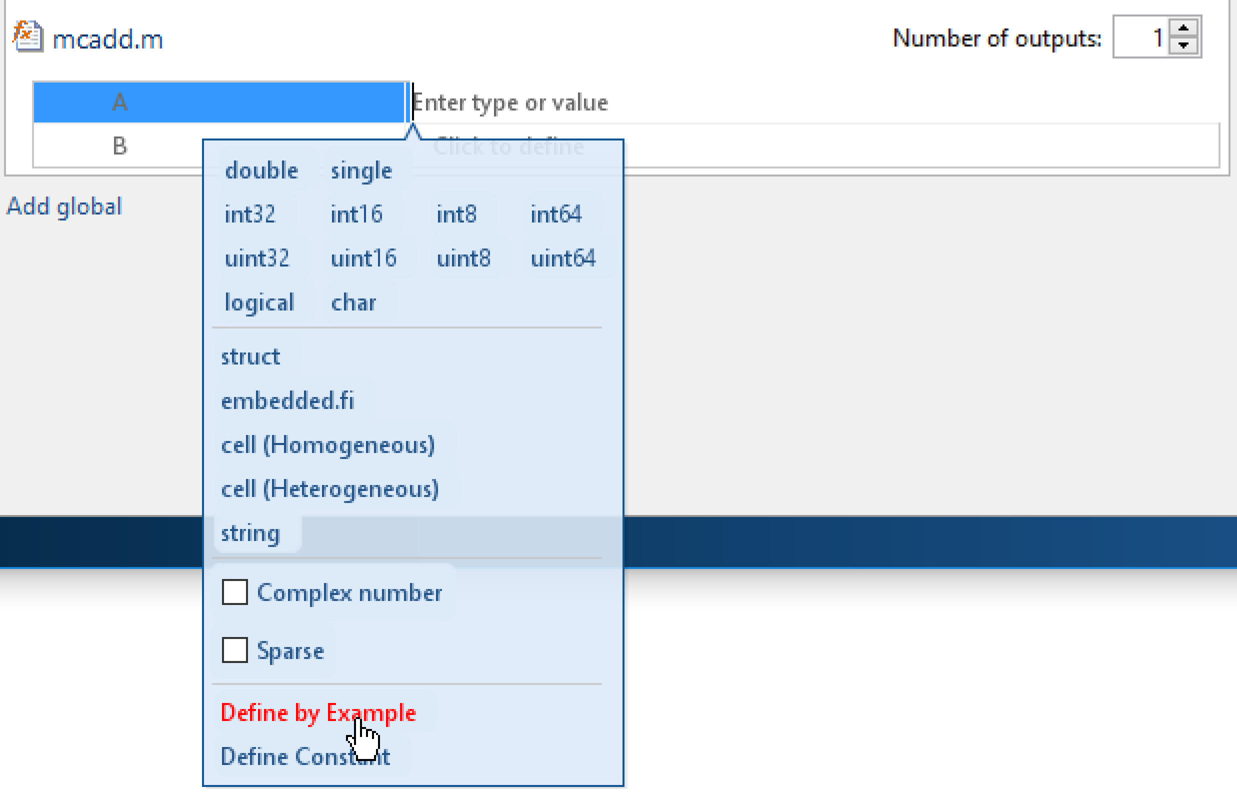
Define Input Parameter by Example by Using the App

In this section...

"Define an Input Parameter by Example" on page 24-6
"Specify Input Parameters by Example" on page 24-7
"Specify a String Scalar Input Parameter by Example" on page 24-8
"Specify a Structure Type Input Parameter by Example" on page 24-8
"Specify a Cell Array Type Input Parameter by Example" on page 24-9
"Specify an Enumerated Type Input Parameter by Example" on page 24-10
"Specify an Object Input Type Parameter by Example" on page 24-11
"Specify a Fixed-Point Input Parameter by Example" on page 24-12
"Specify an Input from an Entry-Point Function Output Type" on page 24-13

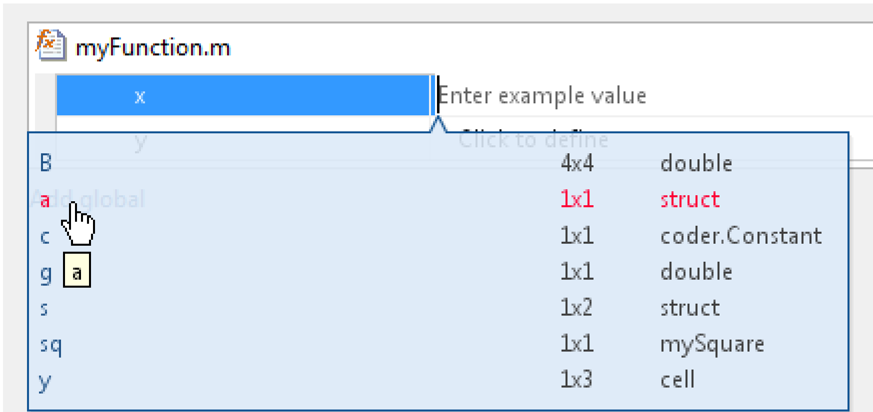
Define an Input Parameter by Example

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.



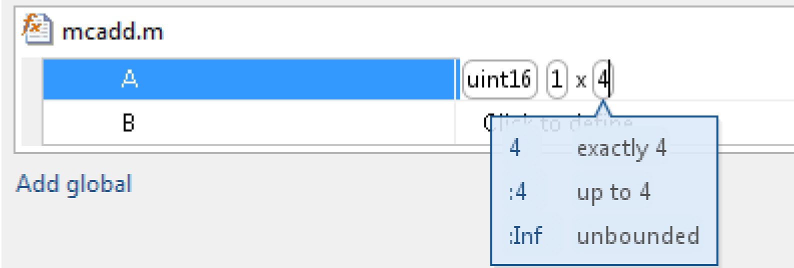
Specify Input Parameters by Example

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`.

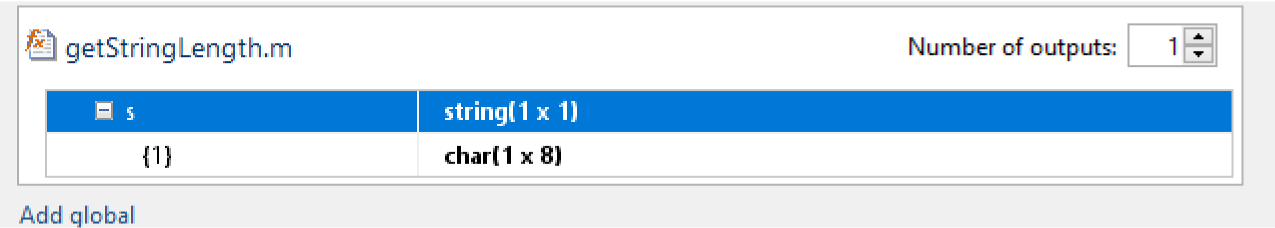
Specify a String Scalar Input Parameter by Example

This example shows how to specify a string scalar type by providing an example string.

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:

   ```
   "mystring"
   ```

   The input parameter is a 1-by-1 string array (string scalar) that contains a 1-by-8 character vector.



5. To make the string variable-size, click the second dimension.

   - To specify that the second dimension is unbounded, select `:Inf`.
   - To specify that the second dimension has an upper bound, enter the upper bound, for example `8`. Then, select `:8`.

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 24-16.

Specify a Cell Array Type Input Parameter by Example

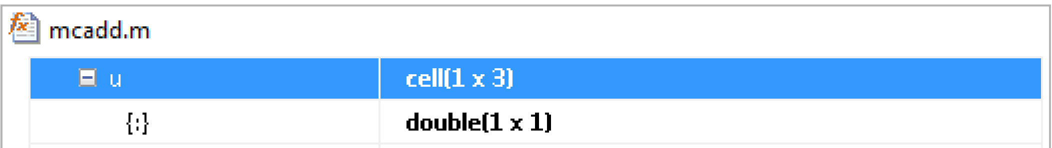
This example shows how to specify a cell array input by example. When you define a cell array by example, the app determines whether the cell array is homogeneous or heterogeneous. See “Code Generation for Cell Arrays” on page 9-2. If you want to control whether the cell array is homogeneous or heterogeneous, specify the cell array by type. See “Specify a Cell Array Input Parameter” on page 24-18.

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.
3. Select **Define by Example**.
4. In the field to the right of the parameter, enter an example cell array.

   - If all cell array elements have the same properties, the cell array is homogeneous. For example, enter:

     ```
     {1 2 3}
     ```

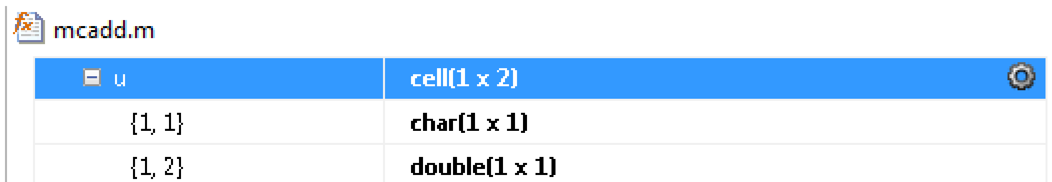
     The input is a 1x3 cell array. The type of each element is `double(1x1)`.

     The 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}
     ```

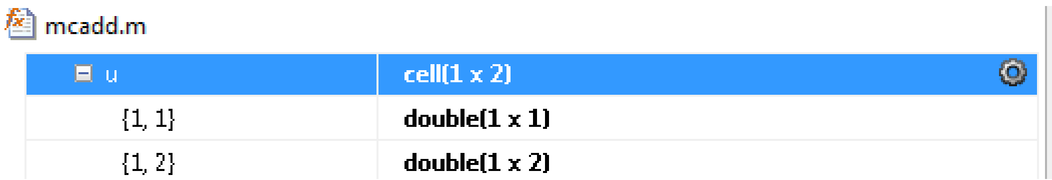
     The input is a 1x2 cell array. For a heterogeneous cell array, the app lists each element. The type of the first element is `char(1x1)`. The type of the second element is `double(1x1)`.

   - 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)`.



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 24-18.
- Right-click the variable. Select **Homogeneous**. Specify that the elements are 1x:2 double.

If you use `coder.typeof` to specify that the example cell array is variable size, the app makes the cell array homogeneous. For example, for the example input, enter:

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

The app determines that the input is a 1x:3 homogeneous cell array whose elements are 1x:2 double.

To modify the type definition, see “Specify a Cell Array Input Parameter” on page 24-18.

Specify an Enumerated Type Input Parameter by Example

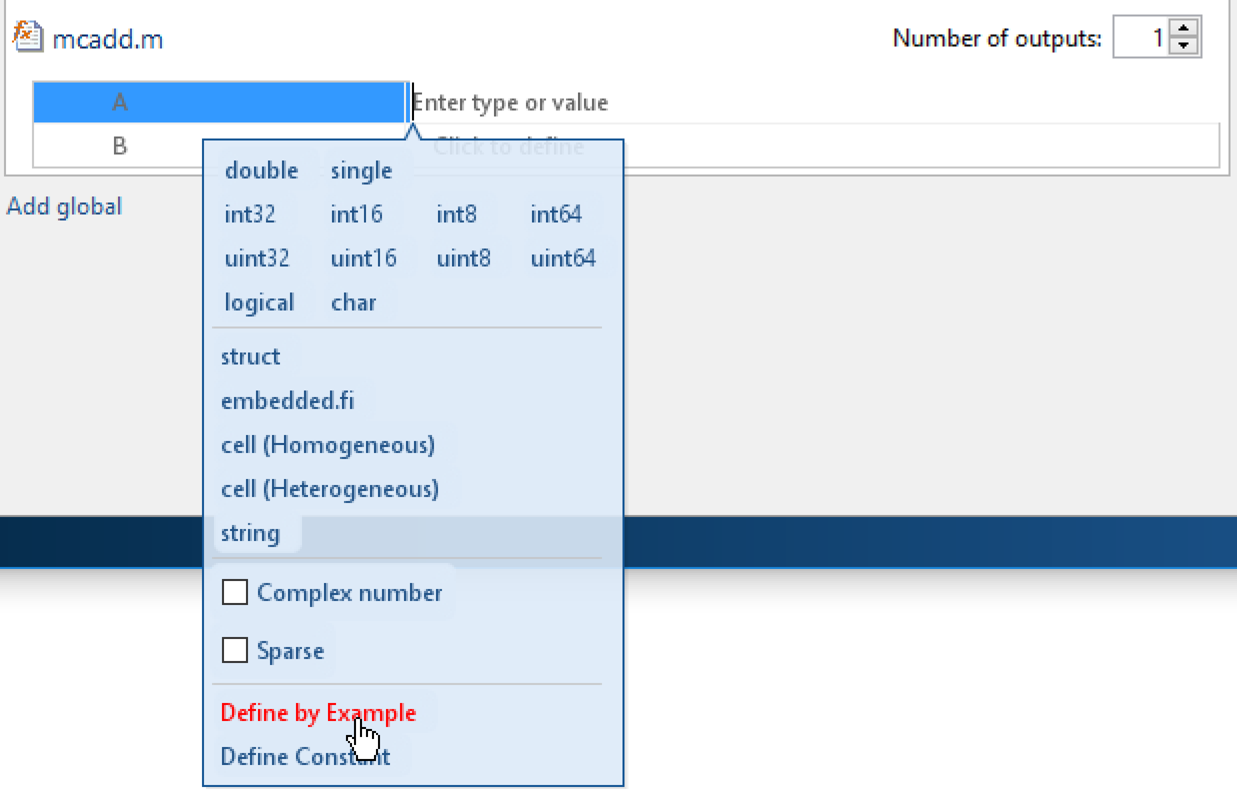
This example shows how to specify that an input uses the enumerated type `MyColors`.

Suppose that `MyColors.m` is on the MATLAB path.

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

To specify that an input has the enumerated type `MyColors`:

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.



3 Select **Define by Example**.
4 In the field to the right of the parameter, enter the 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 15-30.

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)
```

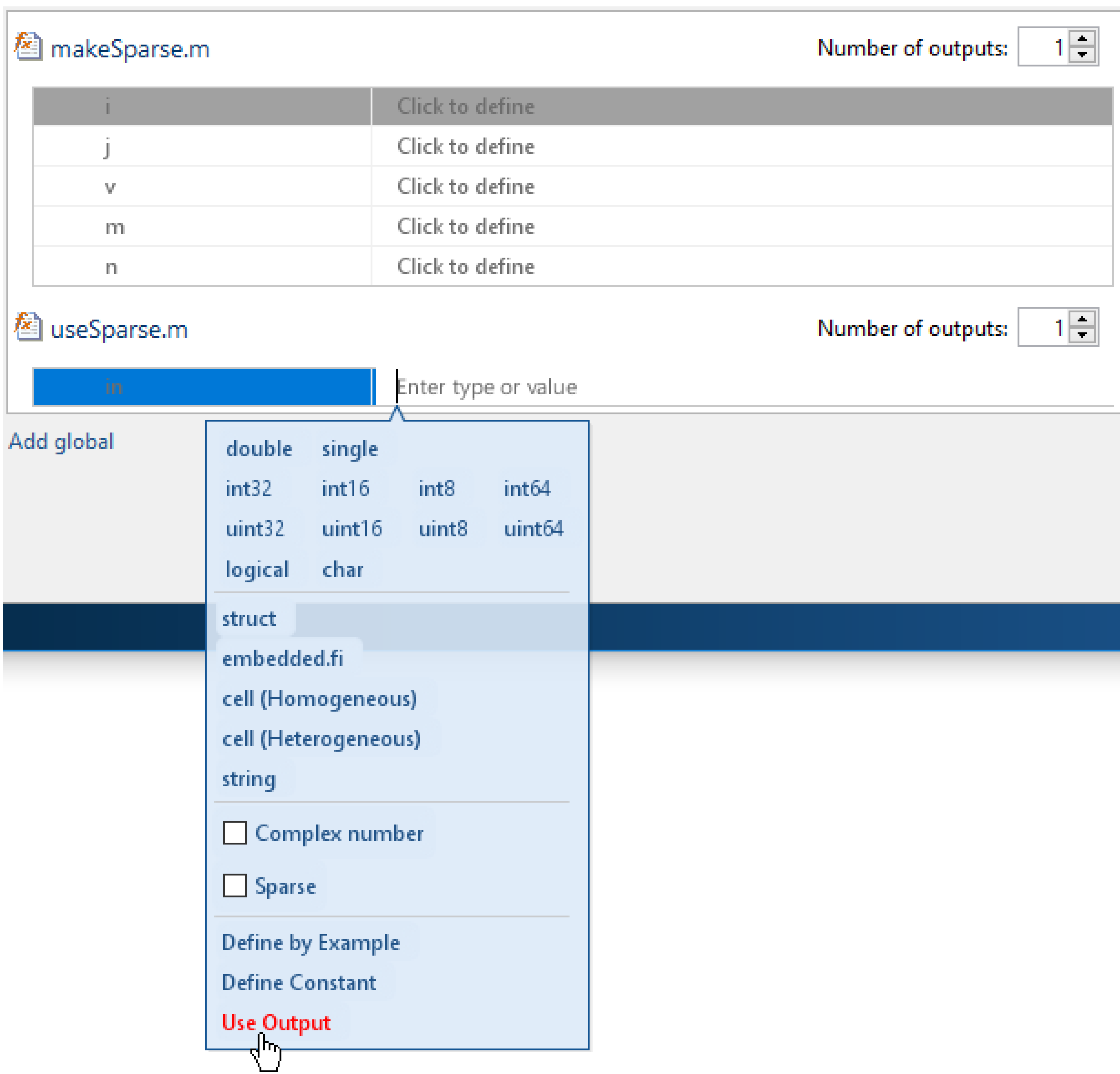
The app sets the type of input `u` to `fi(1x1)`. By default, if you do not specify a local `fimath`, the app uses the default `fimath`. See “fimath for Sharing Arithmetic Rules” (Fixed-Point Designer).

Optionally, modify the fixed-point properties or the size of the input. See “Specify a Fixed-Point Input Parameter” on page 24-16 and “Define or Edit Input Parameter Type by Using the App” on page 24-14.

Specify an Input from an Entry-Point Function Output Type

When generating code for multiple entry-point functions, you can use the output type from one entry-point function as the input type to another entry-point function. For more information, see “Pass an Entry-Point Function Output as an Input” on page 27-85.

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 and select **Use Output**.



3 Select the name of the entry-point function and the corresponding output parameter from which to define the input type.

The MATLAB Coder app is not supported in MATLAB Online.

Define or Edit Input Parameter Type by Using the App

In this section...
"Define or Edit an Input Parameter Type" on page 24-14
"Specify a String Scalar Input Parameter" on page 24-15
"Specify an Enumerated Type Input Parameter" on page 24-15
"Specify a Fixed-Point Input Parameter" on page 24-16
"Specify a Structure Input Parameter" on page 24-16
"Specify a Cell Array Input Parameter" on page 24-18

Define or Edit an Input Parameter Type

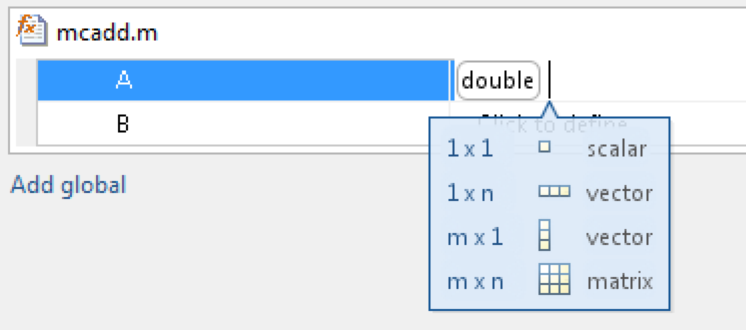
The following procedure shows you how to define or edit `double`, `single`, `int64`, `int32`, `int16`, `int8`, `uint64`, `uint32`, `uint16`, `uint8`, `logical`, and `char` types.

For more information about defining other types, see the information in this table.

Input Type	Link
A string scalar (1-by-1 string array)	"Specify a String Scalar Input Parameter" on page 24-15
A structure (**struct**)	"Specify a Structure Input Parameter" on page 24-16
A cell array (**cell (Homogeneous)** or **cell (Heterogeneous)**)	"Specify a Cell Array Input Parameter" on page 24-18
A fixed-point data type (**embedded.fi**)	"Specify a Fixed-Point Input Parameter" on page 24-16
An input by example (**Define by Example**)	"Define Input Parameter by Example by Using the App" on page 24-6
A constant (**Define Constant**)	"Define Constant Input Parameters Using the App" on page 24-23

1. Click the field to the right of the input parameter name.
2. Optionally, for numeric types, to make the parameter a complex type, select the **Complex number** check box.
3. Select the input type.

   The app displays the selected type. It displays and 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 a String Scalar Input Parameter

To specify that an input is a string scalar:

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 **string**. Then select `1x1 scalar`.

The type is a 1-by-1 string array (string scalar) that contains a character vector.

4 To specify the size of the character vector, click the field to the right of the string array element `{1}`. Select **char**. Then, select `1xn vector` and enter the size.

5 To make the string variable-size, click the second dimension.

- To specify that the second dimension is unbounded, select `:Inf`.
- To specify that the second dimension has an upper bound, enter the upper bound, for example `8`. Then, select `:8`.

Specify an Enumerated Type Input Parameter

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

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 24-17.
7. Add fields to the structure. Specify the class, size, and complexity of the fields. See “Add a Field to a Structure” on page 24-18.

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 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.

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 9-2. A homogeneous cell array is represented as an array in the generated code. All

elements have the same properties. A heterogeneous cell array is represented as a structure in the generated code. Elements can have different properties.

Specify a Homogeneous Cell Array

1. On the **Define Input Types** page, click **Let me enter input or global types directly**.
2. Click the field to the right of the input parameter that you want to define.
3. Select **cell (Homogeneous)**.

   The app displays the selected type, `cell`. The app displays the size options.
4. From the list, select whether your input is a scalar, a `1 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 24-22
8. Optionally, specify properties for the structure that represents the cell array in the generated code. See “Set Structure Properties for a Heterogeneous Cell Array” on page 24-19.

Set Structure Properties for a Heterogeneous Cell Array

A heterogeneous cell array is represented as a structure in the generated code. You can specify the properties for the structure that represents the cell array.

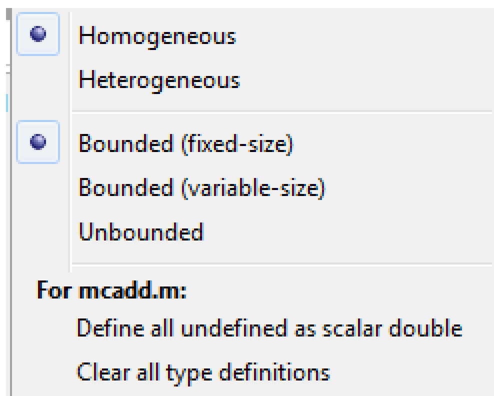
1. Click to the right of the cell array definition. Optionally click ⚙.
2. In the dialog box, specify properties for the structure in the generated code.

Property	Description
C type definition name	Name for the structure type in the generated code.
Type definition is externally defined	Default: `No` — type definition is not externally defined. If you select `Yes` to declare an externally defined structure, the app does not 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.

The MATLAB Coder app is not supported in MATLAB Online.

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 27-60.

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 24-26.

The MATLAB Coder app is not supported in MATLAB Online.

Specify Global Variable Type and Initial Value Using the App

In this section...

"Why Specify a Type Definition for Global Variables?" on page 24-26
"Specify a Global Variable Type" on page 24-26
"Define a Global Variable by Example" on page 24-26
"Define or Edit Global Variable Type" on page 24-27
"Define Global Variable Initial Value" on page 24-27
"Define Global Variable Constant Value" on page 24-28
"Remove Global Variables" on page 24-28

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.

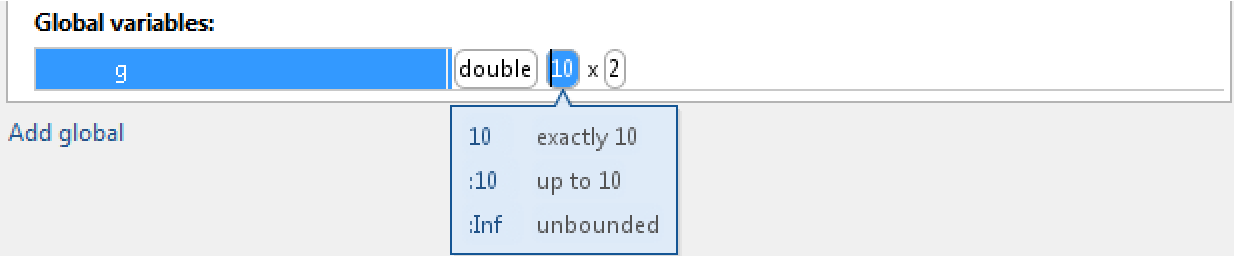
Specify a Global Variable Type

1. Specify the type of each global variable using one of the following methods:
   - Define by example on page 24-26
   - Define type on page 24-27
2. Define an initial value on page 24-27 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`.



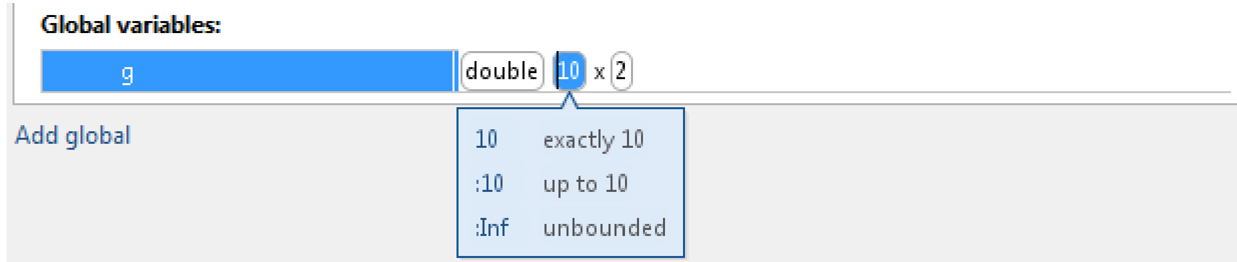
You can specify:

- 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`.

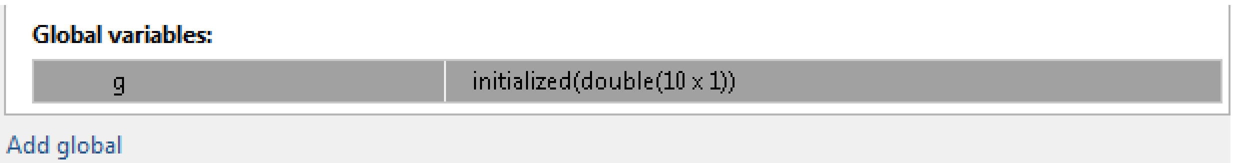
Define Global Variable Initial Value

- “Define Initial Value Before Defining Type” on page 24-28
- “Define Initial Value After Defining Type” on page 24-28

Define Initial Value Before Defining Type

1. Click the field to the right of the global variable.
2. Select `Define Initial Value`.
3. Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable. Because you did not define the type of the global variable before you defined its initial value, MATLAB Coder uses the initial value type as the global variable type.

   The project shows that the global variable is initialized.

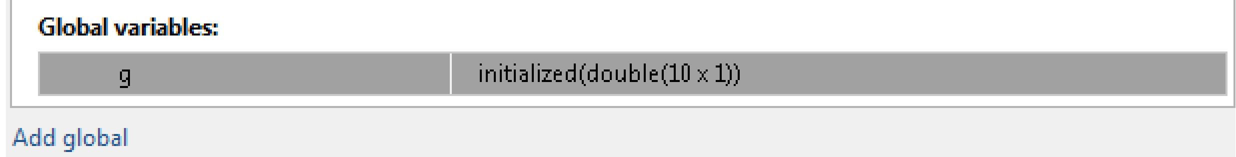
   

   If you change the type of a global variable after defining its initial value, you must redefine the initial value.

Define Initial Value After Defining Type

- Click the type field of a predefined global variable.
- Select `Define Initial Value`.
- Enter a MATLAB expression. MATLAB Coder software uses the value of the specified MATLAB expression as the value of the global variable.

  The project shows that the global variable is initialized.

Define Global Variable Constant Value

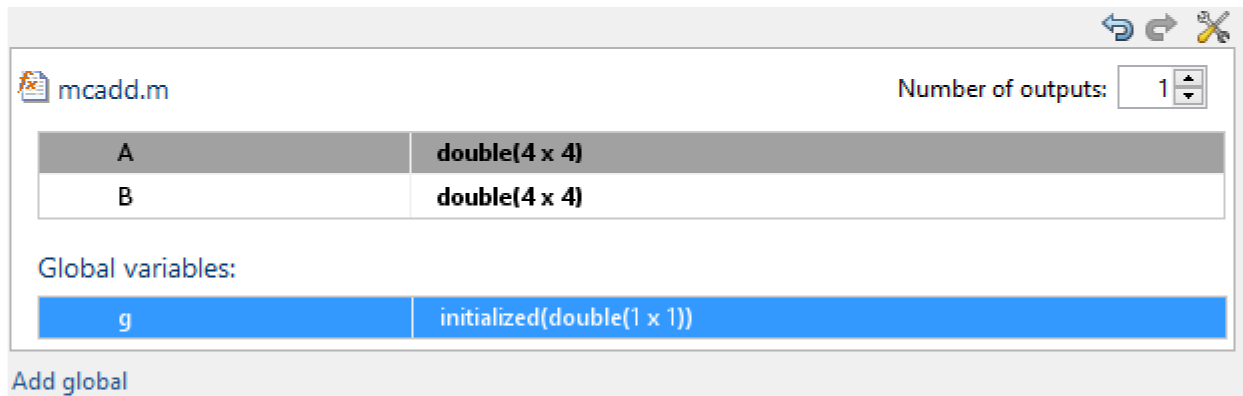
1. Click the field to the right of the global variable.
2. Select `Define Constant Value`.
3. In the field to the right of the global variable, enter a MATLAB expression.

Remove Global Variables

1. Right-click the global variable.
2. From the menu, select **Remove Global**.

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 .



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.

The MATLAB Coder app is not supported in MATLAB Online.

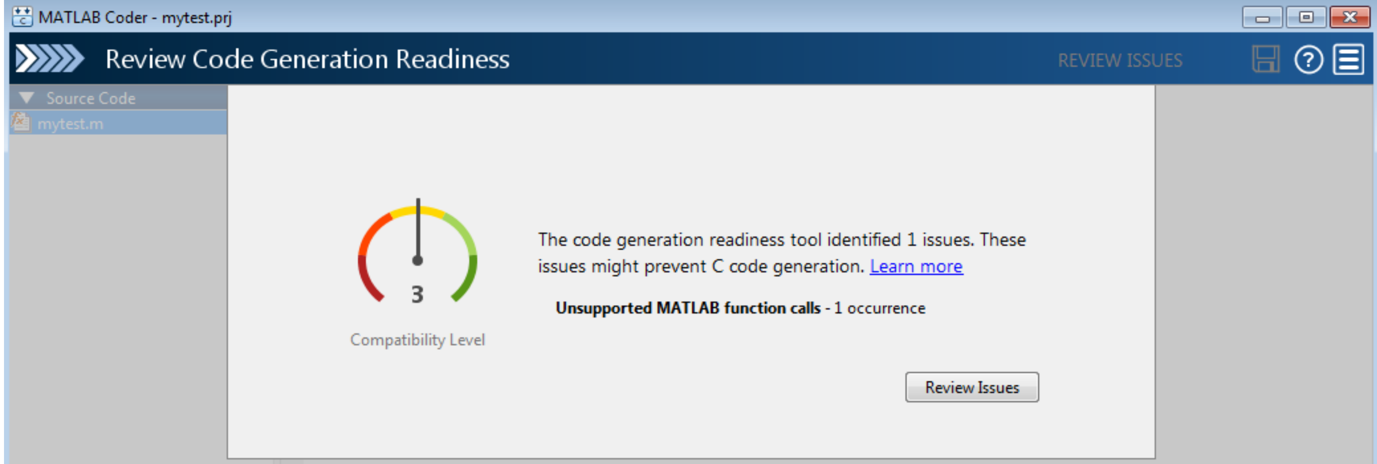
See Also

Related Examples

- "Customize Keyboard Shortcuts"

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.



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.

Code generation readiness screening is not supported in MATLAB Online.

See Also

More About

- "Slow Operations in MATLAB Coder App" on page 24-31
- "Automated Fixed-Point Conversion" on page 21-67

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.

See Also

More About

- "Code Generation Readiness Screening in the MATLAB Coder App" on page 24-30

Unable to Open a MATLAB Coder Project

When you open a project from a different release, if necessary, the MATLAB Coder app updates the project file so that the format is compatible with the release that you are using. Before the app updates the project file, it creates a backup file with the name `project_name.prj.bak`. For example, the backup file name for `myproject.prj` is `myproject.prj.bak`. If the backup file exists, the app inserts an integer between the `prj` and `bak` extensions to make the file name unique. For example, if `myproject.prj.bak` exists, the app creates the backup file `myproject.prj.2.bak`.

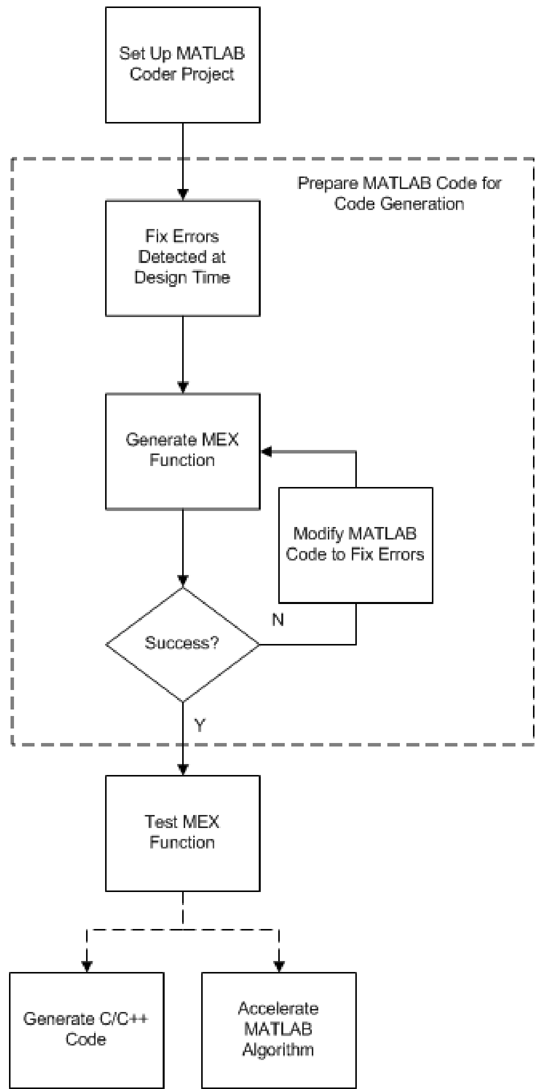
If the project file is from a release before R2015a, the app also displays a message about the project file update and backup. To use the project in a release before R2015a, use the backup project file instead of the updated project file.

To use a backup project file, remove the extensions that follow the `prj` extension. For example, rename `myproject.prj.2.bak` to `myproject.prj`. If you use the backup project file in the release that created it, the project is the same as the original project. If you use the backup project file in a different release than the one that created it, you can possibly lose some information. For example, if you open a project file in a release that does not recognize a setting in the file, that setting is lost. For best results, open the backup project file in the release in which you created it.

Preparing MATLAB Code for C/C++ Code Generation

- “Workflow for Preparing MATLAB Code for Code Generation” on page 25-2
- “Fixing Errors Detected at Design Time” on page 25-3
- “Using the Code Analyzer” on page 25-4
- “Check Code with the Code Analyzer” on page 25-5
- “Check Code by Using the Code Generation Readiness Tool” on page 25-7
- “Code Generation Readiness Tool” on page 25-8
- “Unable to Determine Code Generation Readiness” on page 25-13
- “Generate MEX Functions by Using the MATLAB Coder App” on page 25-14
- “Generate MEX Functions at the Command Line” on page 25-18
- “Fix Errors Detected at Code Generation Time” on page 25-19
- “Running MEX Functions” on page 25-20
- “Debugging Strategies” on page 25-21
- “Collect and View Line Execution Counts for Your MATLAB Code” on page 25-22

Workflow for Preparing MATLAB Code for Code Generation



See Also

- “Set Up a MATLAB Coder Project” on page 24-2
- “Fixing Errors Detected at Design Time” on page 25-3
- “Generate MEX Functions by Using the MATLAB Coder App” on page 25-14
- “Fix Errors Detected at Code Generation Time” on page 25-19
- “Workflow for Testing MEX Functions in MATLAB” on page 26-3
- “Accelerate MATLAB Algorithms” on page 32-6

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 25-5
- "Check Code by Using the Code Generation Readiness Tool" on page 25-7
- "Debugging Strategies" on page 25-21

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 programming for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of code generation analyzer messages is available in the MATLAB Code Analyzer preferences. For more information, see "Running the Code Analyzer Report".

Note The code analyzer might not detect all code generation compliance issues in your MATLAB code. 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" on page 25-7
"Run Code Generation Readiness Tool from the Current Folder Browser" on page 25-7
"Run the Code Generation Readiness Tool Using the MATLAB Coder App" on page 25-7

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 25-8.

The Code Generation Readiness Tool is not supported in MATLAB Online.

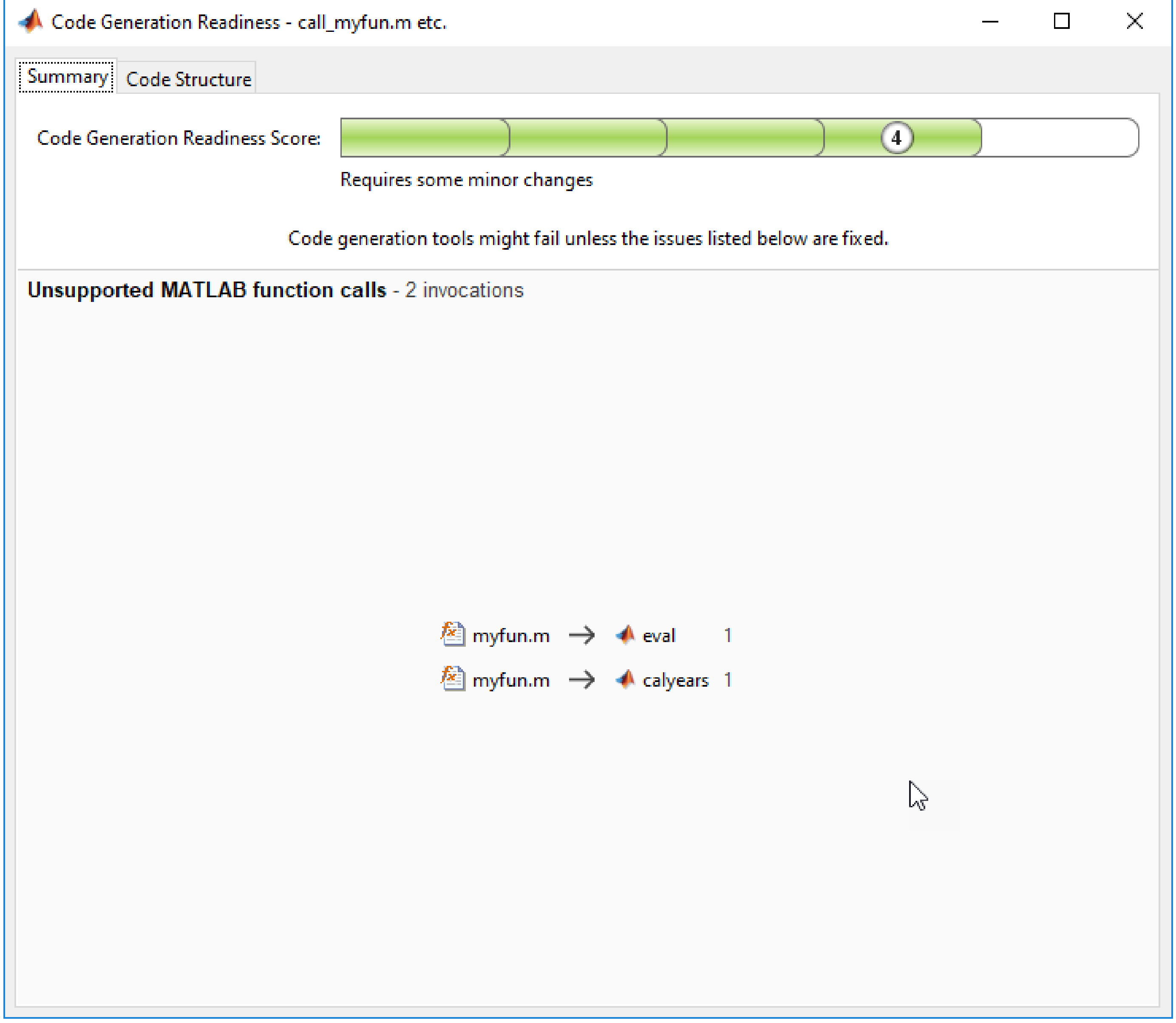
Code Generation Readiness Tool

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.

The code generation readiness tool does not report functions that the code generator automatically treats as extrinsic. Examples of such functions are `plot`, `disp`, and `figure`. See "Use MATLAB Engine to Execute a Function Call in Generated Code" on page 20-8.

The Code Generation Readiness Tool is not supported in MATLAB Online.

Summary Tab



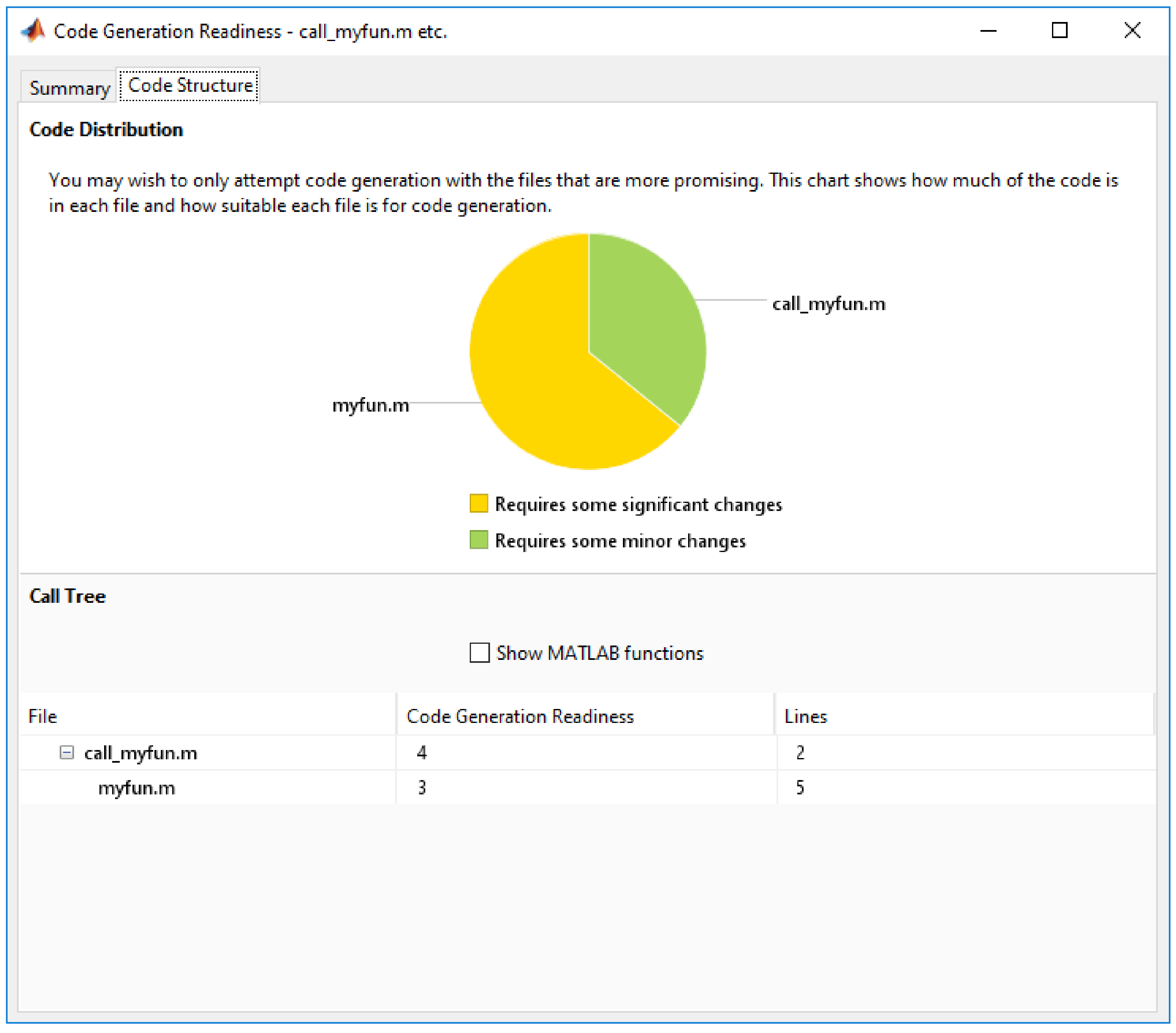
The **Summary** tab provides a **Code Generation Readiness Score**, which ranges from 1 to 5. A score of 1 indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of 5 indicates that the tool does not detect code generation issues; the code is ready to use with minimal or no changes.

On this tab, the tool also displays information about:

- MATLAB syntax issues. These issues are reported in the MATLAB editor. To learn more about the issues and how to fix them, use the Code Analyzer.
- Unsupported MATLAB function calls.
- Unsupported MATLAB language features.

- Unsupported data types.

Code Structure Tab



If the code that you are checking calls other MATLAB functions, or you are checking multiple entry-point functions, the tool displays the **Code Structure Tab**.

This tab displays information about the relative size of each file and how suitable each file is for code generation.

Code Distribution

The **Code Distribution** pane displays a pie chart that shows the relative sizes of the files and how suitable each file is for code generation. During the planning phase of a project, you can use this

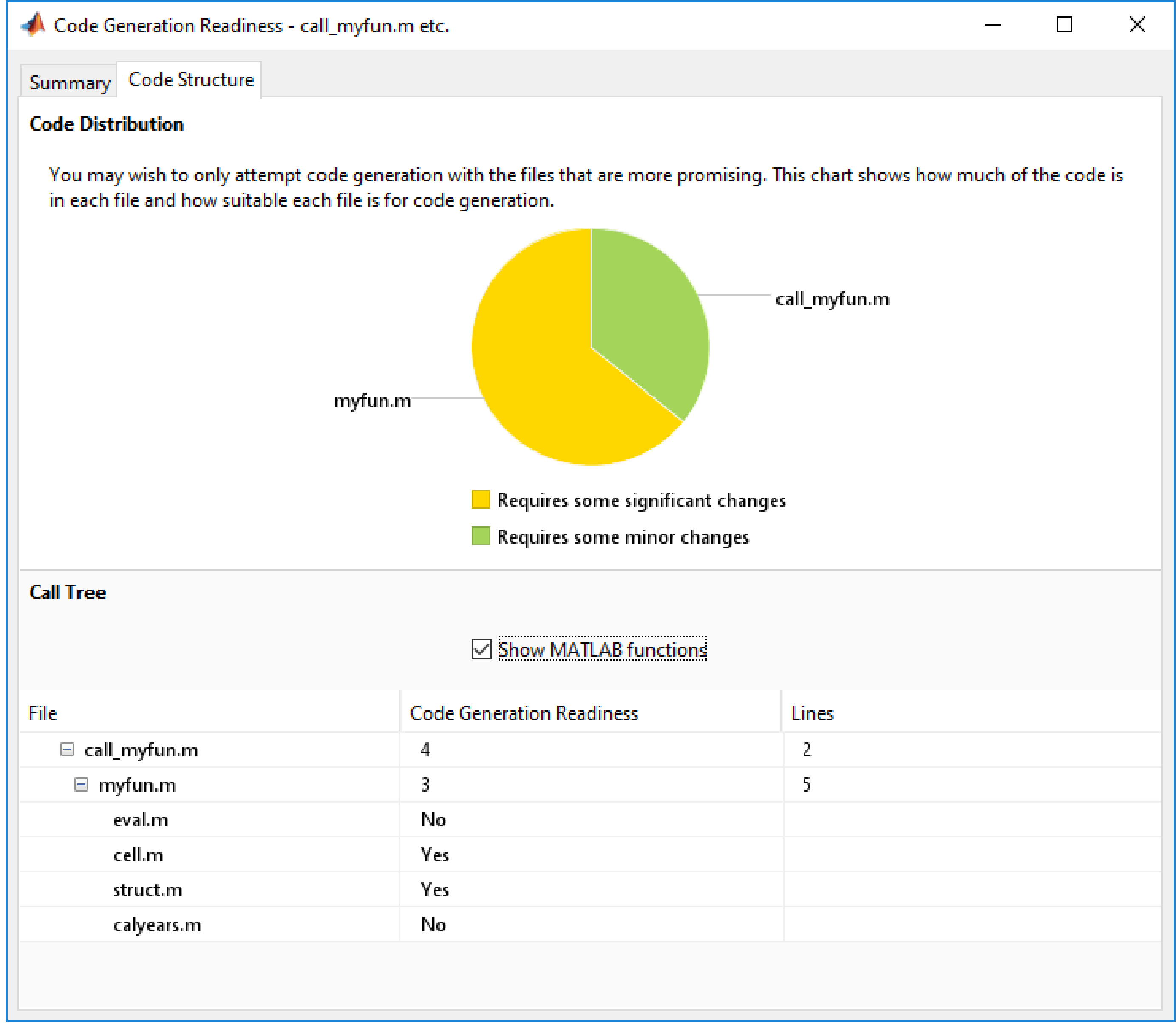
information for estimation and scheduling. If the report indicates that multiple files are not suitable for code generation, consider fixing files that require minor changes before addressing files with significant issues.

Call Tree

The **Call Tree** pane displays information about the nesting of function calls. For each called function, the report provides a **Code Generation Readiness** score, which ranges from `1` to `5`. A score of `1` indicates that the tool detects issues that require extensive changes to the MATLAB code to make it suitable for code generation. A score of `5` indicates that the tool does not detect code generation issues. The code is ready to use with minimal or no changes. The report also lists the number of lines of code in each file.

Show MATLAB Functions

If you select **Show MATLAB Functions**, the report also lists the MATLAB functions that your function calls. For each of these MATLAB functions, if code generation supports the function, the report sets **Code Generation Readiness** to `Yes`.



See Also

Related Examples

- "Check Code by Using the Code Generation Readiness Tool" on page 25-7

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" on page 25-14
"Generate a MEX Function Using the MATLAB Coder App" on page 25-14
"Configure Project Settings" on page 25-16
"Build a MATLAB Coder Project" on page 25-16
"See Also" on page 25-17

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 24-2
2	Specify the build configuration parameters. Set **Build type** to MEX.	"Configure Project Settings" on page 25-16
3	Build the project.	"Build a MATLAB Coder Project" on page 25-16

The MATLAB Coder app is not supported in MATLAB Online. To generate MEX functions in MATLAB Online, use the codegen command.

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 27-76.

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.

Certain configuration parameters are relevant for both MEX and standalone code generation. If you enable any of these 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`.

See Also

- “Using the MATLAB Coder App” on page 27-106
- “How to Disable Inlining Globally Using the MATLAB Coder App” on page 27-113
- “Disabling Run-Time Checks Using the MATLAB Coder App” on page 32-13

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 27-13

See Also

Related Examples

- “Configure Build Settings” on page 27-13
- “Generate C Code by 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 27-76
3	Fix errors detected by the code analyzer.	“Fixing Errors Detected at Design Time” on page 25-3
4	Specify build configuration parameters.	“Specify Build Configuration Parameters” on page 27-17
5	Specify properties of primary function inputs.	“Specify Properties of Entry-Point Function Inputs” on page 27-43
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 27-76.

See Also

Related Examples

- “Specify Properties of Entry-Point Function Inputs” on page 27-43
- “Accelerate MATLAB Algorithm by Generating MEX Function”

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 25-21.

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 20-2. 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 28-7
- “Why Test MEX Functions in MATLAB?” on page 26-2
- “When to Generate Code from MATLAB Algorithms” on page 2-2
- “Debugging Strategies” on page 25-21
- “Using the coder.extrinsic Construct” on page 20-9

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 Toolbox™ function or System object, to run the MEX function, you must have a Computer Vision Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

Debug 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. You can also use the fprintf function to inspect the contents of your MEX function variables.

Debug MEX Functions by Using a C/C++ Debugger

To debug your MEX functions by using a C/C++ debugger, set the MEX configuration object property EnableDebugging to 1.

```
cfg = coder.config('mex');
cfg.EnableDebugging = 1;
codegen -config cfg foo_mex
```

Alternatively, you can debug your MEX function by executing this command:

```
codegen -g foo_mex
```

The foo_mex file is the MEX file that you intend to debug. You can debug this file by using a C or C++ debugger. For more information on debugging by using a C/C++ debugger on a Microsoft Windows platform, see “Debug on Microsoft Windows Platforms”.

For more information on debugging by using a C/C++ debugger on a Linux® or Mac platform, see “Debug on Linux Platforms” or “Debug on Mac Platforms”.

Debugging Strategies

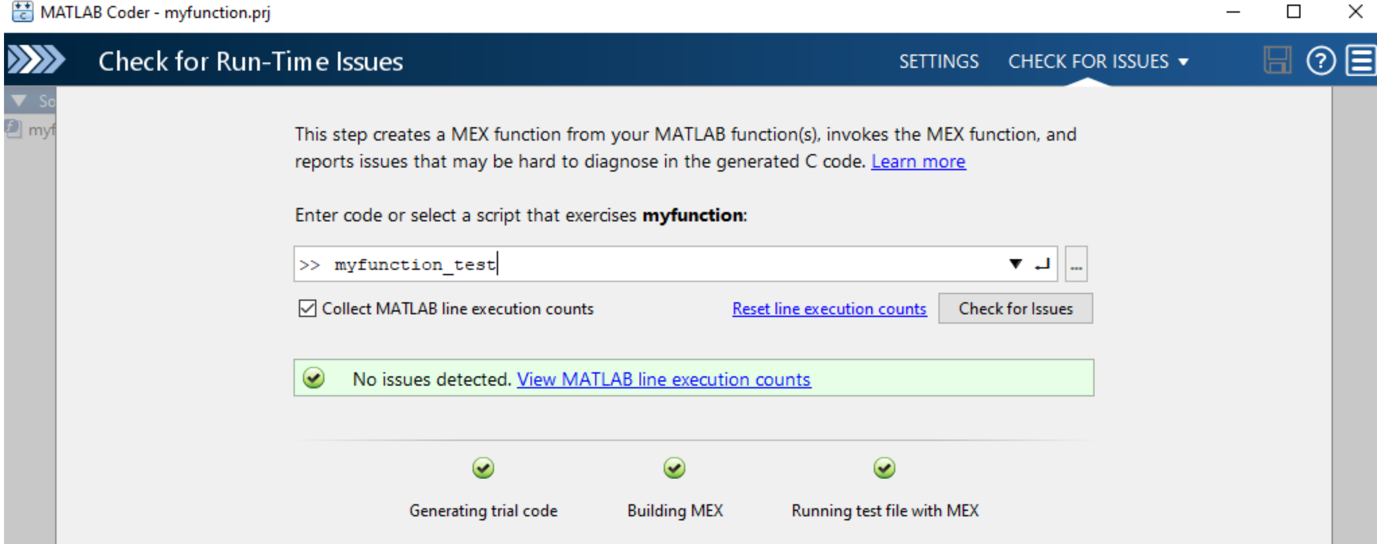
Before you perform code verification, choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of many MATLAB files that call each other's functions. The following table describes two general strategies, each of which has advantages 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 “Using the coder.extrinsic Construct” on page 20-9. **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 20-11).

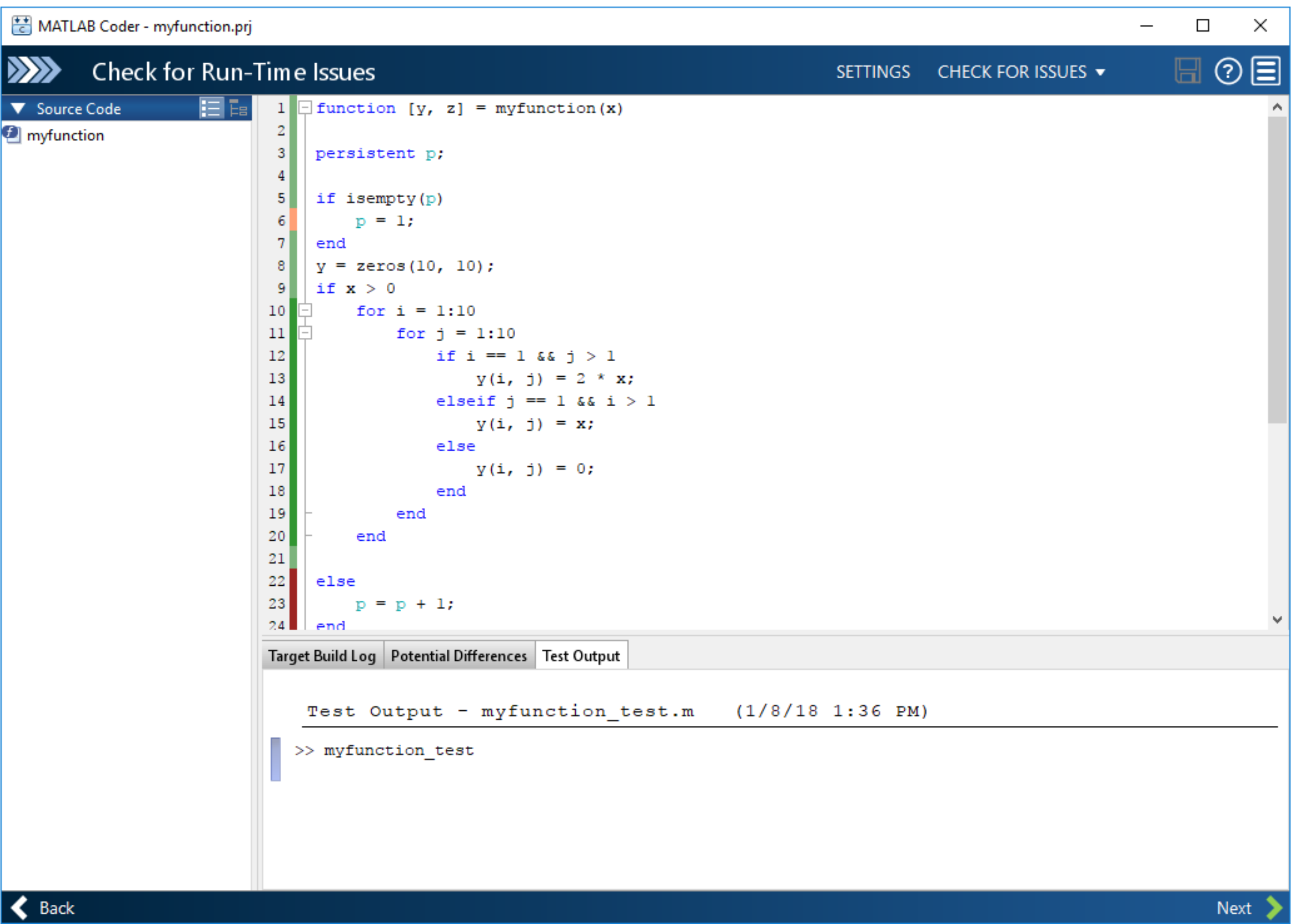
Collect and View Line Execution Counts for Your MATLAB Code

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**.



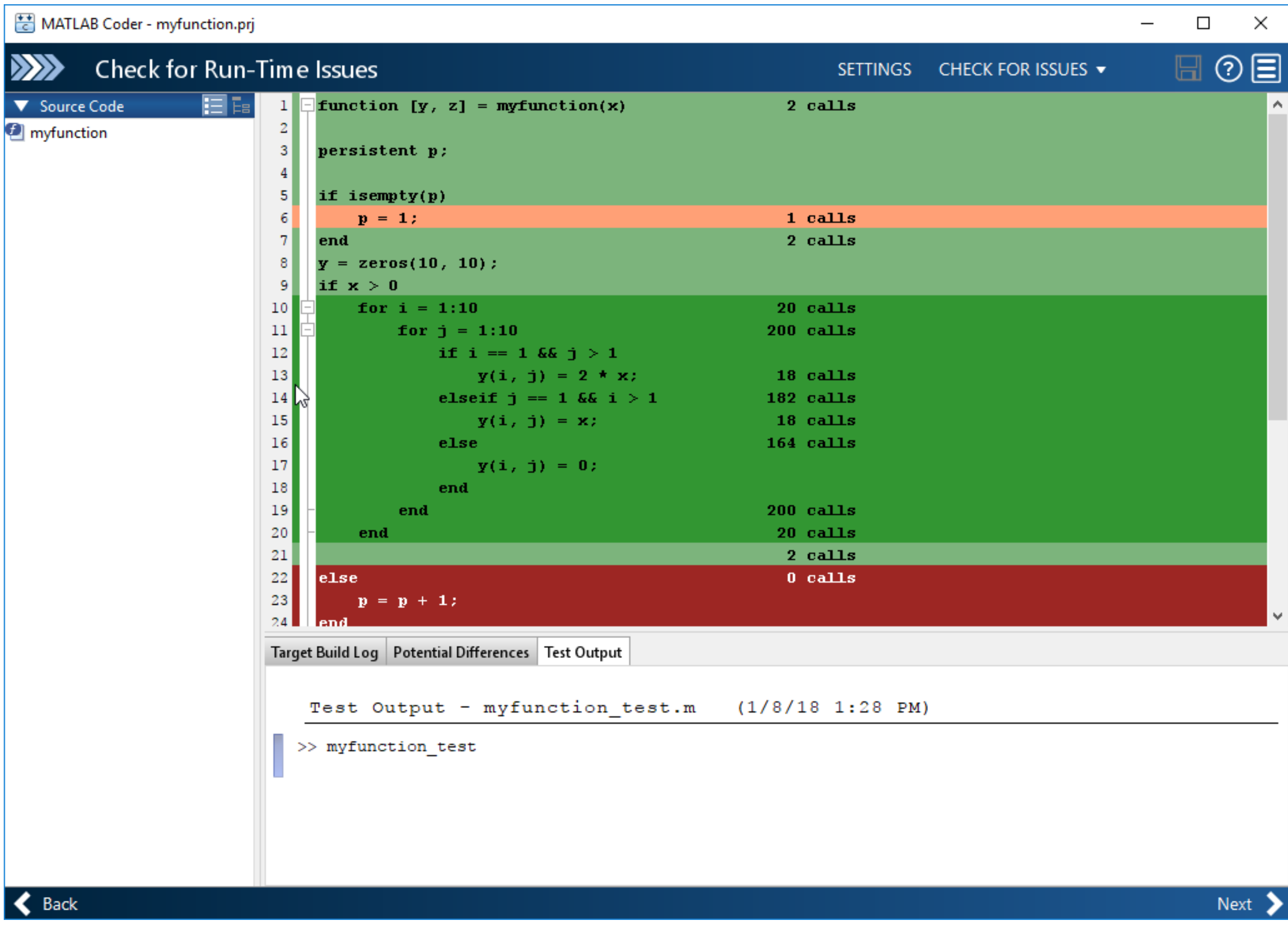
In the app editor, the app displays a color coded bar to the left of your MATLAB code.



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.
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you place your pointer 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.



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.

If you check for run-time issues multiple times, the line execution counts accumulate. To set the counts to zero, click **Reset line execution counts**.

The MATLAB Coder app is not supported in MATLAB Online.

See Also

Related Examples

- “Check for Run-Time Issues by Using the App” on page 26-5

More About

- “Why Test MEX Functions in MATLAB?” on page 26-2

Testing MEX Functions in MATLAB

- “Why Test MEX Functions in MATLAB?” on page 26-2
- “Workflow for Testing MEX Functions in MATLAB” on page 26-3
- “Running MEX Functions” on page 26-4
- “Check for Run-Time Issues by Using the App” on page 26-5
- “Verify MEX Functions in the MATLAB Coder App” on page 26-7
- “Verify MEX Functions at the Command Line” on page 26-8
- “Debug Run-Time Errors” on page 26-9
- “Using MEX Functions That MATLAB Coder Generates” on page 26-11

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 26-5 and “Verify MEX Functions in the MATLAB Coder App” on page 26-7. For more information about how to test a MEX function at the command line, see “Verify MEX Functions at the Command Line” on page 26-8.

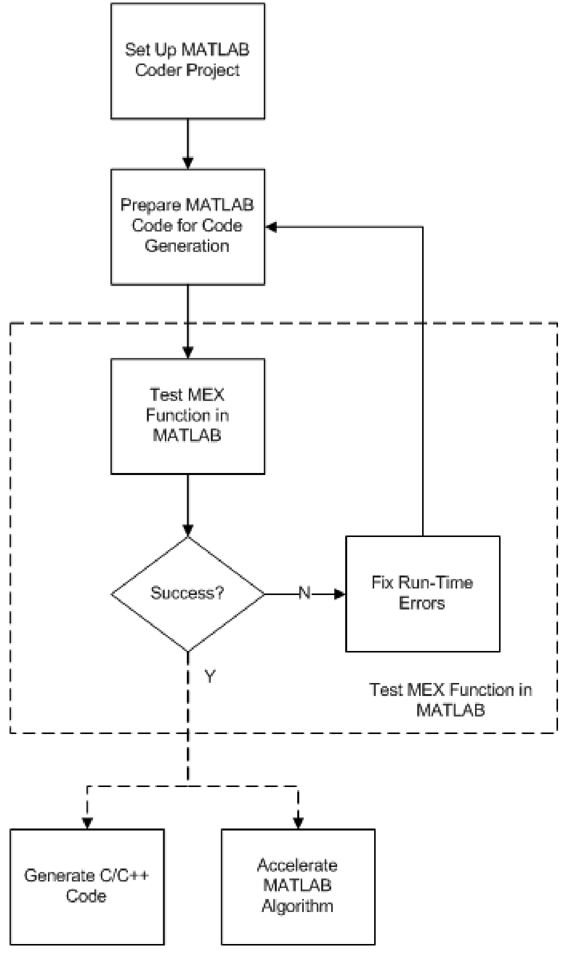
Running the MEX function in MATLAB before generating code enables you to detect and fix run-time errors that are much harder to diagnose in the generated code. If you encounter run-time errors in your MATLAB functions, fix them before generating code. See “Fix Errors Detected at Code Generation Time” on page 25-19 and “Debug Run-Time Errors” on page 26-9.

When you run your MEX function in MATLAB, by default, the following run-time checks execute:

- Memory integrity checks. These checks perform array bounds checking, dimension checking, and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution and provides a diagnostic message.
- Responsiveness checks in code generated for MATLAB functions. These checks enable periodic checks for **Ctrl+C** breaks in code generated for MATLAB functions, allowing you to terminate execution with **Ctrl+C**.

For more information, see “Control Run-Time Checks” on page 32-12.

Workflow for Testing MEX Functions in MATLAB



See Also

- “Set Up a MATLAB Coder Project” on page 24-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 25-2
- “Why Test MEX Functions in MATLAB?” on page 26-2
- “Debug Run-Time Errors” on page 26-9
- “Accelerate MATLAB Algorithms” on page 32-6

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 Toolbox function or System object, to run the MEX function, you must have a Computer Vision Toolbox license.

When you upgrade MATLAB, before running MEX functions with the new version, rebuild the MEX functions.

Debug 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. You can also use the `fprintf` function to inspect the contents of your MEX function variables.

Debug MEX Functions by Using a C/C++ Debugger

To debug your MEX functions by using a C/C++ debugger, set the MEX configuration object property `EnableDebugging` to `1`.

```
cfg = coder.config('mex');
cfg.EnableDebugging = 1;
codegen -config cfg foo_mex
```

Alternatively, you can debug your MEX function by executing this command:

```
codegen -g foo_mex
```

The `foo_mex` file is the MEX file that you intend to debug. You can debug this file by using a C or C++ debugger. For more information on debugging by using a C/C++ debugger on a Microsoft Windows platform, see “Debug on Microsoft Windows Platforms”.

For more information on debugging by using a C/C++ debugger on a Linux or Mac platform, see “Debug on Linux Platforms” or “Debug on Mac Platforms”.

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 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 **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**.

The MATLAB Coder app is not supported in MATLAB Online.

Collect MATLAB Line Execution Counts

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 25-22.

Disable JIT Compilation for Parallel Loops

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`.

See Also

More About

- “Why Test MEX Functions in MATLAB?” on page 26-2
- “Generate C Code by Using the MATLAB Coder App”
- “Fix Errors Detected at Code Generation Time” on page 25-19
- “Collect and View Line Execution Counts for Your MATLAB Code” on page 25-22
- “Control Run-Time Checks” on page 32-12
- “Verify MEX Functions at the Command Line” on page 26-8

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 script. The test file must be in the same folder as the original entry-point function.

1. On the **Generate Code** page, click **Verify Code**.
2. Type or select the test file name.
3. To run the test file without replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **MATLAB code**. Click **Run Generated Code**.
4. To run the test file, replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **Generated code**. Click **Run Generated Code**.
5. Compare the results of running the original MATLAB function with the results of running the MEX function.

The MATLAB Coder app is not supported in MATLAB Online. To verify MEX functions in MATLAB Online, see “Verify MEX Functions at the Command Line” on page 26-8.

See Also

`codegen` | `coder.runTest`

More About

- “Why Test MEX Functions in MATLAB?” on page 26-2
- “Verify MEX Functions at the Command Line” on page 26-8
- “Unit Test Generated Code with MATLAB Coder” on page 28-27

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 26-2
- “Check for Run-Time Issues by Using the App” on page 26-5
- “Unit Test Generated Code with MATLAB Coder” on page 28-27

Debug Run-Time Errors

In this section...
“Viewing Errors in the Run-Time Stack” on page 26-9
“Handling Run-Time Errors” on page 26-10

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 26-9“Handling Run-Time Errors” on page 26-10.

Viewing Errors in the Run-Time Stack

About the Run-Time Stack

The run-time stack is enabled by default for MEX code generation from MATLAB. 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 32-13.

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 26-9, create a test script containing:

```
try
    mlstack_mex(u)
catch
    % Add your error handling code here
end
```

For more information, see “The try/catch Statement”.

Using MEX Functions That MATLAB Coder Generates

When you specify MEX for the output (build) type, MATLAB Coder generates a binary MATLAB executable (MEX) version of your MATLAB function. You can call the MEX function from MATLAB. See “MEX File Functions”.

How you use the MEX function depends on your goal.

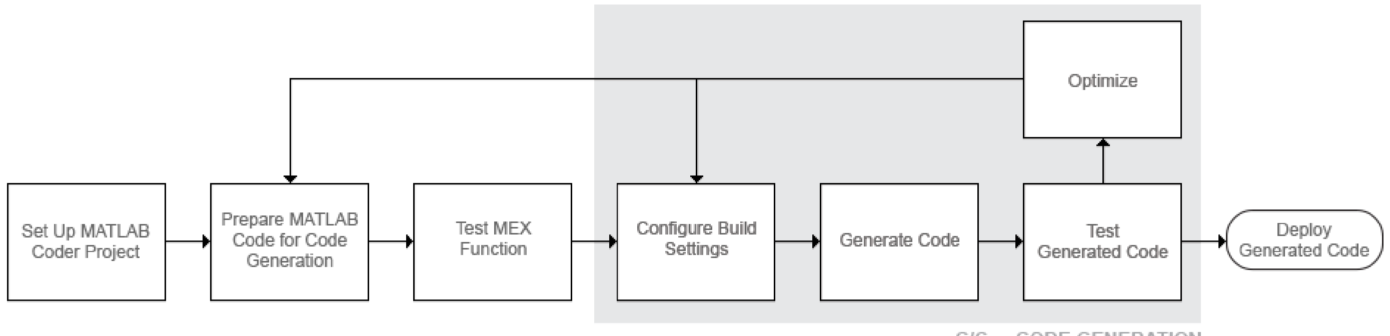
Goal	See
Accelerate your MATLAB function.	“MATLAB Algorithm Acceleration”
Test generated function for functionality and run-time issues.	“Why Test MEX Functions in MATLAB?” on page 26-2
Debug your MEX function.	“Debug Run-Time Errors” on page 26-9

Generating C/C++ Code from MATLAB Code

- “Code Generation Workflow” on page 27-3
- “Generating Standalone C/C++ Executables from MATLAB Code” on page 27-4
- “Configure Build Settings” on page 27-13
- “Specify Configuration Parameters in Command Line Workflow Interactively” on page 27-22
- “Specify Data Types Used in Generated Code” on page 27-24
- “Use Generated Initialize and Terminate Functions” on page 27-25
- “Change the Standard Math Library” on page 27-29
- “Convert codegen Command to Equivalent MATLAB Coder Project” on page 27-30
- “Share Build Configuration Settings” on page 27-33
- “Convert MATLAB Coder Project to MATLAB Script” on page 27-35
- “Preserve Variable Names in Generated Code” on page 27-38
- “Reserved Keywords” on page 27-39
- “Specify Properties of Entry-Point Function Inputs” on page 27-43
- “Specify Cell Array Inputs at the Command Line” on page 27-52
- “Constant Input Checking in MEX Functions” on page 27-57
- “Define Input Properties Programmatically in the MATLAB File” on page 27-60
- “Create and Edit Input Types by Using the Coder Type Editor” on page 27-69
- “Speed Up Compilation by Generating Only Code” on page 27-74
- “Disable Creation of the Code Generation Report” on page 27-75
- “Paths and File Infrastructure Setup” on page 27-76
- “Generate Code for Multiple Entry-Point Functions” on page 27-78
- “Generate One MEX Function for Multiple Signatures” on page 27-82
- “Pass an Entry-Point Function Output as an Input” on page 27-85
- “Generate Code for Global Data” on page 27-88
- “Specify Global Cell Arrays at the Command Line” on page 27-96
- “Generate Code for Enumerations” on page 27-97
- “Generate Code for Variable-Size Data” on page 27-98
- “How MATLAB Coder Partitions Generated Code” on page 27-106
- “Requirements for Signed Integer Representation” on page 27-115
- “Build Process Customization” on page 27-116
- “Run-time Stack Overflow” on page 27-119
- “Compiler and Linker Errors” on page 27-120
- “Pass Structure Arguments by Reference or by Value in Generated Code” on page 27-122

- “Name the C Structure Type to Use With a Global Structure Variable” on page 27-129
- “Generate Code for an LED Control Function That Uses Enumerated Types” on page 27-131
- “Generate Code That Uses N-Dimensional Indexing” on page 27-133
- “Install OpenMP Library on macOS Platform” on page 27-137
- “Generate Code to Detect Edges on Images” on page 27-138
- “C Code Generation for a MATLAB Kalman Filtering Algorithm” on page 27-142
- “Generate Code to Optimize Portfolio by Using Black Litterman Approach” on page 27-151
- “Generate Code for Persistent Variables” on page 27-159
- “Generate Code for Structure Arrays” on page 27-163
- “Add Custom Toolchains to MATLAB® Coder™ Build Process” on page 27-165
- “Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type” on page 27-174
- “Build Process Support for Folder Names with Spaces or Special Characters” on page 27-179

Code Generation Workflow



See Also

- “Set Up a MATLAB Coder Project” on page 24-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 25-2
- “Workflow for Testing MEX Functions in MATLAB” on page 26-3
- “Configure Build Settings” on page 27-13

Generating Standalone C/C++ Executables from MATLAB Code

In this section...
"Generate a C Executable Using the MATLAB Coder App" on page 27-4
"Generate a C Executable at the Command Line" on page 27-10
"Specifying main Functions for C/C++ Executables" on page 27-11
"Specify main Functions" on page 27-11

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 an example 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++ main 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 31-23. 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 "main.h"
#include "coderand.h"
#include "coderand_terminate.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;

  /* The initialize function is being called automatically from your entry-point function. Sc
  /* 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 "main.h"
#include "coderand.h"
#include "coderand_terminate.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()
{
  /* The initialize function is being called automatically from your entry-point function. So
  /* 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 "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 "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_terminate.h"

   int main()
   {
       /* The initialize function is called automatically from the generated entry-point functio
          So, a call to initialize is not included here. */

       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_terminate.h". If your file partitioning method is set to generate one file for all functions, do **not** include "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 31-23. 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 terminate header function in main.c. Otherwise, do not include it in main.c.
- For more information about calling the initialize and terminate functions, see “Use Generated Initialize and Terminate Functions” on page 27-25.

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 27-11.

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 27-18). 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 27-11.) 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" on page 27-13
"Specify a Language for Code Generation" on page 27-15
"Specify Output File Name" on page 27-16
"Specify Output File Locations" on page 27-16
"Parameter Specification Methods" on page 27-17
"Specify Build Configuration Parameters" on page 27-17

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 27-11.

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 27-110.

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.

Certain configuration parameters are relevant for both MEX and standalone code generation. If you enable any of these 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`.

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 27-43.

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 27-15.

To Generate:	Use This Command:
MEX function using the default code generation options	`codegen foo`

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 27-11

To Generate:	Use This Command:

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 27-11

Specify a Language for Code Generation

- "Specify the Language Using the MATLAB Coder App" on page 27-15
- "Specifying the Language Using the Command-Line Interface" on page 27-15

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 27-18
- "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 27-17
`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 27-18
`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 27-21

Specify Build Configuration Parameters

- “Specify Build Configuration Parameters MATLAB Coder App” on page 27-17
- “Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 27-18
- “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 27-21

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.

   Certain configuration parameters are relevant for both MEX and standalone code generation. If you enable any of these 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`.
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 27-19.

   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 27-18
   - Dialog boxes, as described in “Specifying Build Configuration Parameters at the Command Line Using Dialog Boxes” on page 27-21
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');`

To Create...	Use a Command Such As...

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 27-20.

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

After you have created a configuration object, you can modify the properties of the object by using the configuration parameter dialog box. See “Specify Configuration Parameters in Command Line Workflow Interactively” on page 27-22.

Specify Configuration Parameters in Command Line Workflow Interactively

After you have created a code generation configuration object at the command line, you can modify the properties of the object interactively by using the configuration parameter dialog box.

For more information on configuring the code generation process by using configuration objects, see "Configure Build Settings" on page 27-13.

Using the Dialog Box

To create, modify, and use a configuration object, follow these steps:

1. Create a configuration object as described in "Creating Configuration Objects" on page 27-19.

   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:

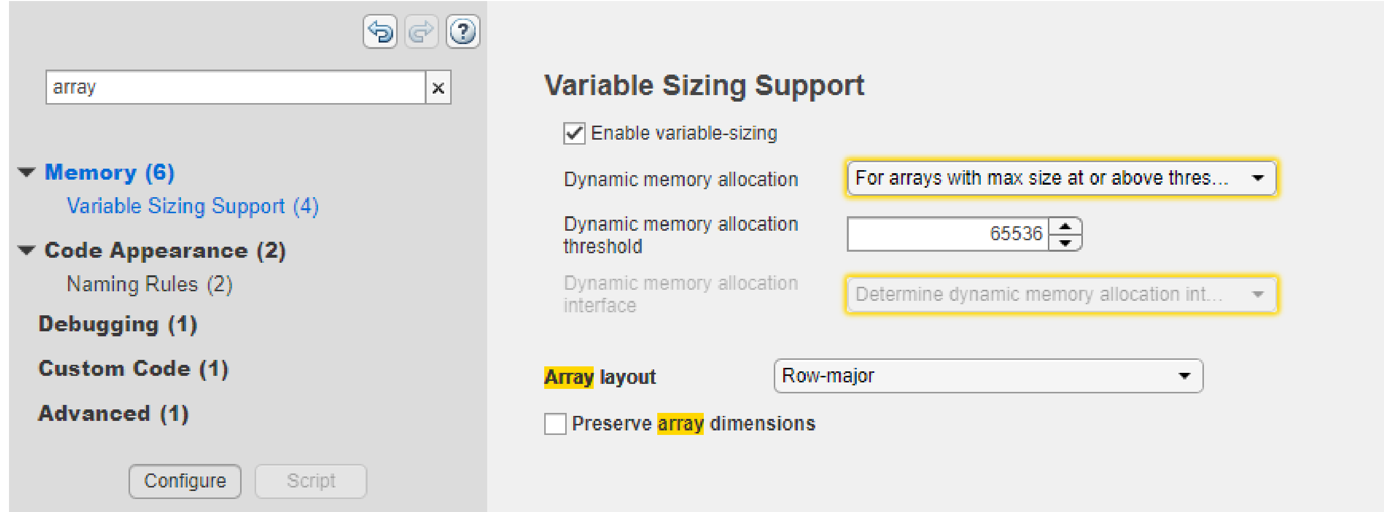
   - In the MATLAB workspace, double-click the configuration object variable.
   - At the MATLAB prompt, issue the `open` command, passing it the configuration object variable, as in this example:

     ```
     open mexcfg
     ```
3. In the dialog box, modify configuration parameters as required.

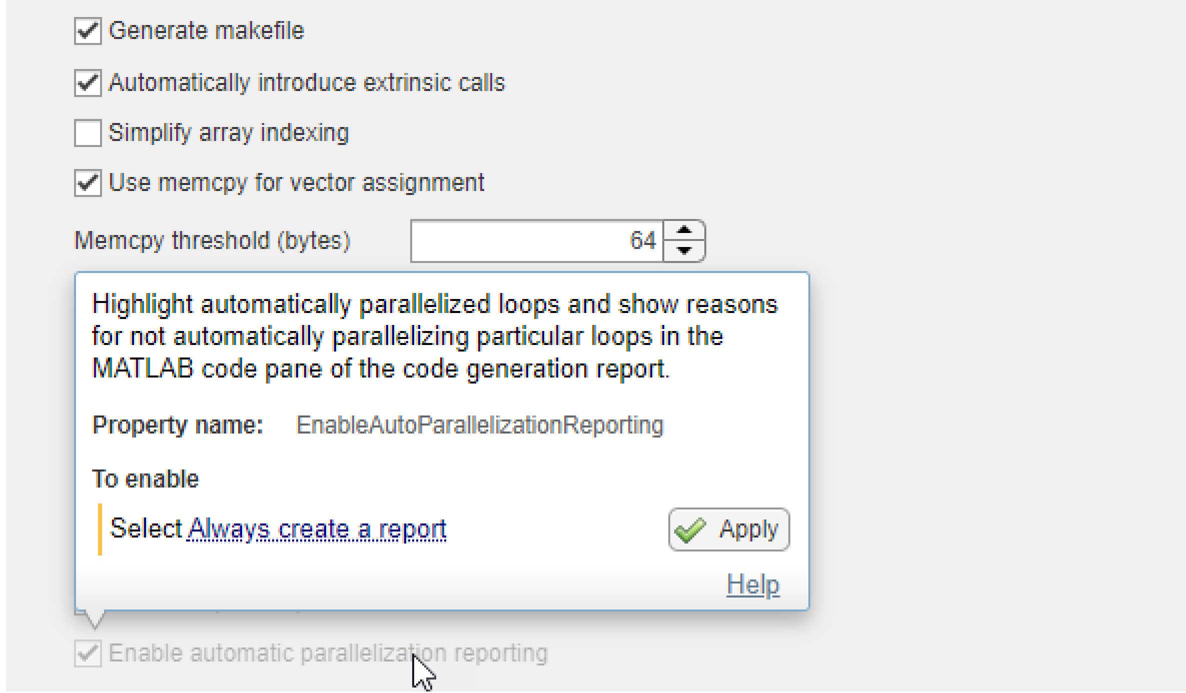
Additional Functionalities in the Dialog Box

To enable you to easily modify the configuration parameters in an interactive fashion, the dialog box provides these functionalities:

- *Search*: When you search for a string, you see the filtered results across all the settings categories. The search string might be present in a setting name, the name of an option for a setting, or in a tooltip.



- *Informative tooltips*: The tooltip for each individual setting contains the corresponding configuration object property name, a **Help** link for that property, and the name of any additional product that using that property requires. If the property is disabled, the tooltip also contains links to other properties that you must set to enable this property. You can make that change in the tooltip itself.



- *Settings with nondefault values*: The dialog box shows settings that have nondefault values in bold font. To reset such a setting to its default values, click the **Reset** button in the tooltip.
- *MISRA Compliance pane*: If you have Embedded Coder, the **MISRA Compliance** pane displays the settings that might impact MISRA compliance of the generated code. To set all of these settings to the recommended values, click **Set to Recommended Values**.

  See “Generate C/C++ Code with Improved MISRA Compliance” (Embedded Coder).
- *Generate equivalent script*: You can view the command-line script that produces your current settings by clicking the **Script** button located at the bottom of the list of categories. You can switch from the script mode back to the interactive mode by clicking the **Configure** button.

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

- “Configure Build Settings” on page 27-13

Specify Data Types Used in Generated Code

In this section...

"Specify Data Type Using the MATLAB Coder App" on page 27-24
"Specify Data Type at the Command Line" on page 27-24

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'`.

Use Generated Initialize and Terminate Functions

When generating C/C++ code from MATLAB code, the code generator automatically produces two housekeeping functions, initialize and terminate. The initialize function initializes the state on which the generated C/C++ entry-point functions operate. It must be called before you call the entry-point functions for the first time. The terminate function frees allocated memory and performs other cleanup operations. It must be called after you call the entry-point functions for the last time.

Initialize Function

The name of the generated initialize function is *primary_function_name*`_initialize`, where *primary_function_name* is the name of the first MATLAB entry-point function that you specify while generating code. The initialize function initializes the state on which the generated entry-point functions operate. The initialize function can include:

- Calls to supporting code for nonfinite data (`Inf` and `NaN`). These calls are generated if your MATLAB code contains operations that can generate nonfinite values.
- Code that initializes `global` or `persistent` variables.
- Custom code for creating an initial state that you specify. To include custom code in the initialize function, do one of the following:
  - In a code configuration object, set `CustomInitializer` to a character vector that contains the custom code.
  - In the MATLAB Coder app, on the **Custom Code** tab, specify custom code for the initialize function.

In certain situations, no initialization code is necessary and the generated initialize function is empty.

Calling Initialize Functions

If you generate a MEX function, the generated code automatically includes a call to the initialize function. If you generate standalone code, there are two possible situations:

- By default, if the initialize function is nonempty, the code generator includes a call to the initialize function at the beginning of the generated C/C++ entry-point functions. The generated code also includes checks to make sure that the initialize function is called automatically only once, even if there are multiple entry-point functions. In this situation, you do not need to manually call the initialize function.

  If the initialize function is empty, the generated C/C++ entry-point functions do not include a call to the initialize function.
- You can choose to not include a call to the initialize function in the generated entry-point functions. Do one of the following:
  - In a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object, set `RunInitializeFcn` to `false`.
  - In the MATLAB Coder app, on the **All Settings** tab, set **Automatically run the initialize function** to `No`.

  If you make this choice, you must manually call the initialize function before you call a generated entry-point function for the first time. Not calling the initialize function causes the generated entry-point functions to operate on an invalid state.

If you generate C++ code with a class interface, then the code generator produces a class constructor and destructor that perform initialization and termination operations. You do not need to manually call the `initialize` and `terminate` functions. See “Generate C++ Code with Class Interface” on page 39-4.

Examples of Generated Initialize Functions

Examples of MATLAB code patterns and the corresponding generated initialize functions:

- Your MATLAB code uses `global` or `persistent` variables. For example, define this MATLAB function:

  ```
  function y = bar
  global g
  y = g;
  end
  ```

  Generate a static library for `bar`. Specify the initial value of `g` as `1`.

  ```
  codegen -config:lib -globals {'g',1} bar
  ```

  The code generator produces the file `bar_initialize.c` in *`work`*`\codegen\lib\bar`, where *`work`* is the folder that contains `bar.m`. The function `bar_initialize` initializes the global variable `g`.

  ```
  void bar_initialize(void)
  {
    g = 1.0;
    isInitialized_bar = true;
  }
  ```

  The generated C function `bar` includes a call to `bar_initialized`. It uses the boolean `isInitialized_bar` to make sure that the initialize function is called automatically only once.

  ```
  double bar(void)
  {
    if (!isInitialized_bar) {
      bar_initialize();
    }

    return g;
  }
  ```

- Your MATLAB code contains an operation that can generate nonfinite values (`Inf` or `NaN`). For example, define a MATLAB function `foo` that calls `factorial`. The `factorial` function grows quickly and returns `Inf` for inputs greater than a certain threshold. For an input of type `double`, the threshold is `170`. Executing `factorial(171)` in MATLAB returns `Inf`.

  ```
  function y = foo(a)
  y = factorial(a);
  end
  ```

  Generate a static library for `foo`.

  ```
  codegen -config:lib foo -args {1}
  ```

  The code generator produces the file `foo_initialize.c` in *`work`*`\codegen\lib\foo`, where *`work`* is the folder that contains `foo.m`. The function `foo_initialize` calls supporting code for nonfinite data, `rt_InitInfAndNaN`, which is defined in another generated file `rt_nonfinite.c`.

```
void foo_initialize(void)
{
  rt_InitInfAndNaN();
  isInitialized_foo = true;
}
```

Terminate Function

The name of the generated terminate function is *primary_function_name*`_terminate`, where *primary_function_name* is the name of the first MATLAB entry-point function that you specify while generating code. The terminate function frees allocated memory and performs other cleanup operations.

The terminate function can also include custom cleanup code that you specify. To include custom code in the terminate function, do one of the following:

- In a code configuration object, set `CustomTerminator` to a character vector that contains the custom code.
- Alternatively, in the MATLAB Coder app, on the **Custom Code** tab, specify custom code for the terminate function.

If you generate a MEX function, the generated code automatically includes a call to the terminate function.

If you generate standalone code, the generated code does not automatically include a call to the terminate function. In this situation, you must manually invoke the terminate function after you call the generated entry-point functions for the last time.

Terminate functions are also used to clear the state of persistent variables. A persistent variable retains its state until a terminate function is invoked. For more information, see "Generate Code for Persistent Variables" on page 27-159.

Example of Generated Terminate Function

Define this MATLAB function:

```
function y = bar
global g
y = g;
end
```

Generate a static library for `bar`. Specify the initial value of `g` as `1`.

```
codegen -config:lib -globals {'g',1} bar
```

The code generator produces the file `bar_terminate.c` in *work*`\codegen\lib\bar`, where *work* is the folder that contains `bar.m`. The function `bar_terminate` sets the boolean `isInitialized_bar` (that was set to `true` after the initialize function call) to `false`.

```
void bar_terminate(void)
{
  isInitialized_bar = false;
}
```

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

- “Deploy Generated Code” on page 31-71

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 C99 (ISO). For C++, it is C++11 (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
C++11 (ISO)	C++	ISO/IEC 14882:2011(E)

The C++03 (ISO) and C++11 (ISO) math libraries are 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.

See Also

More About

- “Specify Build Configuration Parameters MATLAB Coder App” on page 27-17
- “Specify Build Configuration Parameters at the Command Line Using Configuration Objects” on page 27-18

Convert codegen Command to Equivalent MATLAB Coder Project

You can use the codegen command with the -toproject option to convert a codegen command to an equivalent MATLAB Coder project file. You can then generate code from the project file by using another codegen command or the MATLAB Coder app.

For example, to convert a codegen command with input arguments input_arguments to the project file myProject.prj, run:

```
codegen input_arguments -toproject myProject.prj
```

Input arguments to codegen include:

- Names of entry-point functions
- Input type definitions specified by using the -args option
- Code generation options, including parameters specified in configuration objects
- Names of custom source files to include in the generated code

You can also use the -toproject option to convert an incomplete codegen command to a project file. For example, to create a project file myProjectTemplate.prj that contains only the code generation parameters stored in the configuration object cfg, run:

```
codegen -config cfg -toproject myProjectTemplate.prj
```

myProjectTemplate.prj does not contain specifications of entry-point functions or input types. So, you cannot generate code from this project file. You can open myProjectTemplate.prj in the MATLAB Coder app and use it as a template to create full project files that you can use to generate code.

Note Running the codegen command with the -toproject option does not generate code. It creates only the project file.

Example: Convert a Complete codegen Command to a Project File

Define a MATLAB function, myadd, that returns the sum of two values.

```
function y = myadd(u,v) %#codegen
y = u + v;
end
```

Create a coder.CodeConfig object for generating a static library. Set TargetLang to 'C++'.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
```

At the MATLAB command line, create and run a codegen command. Specify myadd as the entry-point function. Specify the inputs to myadd as variable-size matrices of type double whose dimensions are unbounded. Specify cfg as the code configuration object. Include the -toproject option to convert the codegen command to an equivalent MATLAB Coder project file with name myadd_project.prj.

```
codegen -config cfg myadd -args {coder.typeof(1,[Inf,Inf]),coder.typeof(1,[Inf,Inf])} -toproject
```

```
Project file 'myadd_project.prj' was successfully created.
Open Project
```

The code generator creates the project file `myadd_project.prj` in the current working folder. Running `codegen` with the `-toproject` option does not generate code. It creates only the project file.

Generate code from `myadd_project.prj` by using another `codegen` command.

```
codegen myadd_project.prj
```

The code generator produces a C++ static library function `myadd` in the `*work*\codegen\lib\myadd` folder, where `*work*` is your current working directory.

Example: Convert an Incomplete codegen Command to a Template Project File

Create a `coder.CodeConfig` object for generating a static library. Set `TargetLang` to `'C++'`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
```

At the MATLAB command line, create and run a `codegen` command. Specify `cfg` as the code configuration object. Include the `-toproject` option to convert the `codegen` command to an equivalent MATLAB Coder project file with name `myProjectTemplate.prj`.

```
codegen -config cfg -toproject myProjectTemplate.prj
```

```
Project file 'myProjectTemplate.prj' was successfully created.
Open Project
```

You can now open `myProjectTemplate.prj` in the MATLAB Coder app and use it as a template to create full project files that you can use to generate code.

Limitations

When you use the `codegen` command with the `-toproject` option, these limitations apply:

- Exporting the `CodeTemplate` parameter of a `coder.EmbeddedCodeConfig` object to a project file is not supported.
- Suppose that your `codegen` command for generating a MEX function uses `coder.Constant` to define a constant input that is a `fi` object `obj`.

  Certain `fi` object properties are enabled by other properties. When you construct a `fi` object, these properties are set to their default values unless you explicitly modify them. In `obj`, you set one or more properties that are not enabled to non-default values. See “fi Object Properties” (Fixed-Point Designer).

  You convert this `codegen` command to a project file by using the `-toproject` option. You build the project file and generate a MEX function. When you pass `obj` as the constant input argument to the generated MEX function and run the MEX, the MEX might throw an error.

To fix this issue, you must set the properties of obj that are not enabled to their default values before passing it to the MEX function. To do this, define a new fi object obj_new:

```
a = mat2str(obj);
obj_new = eval(a);
```

Pass obj_new as the constant input to the generated MEX function.

See Also

codegen

More About

- "Convert MATLAB Coder Project to MATLAB Script" on page 27-35
- "Share Build Configuration Settings" on page 27-33

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.

This functionality is not supported in MATLAB Online.

Export Settings

You can export project file settings to a code configuration object by using the MATLAB Coder app or at the command line. The type of the configuration object depends on the project file settings.

Project File Settings in MATLAB Coder App	Code Configuration Object
Build type is MEX.	`coder.MexCodeConfig`
Build type is static library, dynamically linked library, or executable. One of the following conditions is true: • You do not have Embedded Coder. • You have Embedded Coder. On the **All Settings** tab, **Use Embedded Coder features** is set to `No`.	`coder.CodeConfig`
Build type is static library, dynamically linked library, or executable. You have Embedded Coder. On the **All Settings** tab, **Use Embedded Coder features** is set to `Yes`.	`coder.EmbeddedCodeConfig`

You can then either import these settings into another project or use the configuration object with the `codegen` function `-config` option to generate code at the command line.

Export Settings by Using the MATLAB Coder App

In 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. 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.

Export Settings at the Command Line

At the MATLAB command line, use the `-toconfig` option with the `coder` command to export the code configuration settings stored in a MATLAB Coder project file to a code configuration object. For example, executing this command returns a code configuration object `cfg` corresponding to `myProject.prj`.

```
cfg = coder('-toconfig','myProject.prj')
```

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

`coder` | `coder.config`

More About

- “Configure Build Settings” on page 27-13
- “Convert MATLAB Coder Project to MATLAB Script” on page 27-35
- “Convert codegen Command to Equivalent MATLAB Coder Project” on page 27-30

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.

Project to script conversion is not supported in MATLAB Online.

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

Variable	For
`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 file settings.

Project File Settings in MATLAB Coder App	Code Configuration Object
Build type is MEX.	`coder.MexCodeConfig`
Build type is static library, dynamically linked library, or executable. One of the following conditions is true: • You do not have an Embedded Coder license. • You have an Embedded Coder license. On the **All Settings** tab, **Use Embedded Coder features** is set to `No`.	`coder.CodeConfig`
Build type is static library, dynamically linked library, or executable. You have an Embedded Coder license. On the **All Settings** tab, **Use Embedded Coder features** is set to `Yes`.	`coder.EmbeddedCodeConfig`

You can import the settings from the configuration object `cfg` into a project. See “Share Build Configuration Settings” on page 27-33.

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 21-85.

Special Cases That Generate Additional MAT-File

Suppose that you convert a project file `myproject.prj` to a script `myscript.m`. In certain situations the code generator can produce an additional MAT-file in the current working folder. In such cases, the generated script loads the MAT-file and uses the stored values to define constant inputs or constant global variables in the generated code.

This behavior happens if all of these conditions are true:

- The project file `myproject.prj` was generated by converting a `codegen` command to an equivalent MATLAB Coder project. See “Convert codegen Command to Equivalent MATLAB Coder Project” on page 27-30.
- The original `codegen` command uses `coder.Constant` objects to define constant inputs or constant global variables.

- One or more of these `coder.Constant` objects are created from values that are structures, cell arrays, value classes, or large numeric constants (greater than a certain threshold). The generated MAT-file stores these values.

Even if all of the preceding conditions are true, you can avoid the creation of the auxiliary MAT-file. Before generating the script, modify the project file `myproject.prj`:

- Open `myproject.prj` in the MATLAB Coder app.
- Navigate to the **Define Input Types** page.
- Enter the constant values of the inputs or the global variables directly in the app. This action automatically saves the modified `myproject.prj`.

See Also

`codegen` | `coder`

More About

- “Convert codegen Command to Equivalent MATLAB Coder Project” on page 27-30
- “Share Build Configuration Settings” on page 27-33

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.

See Also

More About

- “Reuse Large Arrays and Structures” on page 34-50
- “Configure Build Settings” on page 27-13

Reserved Keywords

The code generator reserves the use of certain identifiers in the generated code. These identifiers include C and C++ keywords and C and C++ standard library names. Using these keywords in your MATLAB code as identifiers or function names might cause the code generator to rename them. If you do not find variables or functions that have reserved keywords as names in your generated code, they might have been renamed by the code generator.

Note You can preserve most variable names, apart from the reserved keywords, in your generated code. See “Preserve Variable Names in Generated Code” on page 27-38.

C Reserved Keywords

_Bool	_Complex	_Generic	_Imaginary
_Noreturn	_Static_assert	_Thread_local	threads
asm	auto	assert	case
char	const	continue	default
complex	void	time	tgmath
ctype	iso646	stdatomic	stddef
do	double	else	enum
extern	float	for	goto
if	inline	int	long
limits	locale	stdbool	stdio
register	restrict	return	short
signal	wctype	setjmp	string
signed	sizeof	static	struct
single	_Alignas	_Alignof	_Atomic
stdalign	inttypes	stdarg	uchar
stdint	math	errno	wchar
stdlib	stdnoreturn	break	fenv
switch	typedef	typeof	union
true	false	bool	fortran
unsigned	while	volatile	

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	

Keywords Reserved 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	

Some identifiers from the C/C++ standard libraries such as `fprintf`, `freadf`, and `I` are also reserved.

If you include these names in your MATLAB code as identifiers, they are renamed in the generated code by prepending a letter in front of the name. For example, `asm` might be renamed as `b_asm`.

This code snippet uses an input and output variable that is named `real_T`, which is a reserved keyword for code generation.

```
function real_T = foo(real_T)
real_T = real_T + 1;
end
```

In the generated code, the variable `real_T` is renamed to `b_real_T`.

```
void foo(double *b_real_T)
{
  (*b_real_T)++;
}
```

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`.

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.

See Also

More About

- "Preserve Variable Names in Generated Code" on page 27-38
- "Configure Build Settings" on page 27-13

Specify Properties of Entry-Point Function Inputs

In this section...

"Why You Must Specify Input Properties" on page 27-43
"Properties to Specify" on page 27-43
"Rules for Specifying Properties of Primary Inputs" on page 27-46
"Methods for Defining Properties of Primary Inputs" on page 27-46
"Define Input Properties by Example at the Command Line" on page 27-47
"Specify Constant Inputs at the Command Line" on page 27-49
"Specify Variable-Size Inputs at the Command Line" on page 27-49

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**
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 all fields 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 27-67 and “Specifying Class and Size of Structure Array” on page 27-68.

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
```

Specifying Multiple Signatures for MEX Function

To generate a multisignature MEX function from an entry-point function, provide multiple `-args` specifications for the same entry-point function. The generated MEX function works with the multiple signatures that you provide during code generation. For more information on multisignature MEX, see “Generate One MEX Function for Multiple Signatures” on page 27-82.

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

Class Name	Description
`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 24-3	• 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 27-47 **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

Method	Advantages	Disadvantages
“Define Input Properties Programmatically in the MATLAB File” on page 27-60	• Integrated with MATLAB code; no need to redefine properties each time you invoke MATLAB Coder • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures	• Uses complex syntax • MATLAB Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.

Define Input Properties by Example at the Command Line

- “Command-Line Option -args” on page 27-47
- “Rules for Using the -args Option” on page 27-47
- “Specifying Properties of Primary Inputs by Example at the Command Line” on page 27-48
- “Specifying Properties of Primary Fixed-Point Inputs by Example at the Command Line” on page 27-48

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 or types. 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`.

You can pass the output type from one entry-point function as the input to another. See “Pass an Entry-Point Function Output as an Input” on page 27-85. For information about specifying cell array inputs, see “Specify Cell Array Inputs at the Command Line” on page 27-52.

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 27-65 for `codegen`.

You can also create `coder.Type` objects interactively by using the Coder Type Editor. See “Create and Edit Input Types by Using the Coder Type Editor” on page 27-69.

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 values that match the compile-time values. You can control whether a MEX function signature includes constant inputs and whether the MEX function checks the values that you provide for constant inputs. See “Constant Input Checking in MEX Functions” on page 27-57.

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 27-98.

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}
```

See Also

More About

- “Define String Scalar Inputs” on page 5-12
- “Specify Objects as Inputs at the Command Line” on page 15-27
- “Specify Cell Array Inputs at the Command Line” on page 27-52
- “Specify Number of Entry-Point Function Input or Output Arguments to Generate” on page 19-3
- “Pass an Entry-Point Function Output as an Input” on page 27-85

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 9-2. When you provide an example cell array to `codegen` or `coder.typeof`, the function determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `coder.CellType` `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

If you have a test file, you can use `coder.getArgTypes` to determine input types. In the output cell array of types, for cell array inputs, `coder.getArgTypes` returns a `coder.CellType` object. If you want a different classification (homogeneous or heterogeneous), use the `makeHomogeneous` or `makeHeterogeneous` methods.

Specify Cell Array Inputs by Example

To specify a cell array input by example, provide an example cell array in the `-args` option of the `codegen` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

  ```
  codegen myfunction -args {{1 2 3}} -report
  ```

  The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.
- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

  ```
  codegen myfunction -args {{'a', 1}} -report
  ```

  The input argument is a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x1 double.

Specify the Type of the Cell Array Input

To specify the type of a cell array input, use `coder.typeof` to create a `coder.CellType` object. Pass the `coder.CellType` object to the `-args` option of the `codegen` command.

For example:

- To specify a 1x3 cell array whose elements have class double:

  ```
  t = coder.typeof({1 2 3});
  codegen myfunction -args {t} -report
  ```

  The input argument is a 1x3 homogeneous cell array whose elements are 1x1 double.
- To specify a 1x2 cell array whose first element has class char and whose second element has class double:

  ```
  t = coder.typeof({'a', 1});
  codegen myfunction -args {t}
  ```

  The input argument is a 1x2 heterogeneous cell array whose first element is a 1x1 char and whose second element is a 1x1 double.

You can also use the advanced function `coder.newtype` to create a `coder.CellType` object.

Make a Homogeneous Copy of a Type

If `coder.typeof` returns a heterogeneous cell array type, but you want a homogeneous type, use the `makeHomogeneous` method to make a homogeneous copy of the type.

The following code creates a heterogeneous type.

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

```
t = 

coder.CellType
   1x2 heterogeneous cell 
      f0: 1x1 double
      f1: 1x2 double
```

To make a homogeneous copy of the type, use:

```
t = makeHomogeneous(t)
```

```
t = 

coder.CellType
   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 27-47
- “Specify Constant Inputs at the Command Line” on page 27-49

More About

- “Code Generation for Cell Arrays” on page 9-2

Constant Input Checking in MEX Functions

When you specify a constant input argument for generation of a MEX function, by default the generated MEX function signature includes this argument. When you call the MEX function, it checks that the value that you provide for the constant argument is the value specified at code generation time.

To generate a MEX function that does not check constant input values or that does not include constant input arguments, modify the constant input checking configuration parameter:

- If you use the MATLAB Coder app:
  1. On the **Generate Code** page, set **Build type** to `MEX`.
  2. Click **More Settings**.
  3. On the **All Settings** tab, set **Constant Inputs** to one of the values in the table.
- If you use `codegen`, in a MEX configuration object, set the `ConstantInputs` property to one of the values in the table.

Constant Inputs (App)	`ConstantInputs` (Configuration Object)	Description
`Check values at run time`	`'CheckValues'`	This value is the default value. When you call the MEX function, it checks that the value you provide for a constant input argument is the value specified at code generation time. You can call the MEX function and the original MATLAB function with the same arguments. Therefore, you can use the same test file for both functions. Checking the values can add to the execution time of the MEX function.
`Ignore input value`	`'IgnoreValues'`	When you call the MEX function, it ignores the value that you provide for a constant input argument. It uses the value specified at code generation time. You can use the same test file without the overhead of checking the constant argument values.

Constant Inputs (App)	ConstantInputs (Configuration Object)	Description
Remove from MEX signature	'Remove'	The code generator removes constant input arguments from the MEX function signature. When you call the MEX function, you do not provide a value for a constant input argument. This option is for backward compatibility.

Control Whether a MEX Function Checks the Value of a Constant Input

This example shows how to use the ConstantInputs parameter to control whether a MEX function checks the value of a constant input argument.

Write a function myadd that returns the sum of its inputs.

```
function c = myadd(a,b)
c = a + b;
end
```

Create a configuration object for MEX code generation.

```
mexcfg = coder.config('mex');
```

Look at the value of the constant input checking configuration parameter, ConstantInputs.

```
mexcfg.ConstantInputs
```

```
ans =

    'CheckValues'
```

It has the default value, CheckValues.

Generate a MEX function myadd_mex. Specify that the first argument is a double scalar and that the second argument is a constant with value 3.

```
codegen myadd -config mexcfg -args {1, coder.Constant(3)}
```

```
Code generation successful.
```

Call myadd_mex. You must provide the input 3 for the second argument.

```
myadd_mex(1,3)
```

```
ans =
```

```
     4
```

Modify ConstantInputs so that the MEX function does not check that the input value matches the value specified at code generation time.

```
mexcfg.ConstantInputs = 'IgnoreValues';
```

Generate myadd_mex.

```
codegen myadd -config mexcfg -args {1, coder.Constant(3)}
```

```
Code generation successful.
```

Call myadd_mex with a constant input value other than 3, for example, 5.

```
myadd_mex(1,5)
```

```
ans =

     4
```

The MEX function ignores the input value 5. It uses the value 3, which is the value that you specified for the constant argument b when you generated myadd_mex.

Modify ConstantInputs so that the MEX function signature does not include the constant input argument.

```
mexcfg.ConstantInputs = 'Remove';
```

Generate myadd_mex.

```
codegen myadd -config mexcfg -args {1, coder.Constant(3)}
```

```
Code generation successful.
```

Call myadd_mex. Provide the value 1 for a. Do not provide a value for the constant argument b.

```
myadd_mex(1)
```

```
ans =

     4
```

See Also

coder.MexCodeConfig

More About

- “Specify Properties of Entry-Point Function Inputs” on page 27-43
- “Configure Build Settings” on page 27-13

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 27-60
- “Specify fi Class” on page 27-60
- “Specify Structure Class” on page 27-61
- “Specify Cell Array Class” on page 27-61
- “Specify Fixed Size” on page 27-61
- “Specify Scalar Size” on page 27-62
- “Specify Upper Bounds for Variable-Size Inputs” on page 27-62
- “Specify Inputs with Fixed- and Variable-Size Dimensions” on page 27-62
- “Specify Size of Individual Dimensions” on page 27-63
- “Specify Real Input” on page 27-63
- “Specify Complex Input” on page 27-63
- “Specify numerictype of Fixed-Point Input” on page 27-63
- “Specify fimath of Fixed-Point Input” on page 27-64
- “Specify Multiple Properties of Input” on page 27-64

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 27-63. You can also set the `fimath` properties, see “Specify fimath of Fixed-Point Input” on page 27-64. 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 27-66.

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 27-63. You can also set the fimath properties. See “Specify fimath of Fixed-Point Input” on page 27-64. 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 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 9-2.

If you specify the properties of the first element only, the cell array is homogeneous. For example, the following code specifies that `C` is a 1x3 homogeneous cell array whose elements are 1x1 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1  3]));
assert(isa(C{1}, 'double'));
...
```

If you specify the properties of the first element only, but also assign a structure type name to the cell array, the cell array is heterogeneous. Each element has the properties of the first element. For example, the following code specifies that C is a 1x3 heterogeneous cell array. Each element is a 1x1 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1  3]));
assert(isa(C{1}, 'double'));
coder.cstructname(C, 'myname');
...
```

If you specify the properties of each element, the cell array is heterogeneous. For example, the following code specifies a 1x2 heterogeneous cell array whose first element is 1x1 char and whose second element is 1x3 double.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) == [1  2]));
assert(isa(C{1}, 'char'));
assert(all(size(C{2}) == [1 3]));
assert(isa(C{2}, 'double'));
...
```

If you specify more than one element, you cannot specify that the cell array is variable size, even if all elements have the same properties. For example, the following code specifies a variable-size cell array. Because the code specifies the properties of the first and second elements, code generation fails.

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1  2]));
assert(isa(C{1}, 'double'));
assert(isa(C{2}, 'double'));
...
```

In the previous example, if you specify the first element only, you can specify that the cell array is variable-size. For example:

```
...
assert(isa(C, 'cell'));
assert(all(size(C) <= [1  2]));
assert(isa(C{1}, 'double'));
...
```

Specifying Class and Size of Scalar Structure

Suppose that you define S as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

The following code specifies the properties of the function input S and its fields:

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


% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8');
...
```

In most cases, when you do not explicitly specify values for properties, MATLAB Coder uses defaults —except for structure fields. The only way to name a field in a structure is to set at least one of its properties. At a minimum, you must specify the class of a structure field.

Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume that you have defined S as the following 1-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input S by using the first element of the array:

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

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [1 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

The only way to name a field in a structure is to set at least one of its properties. At a minimum, you must specify the class of all fields.

Create and Edit Input Types by Using the Coder Type Editor

C/C++ source code includes type declarations for all variables. MATLAB code does not include explicit type declarations. To allow the generation of C/C++ code with specific types, you must specify the properties (class, size, and complexity) of all input variables to the MATLAB entry-point functions during C/C++ or MEX code generation. An entry-point function is a top-level MATLAB function from which you generate code. The code generator uses these input properties to determine the properties of all variables in the generated code. Different input type specifications can cause the same MATLAB code to produce different versions of the generated code.

When you generate C/C++ or MEX code at the command line, one of the ways to specify the properties of an input argument is by using a `coder.Type` object that contains information about class, size, and complexity (and sometimes other properties) of the argument. You can create and edit `coder.Type` objects programmatically at the command line, or interactively by using the Coder Type Editor.

For more information about creating `coder.Type` objects at the command line, see `coder.typeof` and `coder.newtype`.

Note To create and edit composite types such as structures and cell arrays, or types that have many customizable parameters such as `embedded.fi`, use the Coder Type Editor. Examples of such types are shown later in this topic.

Open the Coder Type Editor

To launch the Coder Type Editor, do one of the following:

- Launch an empty type editor by using the `coderTypeEditor` command:

  ```
  coderTypeEditor
  ```
- Open the type editor pre-populated with `coder.Type` objects corresponding to the workspace variables `var1`, `var2`, and `var3` by typing:

  ```
  coderTypeEditor var1 var2 var3
  ```
- Open a `coder.Type` object `myType` that already exists in your base MATLAB workspace:
  - Double click `myType` in the workspace.
  - Display `myType` at the command line and click the *Edit Type Object* link that appears at the end of the display.
  - Use this command at the MATLAB command line:

    ```
    open myType
    ```

Common Editor Actions

By using the toolstrip buttons in the type editor, you can perform these actions:

- Create a new type by clicking **New Type** and specifying the type, size, complexity, and other properties of the `coder.Type` object.
- Convert an existing variable to a type by clicking **From Variable** and specifying a variable that already exists in the base workspace.

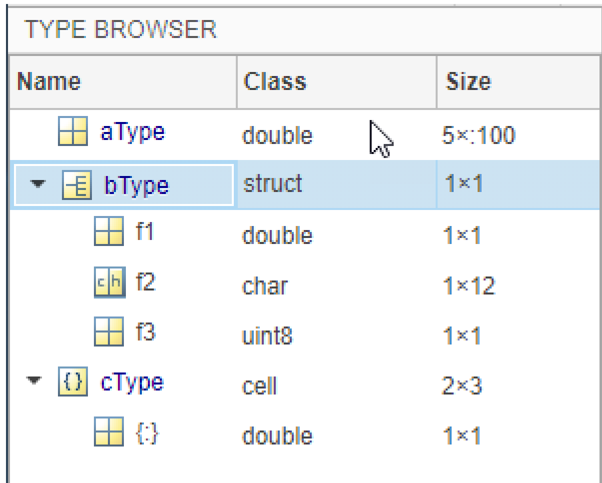
- Create a new type from an example value by clicking **From Example** and entering MATLAB code that the software converts to a `coder.Type` object.
- Load all `coder.Type` objects from the base workspace to the **Type Browser** pane of the type editor by clicking **Load All**.
- Edit an existing type by selecting it in the **Type Browser** and modifying its properties.
- Save all `coder.Type` objects in the type editor by clicking **Save All**.
- Remove a selected type from **Type Browser** by clicking **Delete**. Alternatively, remove all types from the **Type Browser** by clicking **Delete** > **Delete all**. Deleting a `coder.Type` object from the **Type Browser** does not delete the object from the base MATLAB workspace.
- Export a MATLAB script that contains the code to recreate all the types by clicking **Share** > **MATLAB Script**. Or, create a MAT file that contains all the types by clicking **Share** > **MAT File**.
- Undo and redo your last action in the type editor by using the buttons.

These are some additional actions that you can perform in the Coder Type Editor:

- In both the **Type Browser** pane and the **Type Properties** pane, copy a type object and paste it either as a new type or a field of an existing structure type. You can also copy the properties of one existing type into another existing type.
- Change the order of fields of a structure type. View the type in the properties pane and use drag-and-drop action.

Type Browser Pane

The **Type Browser** pane shows the name, class, and size of the `coder.Type` objects that are currently loaded in the type editor. For composite types such as structures, cell arrays, or classes, you can expand the display of the `code.Type` object in the **Type Browser** pane. The expanded view shows the name, class, and complexity of the individual fields or properties of the composite type.

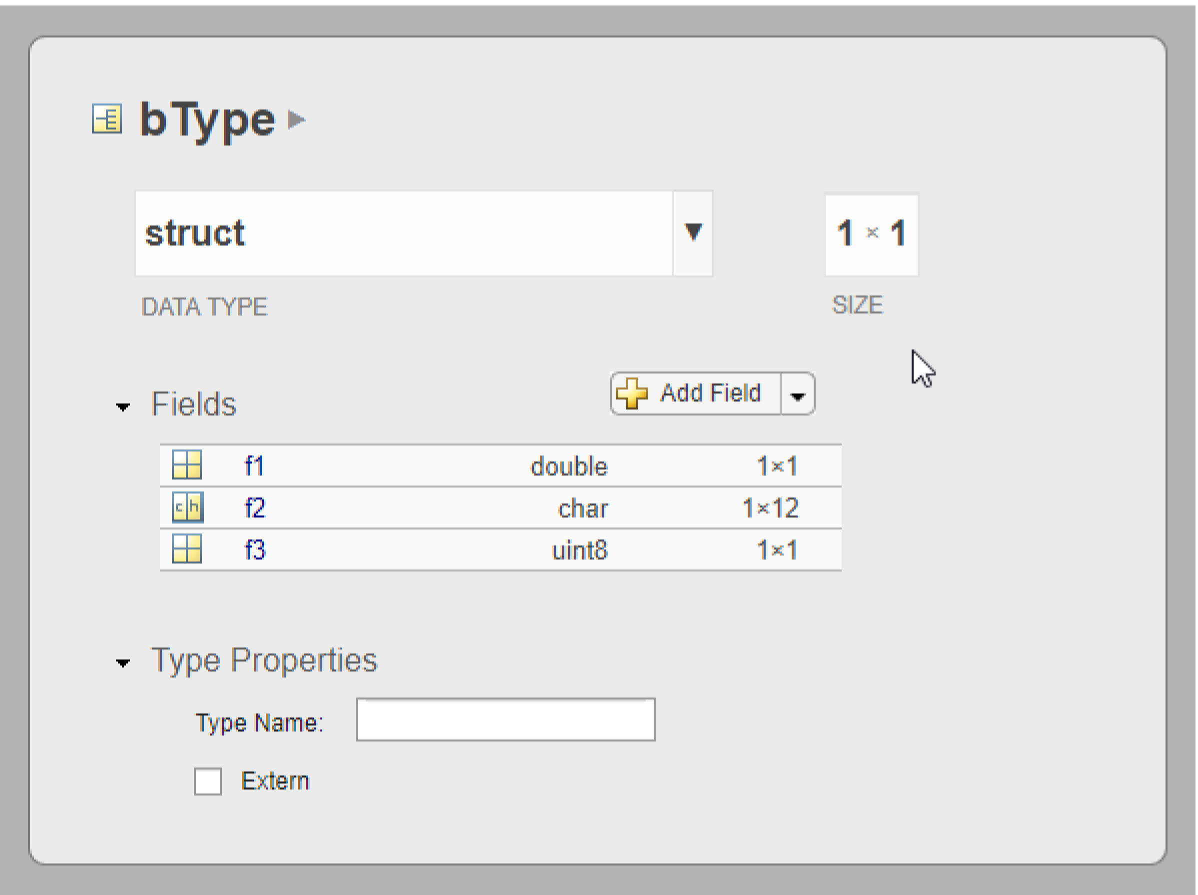


Visual Indicators on the Type Browser

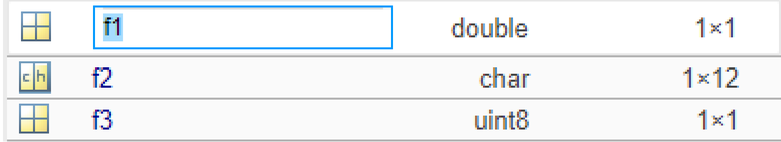
Indicator	Description
expander	The type has fields or properties that you can see by clicking the expander.
`{:}`	Homogeneous cell array (all elements have the same properties).
`{n}`	nth element of a heterogeneous cell array.
`:n`	Variable-size dimension with an upper bound of n.
`:inf`	Variable-size dimension that is unbounded.

Type Properties Pane

The type properties pane displays the class (data type), size, and other properties of the `coder.Type` object that is currently selected in the **Type Browser**. For composite types such as structures and classes, this pane also shows the name, class, and size of each constituent field or property.



To edit the name, class, and size of a field in place, double-click the item.



Alternatively, click a field. The view in the type editor pane changes to display the properties of that field. Edit name, class(data type), size, or other properties in the pane.

The breadcrumb shows the nested path to the field that is currently open in the type properties pane. Click a field in the breadcrumb to display it in the pane. You can also edit the name of a type directly in the breadcrumb.

MATLAB Code Pane

The MATLAB Code pane displays the MATLAB script that creates the `coder.Type` object that is currently selected in the **Type Browser**. To automate the creation of this type, copy this script and include it in your build script.

MATLAB CODE

```
1  childTypes.f1 = coder.newtype('double', [1 1], [0 0]);
2  childTypes.f2 = coder.newtype('char', [1 12], [0 0]);
3  childTypes.f3 = coder.newtype('uint8', [1 1], [0 0]);
4  bType = coder.newtype('struct', childTypes, [1 1], [0 0]);
5
   clear childTypes;
```

See Also

`coder` | `coder.newtype` | `coder.typeof` | `coderTypeEditor`

More About

- "Specify Properties of Entry-Point Function Inputs" on page 27-43

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 34-67

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 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.

See Also

More About

- “Configure Build Settings” on page 27-13
- “Code Generation Reports” on page 28-7

Paths and File Infrastructure Setup

In this section...
“Compile Path Search Order” on page 27-76
“Specify Folders to Search for Custom Code” on page 27-76
“Naming Conventions” on page 27-76

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 20-2.

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**. **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.

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
Mac (64-bit)	`.mexmaci64`	`.a`	`.dylib`	None
Windows (64-bit)	`.mexw64`	`.lib`	`.dll` Also, generates an import library with a `.lib` extension that is required for linking against the `.dll`.	`.exe`

See Also

`codegen`

More About

- "Resolution of Function Calls for Code Generation" on page 20-2
- "Reserved Keywords" on page 27-39

Generate Code for Multiple Entry-Point Functions

In this section...

"Generating Code for Multiple Entry-Point Functions" on page 27-78
"Call a Single Entry-Point Function from a MEX Function" on page 27-79
"Generate Code for More Than One Entry-Point Function Using the MATLAB Coder App" on page 27-79

An entry-point function is a top-level MATLAB function from which you generate code. For many applications, you may only need to generate code for a single entry-point function. You can also generate C/C++ code from multiple entry-point functions at the same time. By using multiple entry-point functions, you can:

- Generate multi-functional C/C++ libraries that contain larger levels of functionality than if you were to generate independent libraries for each entry-point function.
- Generate code that shares code more efficiently when multiple entry-point functions rely on the same subfunctions.
- Generate library functions that can communicate using shared memory, for example, when they use the same global variables.

As a best practice, generate a MEX function to validate entry-point interactions in MATLAB before generating a C/C++ library.

Generating Code for Multiple Entry-Point Functions

To generate code for more than one entry-point function, use the syntax from the `codegen` reference page. By default, for MEX code generation, `codegen`:

- Generates a MEX function in the current folder. Only a single MEX function is generated when you specify multiple entry-point functions. To call a single entry-point function from a generated MEX function, see "Call a Single Entry-Point Function from a MEX Function" on page 27-79.
- Names the MEX function `*name*_mex`. *`name`* is the name of the first entry-point function from an *alphabetical* order.
- Stores generated files in the subfolder `codegen/mex/`*`subfolder`*. *`subfolder`* is the name of the first entry-point function from a *left-to-right* order (as they are entered after the `codegen` command).

You can specify the output file name and subfolder name using the `-o` option:

```
codegen -o myOutputFileName fun1 fun2
```

In this case, `codegen` generates a MEX function named `myOutputFileName` in the current folder and stores generated files in the subfolder `codegen/mex/myOutputFileName`.

Example: Generating Code for Two Entry-Point Functions

Generate a MEX function for two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input and `ep2` takes two inputs. Using the `-o` option, name the generated MEX function `sharedmex`:

```
codegen -o mySharedMex ep1 -args {single(0)} ep2 -args {0,zeros(1,1024)}
```

`codegen` generates a MEX function named `mySharedMex.mex` in the current folder and stores generated files in the subfolder `codegen/mex/mySharedMex`.

To generate and compile standalone library code, use the `-config:lib` option.

```
codegen -config:lib -o mySharedLib ep1 -args single(0) ep2 -args {0,zeros(1,1024)}
```

The `codegen` command generates the C/C++ library code in the `codegen/lib/mySharedLib` folder.

To use the output type from one entry-point function as the input type to another, see "Pass an Entry-Point Function Output as an Input" on page 27-85. For information on viewing entry-point functions in the code generation report, see "Code Generation Reports" on page 28-7.

Call a Single Entry-Point Function from a MEX Function

Suppose that you have a MEX function `myMex` generated from multiple entry-point functions, `fun1`, `fun2`, …, `funN`. You can call a single entry-point function, `fun_i`, by using this syntax:

```
myMex('fun_i',param1,..,paramM)
```

Here the MATLAB function signature for `fun_i` is `fun_i(param1,..,paramM)`.

For example, consider the MEX function, `mySharedMex`, that has entry-point functions `ep1` and `ep2`. To call `ep1` with an input parameter `u`, enter:

```
mySharedMex('ep1',u)
```

To call `ep2` with input parameters `v` and `x`, enter:

```
mySharedMex('ep2',v,x)
```

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. To avoid code generation errors, you must store the project file and all entry-point MATLAB function files in the same 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. To go to the **Define Input Types** step, click **Next**. 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. To go to the **Check for Run-Time Issues** step, click **Next**.

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. To navigate to the problematic code and fix the issue, click these messages. In this example, the app does not detect issues.

3. To go to the **Generate Code** step, click **Next**.

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 27-76.

   You can now test your MEX function in MATLAB. See “Call a Single Entry-Point Function from a MEX Function” on page 27-79.

   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`.

See Also

`codegen` | `coder.OutputType`

More About

- “Pass an Entry-Point Function Output as an Input” on page 27-85
- “Specify Properties of Entry-Point Function Inputs” on page 27-43

Generate One MEX Function for Multiple Signatures

In this section...

"Generate Multisignature MEX Function for a Single Entry-Point Function" on page 27-82
"Generate Multisignature MEX Function for Multiple Entry-Point Functions" on page 27-83

An entry-point function is a top-level MATLAB function from which you generate code. If your entry-point function has inputs, you must specify the properties of the inputs to generate a MEX function. In this case, the generated MEX function works only with the signature of the entry-point function that you specify during code generation.

If your entry-point function supports multiple signatures, you can generate a single MEX function instead of generating a separate MEX function for each signature. The generated MEX function works with the multiple signatures provided during code generation.

By using multisignature MEX functionality, you can:

- Generate one MEX function that supports the multiple signatures that you specify in the entry-point function.
- Reduce the overhead involved in generating and using separate MEX functions for each signature of your entry-point function.
- Achieve MATLAB function-like behavior in the generated MEX function.

Generate Multisignature MEX Function for a Single Entry-Point Function

To generate a multisignature MEX function, consider this function `myAdd`:

```
function y = myAdd(a,b)
%#codegen
y = a+b;
end
```

Suppose that you want to generate a MEX function from `myAdd` that works with three different data types: `double`, `int8`, and `vector of doubles`. Specify the three arguments as: `{1,2}`, `{int8(2), int8(3)}`, and `{1:10, 1:10}`. You specify the entry-point function followed by a `-args` for each signature of the entry-point function.

To generate code for `myAdd` function, at the MATLAB command prompt, run this `codegen` command:

```
codegen -config:mex myAdd.m -args {1,2} -args {int8(2),int8(3)} -args {1:10,1:10} -report
```

This syntax generates a single MEX function `myAdd_mex` for the signatures specified in the `codegen` command.

At the command prompt, call the generated MEX function `myAdd_mex`. Make sure that the values you pass to `myAdd_mex` match the input properties that you specified in the `codegen` command.

```
myAdd_mex(3,4)
```

```
ans =

     7
```

```
myAdd_mex(int8(5),int8(6))
```

```
ans =

  int8

   11
```

```
myAdd_mex(1:10,2:11)
```

```
ans =

     3     5     7     9    11    13    15    17    19    21
```

Running the MATLAB function `myAdd` with these input values produces the same output. These test cases verify that `myAdd` and `myAdd_mex` have the same behavior.

Generate Multisignature MEX Function for Multiple Entry-Point Functions

During code generation, you can also generate one MEX function for multiple entry-point functions containing multiple signatures.

Suppose that you have two entry-point functions `myAdd` and `myMul`. The first entry-point function, `myAdd` returns the sum of two values:

```
function y = myAdd(a,b)
%#codegen
y = a+b;
end
```

The second entry-point function, `myMul` returns the multiplication of two values:

```
function y = myMul(a,b)
%#codegen
y = a*b;
end
```

You specify the entry-point function followed by a `-args` for each signature of the entry-point function. Consider that the function `myAdd` supports the input types `double` and `int8`. Specify these arguments as: `{1,2}` and `{int8(1), int8(2)}`. Similarly, if the function `myMul` supports the input types `double` and `int16`, specify these arguments as: `{1,2}` and `{int16(1), int16(2)}`. Now, you can generate a MEX function from your entry-point functions.

To generate code for `myAdd` and `myMul` functions, at the MATLAB command prompt, run this `codegen` command:

```
codegen -config:mex myAdd.m -args {1,2} -args {int8(1),int8(2)} myMul.m -args {1,2} -args {int16(1),int16(2)} -o 'myMath'
```

This syntax generates one MEX function `myMath` for all the signatures that you specified in the `codegen` command.

You can verify the output values by using the generated MEX function `myMath` at the command prompt. Make sure that the values you pass to `myMath` match the input properties that you specified before code generation.

```
myMath("myAdd",3,4)
```

```
ans =

     7

myMath("myAdd",int8(5),int8(6))

ans =

   int8
   11

myMath("myMul",3,4)

ans =

     12

myMath("myMul",int16(5),int16(6))

ans =

    int16
    30
```

Running the MATLAB function `myAdd` and `myMul` with these input values produces the same output. These test cases verify that `myAdd`, `myMul`, and the generated MEX function `myMath` have the same behavior.

Limitations

Multisignature MEX generation does not support:

- `fiaccel -float2fixed` configuration.
- Defining input parameters programmatically. See "Define Input Properties Programmatically in the MATLAB File" on page 27-60.

See Also

`codegen` | `coder.MexCodeConfig`

More About

- "Generate Code for Multiple Entry-Point Functions" on page 27-78
- "Specify Properties of Entry-Point Function Inputs" on page 27-43

Pass an Entry-Point Function Output as an Input

When you generate code for multiple entry-point functions, you must specify the input types for each function. Using `coder.OutputType`, you can pass the output type of one function as the input type to another function. For example, to use the type of the second output from a function `foo1` as the input type to a function `foo2`, enter:

```
codegen foo1 -args {7, 42} foo2 -args {coder.OutputType('foo1',2)}
```

You can also use `coder.OutputType` to facilitate the process of partitioning, componentizing, or extending your code base. For example, when your MATLAB code uses or accepts a complicated, aggregate data type, consider creating a separate constructor function that creates that data type. Then, generate code for multiple entry-point functions, using `coder.OutputType` to pass the output type from the constructor to your other entry-point functions.

For more information on using multiple entry-point functions, see "Generate Code for Multiple Entry-Point Functions" on page 27-78.

Pass an Entry-Point Function Output as an Input to Another Entry-Point Function

The `coder.OutputType` function provides a way to chain together entry-point functions that use the same data types. Use `coder.OutputType` to:

- Simplify the input type specification process. When an existing entry-point function creates or defines a data type, you can reuse that definition for the input to a different entry-point function.
- Synchronize and align data between entry-point functions. When you use `coder.OutputType` to pass a data type, there is only a single source for the type definition, and that definition is used by both functions.

To understand these advantages, compare two cases where you generate code with and without using `coder.OutputType`.

Example: Reuse a Nested Structure Output Type as an Input Type

Suppose that you have a complicated data type that is important to your code base. You have multiple entry-point functions that rely on this data type for input, output, and internal computation. You require the interfaces between the generated function code to use the same type definition.

For the purposes of this example, suppose that the data type is a nested structure, with a variable-size array stored in the lowest-level property. You want to name this structure type `squiggle` in the generated code. In MATLAB, you write a constructing function for the data type called `myConstuctor`:

```
function [out] = myConstructor(a, b)
% create a variable-sized array with upper bounds of 100-by-100
coder.varsize('myStruct.f1.f2.f3.f4', [100 100], [1 1]);
% define the nested structure type
myStruct = struct('f1', struct('f2', struct('f3', struct('f4', zeros(a,b) ))));
% specify the name of the structure and one of its fields
coder.cstructname(myStruct.f1.f2.f3,'squiggle_f3');
coder.cstructname(myStruct,'squiggle');
out = myStruct;
```

You write a second function, useConstructor, that takes the squiggle type as input, performs addition, and pushes additional columns on to the end of the data.

```
function x = useConstructor(x, n)
xz = x.f1.f2.f3.f4;
b = zeros(size(xz,1),1);
for i = 1:n
    xz = [(xz + pi), b];
end
x.f1.f2.f3.f4 = xz;
```

To generate code for myConstructor and useConstructor and treat them as multiple entry-point functions, you must specify the input types for both functions. Specify the input types for myConstructor by using two integers. For useConstructor, specify the input type as the output type from myConstructor by using coder.OutputType:

```
v = coder.OutputType('myConstructor');
codegen myConstructor -args {5,1} useConstructor -args {v,3} -report -config:lib
```

In the generated code, the function interfaces are aligned. The two entry-point functions use the same type definition for squiggle. You can use the generated code for the constructor to create an input type for the generated code for useConstructor.

Example: Manually Define an Input Type Without Using coder.OutputType

If you do not use coder.OutputType to define the input type for useConstructor, you must specify the input type by using coder.typeof and coder.StructType class properties:

```
% MATLAB type definition for squiggle
myStruct = struct('f1', struct('f2', struct('f3', struct('f4', zeros(2) ))));
t = coder.typeof(myStruct);
t.Fields.f1.Fields.f2.Fields.f3.Fields.f4 = coder.typeof(zeros(2), [100 100], [1 1]);
t.Fields.f1.Fields.f2.Fields.f3.TypeName = 'squiggle_f3';
t.TypeName = 'squiggle';
```

To generate static library code, enter:

```
codegen myConstructor -args {5,1} useConstructor -args {t,3} -report -config:lib
```

As in the first example, the function interfaces are aligned. However, creating and maintaining the type definition for squiggle is labor-intensive. Changes that you make to the type definition must be replicated in two places: the myConstructor function and the current workspace variable t. These changes can fall out of synchronization, particularly when working with complicated type definitions. Use coder.OutputType to assist in your development process.

Use coder.OutputType to Facilitate Code Componentization

If your MATLAB code uses large, complicated, or aggregate type definitions, you can separate your code into different entry-point function components (such as a constructor and an operator) and use coder.OutputType to pass the type definition between them. The coder.OutputType function enables you to ensure a matching interface between the different entry-point functions.

Example: Create a Constructor and Use coder.OutputType to Pass the Output Type

Consider the function useSparse that performs an operation on a sparse matrix input.

```
function out = useSparse(in)
%#codegen
out = in*2;
```

If you generate code for `useSparse`, you must manually construct the appropriate input type in C/C++. To automate and simplify the type construction, write a constructor for the sparse matrix.

```
function A = makeSparse(i,j,v,m,n)
%#codegen
A = sparse(i,j,v,m,n);
```

To generate code, use `coder.OutputType` to pass the output from the constructor as the input to `useSparse`. Define your input argument as a 3-by-5 matrix.

```
t = coder.OutputType('makeSparse');
S = round(rand(3,5));
[m,n] = size(S);
[i,j,v] = find(S);
i = coder.typeof(i,[inf 1]); % allow number of nonzero entries to vary
codegen makeSparse -args {i,i,i,m,n} useSparse -args {t} -report
```

Using the generated C/C++ code, you can call `makeSparse` to generate the input to `useSparse`. The `coder.OutputType` function makes it easy to create and align the interface for separate entry-point functions that belong to a common code base.

See Also

`coder.OutputType` | `coder.StructType` | `coder.cstructname` | `coder.typeof` | `coder.varsize`

More About

- "Generate Code for Multiple Entry-Point Functions" on page 27-78
- "Specify Properties of Entry-Point Function Inputs" on page 27-43
- "Code Generation for Sparse Matrices" on page 5-14

Generate Code for Global Data

In this section...

"Workflow" on page 27-88
"Declare Global Variables" on page 27-88
"Define Global Data" on page 27-88
"Synchronizing Global Data with MATLAB" on page 27-90
"Define Constant Global Data" on page 27-92
"Global Data Limitations for Generated Code" on page 27-94

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 27-88.
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 27-90.

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 27-88 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 24-26.

   If you do not specify the type, you must create a variable with the same name in the global workspace.

Defining Global Data at the Command Line

To define global data at the command line, use the `codegen -globals` option. For example, to compile the `use_globals` function described in "Declare Global Variables" on page 27-88, 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 27-96.

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 27-90.

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 27-94.

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 27-91.

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 27-90.

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.

Clear Global Data

Because MEX functions and MATLAB each have their own copies of global data, you must `clear` both copies to ensure that consecutive MEX runs produce the same results. The `clear global` command removes only the copy of the global data in the MATLAB workspace. To remove both copies of the data, use the `clear global` and `clear mex` commands together. The `clear all` command also removes both copies.

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.

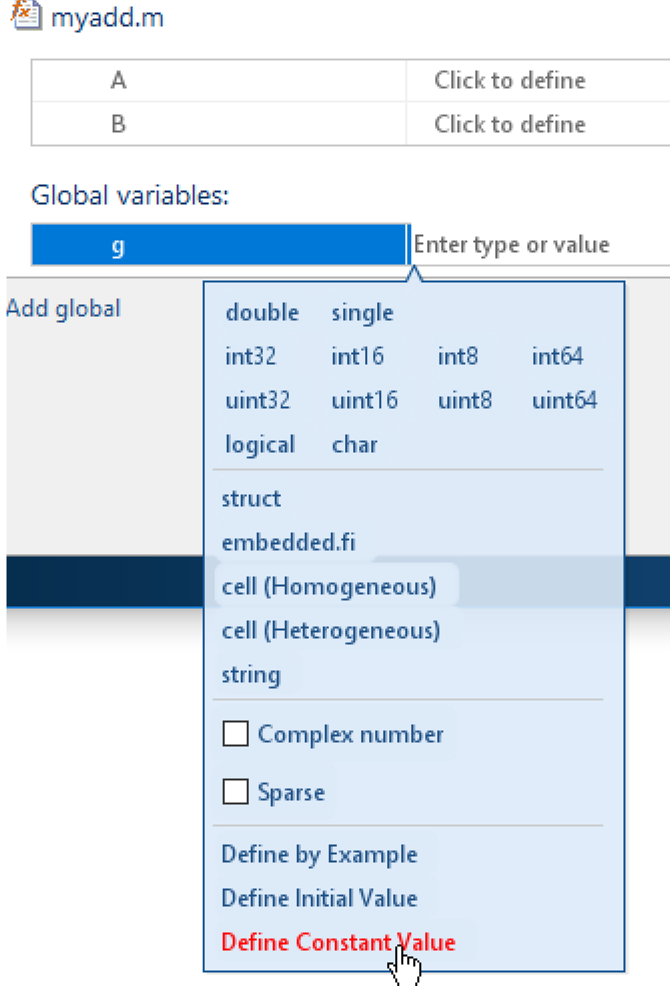
Define Constant Global Data Using the MATLAB Coder App

- 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 “View MATLAB Variables” on page 28-10.

Global Data Limitations for Generated Code

- Global structure variables cannot contain handle objects or sparse arrays.
- You cannot apply `coder.cstructname` directly to a global variable. To name the structure type to use with a global variable, use `coder.cstructname` to create a type object that names the structure type. Then, when you run `codegen`, specify that the global variable has that type. See “Name the C Structure Type to Use With a Global Structure Variable” on page 27-129.

See Also

`global`

More About

- “Specify Global Variable Type and Initial Value Using the App” on page 24-26
- “Name the C Structure Type to Use With a Global Structure Variable” on page 27-129

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 27-88

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.

See Also

More About

- “Code Generation for Enumerations” on page 14-2
- “Generate Code for an LED Control Function That Uses Enumerated Types” on page 27-131
- “Customize Enumerated Types in Generated Code” on page 14-7
- “Specify an Enumerated Type Input Parameter by Example” on page 24-10
- “Specify an Enumerated Type Input Parameter” on page 24-15

Generate Code for Variable-Size Data

In this section...

"Disable Support for Variable-Size Data" on page 27-98
"Control Dynamic Memory Allocation" on page 27-98
"Generating Code for MATLAB Functions with Variable-Size Data" on page 27-100
"Generate Code for a MATLAB Function That Expands a Vector in a Loop" on page 27-101

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 27-98), 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 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 `DynamicMemoryAllocationThreshold` parameter. Variable-size arrays whose size is less than this threshold are allocated on the stack.

3. Optionally, if you set `DynamicMemoryAllocation` to `'Threshold'`, configure `DynamicMemoryAllocationThreshold` 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 27-101

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 6-12.

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 6-6 and “Diagnosing and Fixing Size Mismatch Errors” on page 6-12.
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 27-98.

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” on page 27-101
- “Step 1: Add Compilation Directive for Code Generation” on page 27-101
- “Step 2: Address Issues Detected by the Code Analyzer” on page 27-102
- “Step 3: Generate MEX Code” on page 27-102
- “Step 4: Generate C Code” on page 27-103
- “Step 5: Specify an Upper Bound for the Output Vector” on page 27-103
- “Step 6: Change the Dynamic Memory Allocation Threshold” on page 27-104

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 27-49.
   - 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.

SUMMARY	ALL MESSAGES (0)	BUILD LOGS	CODE INSIGHTS (0)	VARIABLES

Name	Type	Size	Class
B	Output	1 × :?	double
A	Input	1 × :100	double
tol	Input	1 × 1	double
i	Local	1 × 1	double
k	Local	1 × 1	double

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 list of generated files, click 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 “Use C Arrays in the Generated Function Interfaces” on page 31-3.

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);
```

See Also

More About

- “Using Dynamic Memory Allocation for an Atoms Simulation” on page 31-52

How MATLAB Coder Partitions Generated Code

In this section...

"Partitioning Generated Files" on page 27-106
"How to Select the File Partitioning Method" on page 27-106
"Partitioning Generated Files with One C/C++ File Per MATLAB File" on page 27-106
"Generated Files and Locations" on page 27-110
"File Partitioning and Inlining" on page 27-112

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 27-112).

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 27-106. 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;
```

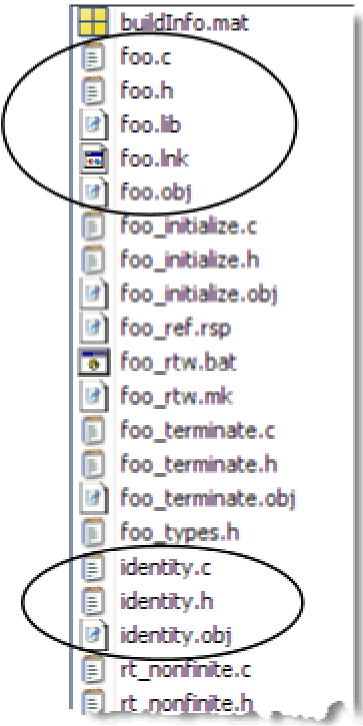
In the MATLAB Coder app, to generate a C static library for `foo.m`:

1. Define the inputs `u` and `v`. For more information, see “Specify Properties of Entry-Point Function Inputs Using the App” on page 24-3.
2. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
3. Set the **Build type** to `Static Library`
4. Click **More Settings**.
5. On the **All Settings** tab, under **Function Inlining**, set the **Inline threshold** parameter to `0`
6. Click **Close**
7. To generate the library, click **Generate**.

To generate a C static library for `foo.m`, at the command line, enter:

```
codegen  -config:lib -O disable:inline foo -args {0, 0}
% Use the -args option to specify that u and v are both
% real, scalar doubles
```

MATLAB Coder generates source, header, and object files for `foo` and `identity` in your output folder.



How MATLAB Coder Partitions Local Functions

For each local function, MATLAB Coder generates code in the same C/C++ file as the calling function. For example, suppose you define a function `foo` that calls a local function `identity`:

```
function y = foo(u,v) %#codegen
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);

function y = identity(u)
y = u;
```

To generate a C++ library, before generating code, select a C++ compiler and set C++ as your target language. For example, at the command line:

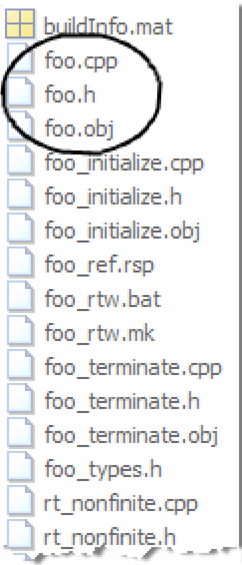
1 Select C++ as your target language:

   ```
   cfg = coder.config('lib')
   cfg.TargetLang='C++'
   ```

2 Generate the C++ library:

   ```
   codegen -config cfg   foo -args {0, 0}
   % Use the -args option to specify that u and v are both
   % real, scalar doubles
   ```

   In the primary function `foo`, MATLAB Coder inlines the code for the `identity` local function.



Note If you specify C++, MATLAB Coder wraps the C code into .cpp files so that you can use a C++ compiler and interface with external C++ applications. It does not generate C++ classes.

Here is an excerpt of the generated code in foo.cpp:

```
...
/* Function Definitions */
double foo(double u, double v)
{
  return (double)(float)u + v;
}
...
```

How MATLAB Coder Partitions Overloaded Functions

An overloaded function is a function that has multiple implementations to accommodate different classes of input. For each implementation (that is not inlined), MATLAB Coder generates a separate C/C++ file with a unique numeric suffix.

For example, suppose you define a simple function multiply_defined:

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

y = u+1;
```

You then add two more implementations of multiply_defined, one to handle inputs of type single (in an @single subfolder) and another for inputs of type double (in an @double subfolder).

To call each implementation, define the function call_multiply_defined:

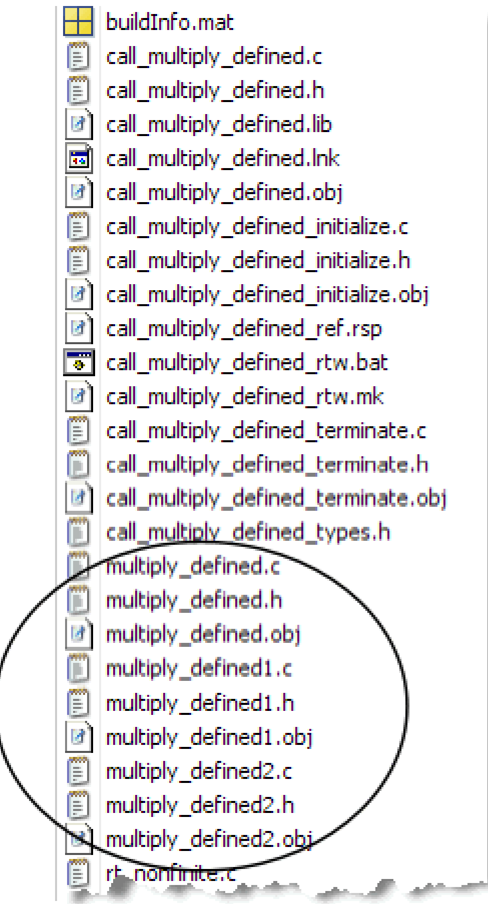
```
%#codegen
function [y1,y2,y3] = call_multiply_defined

y1 = multiply_defined(int32(2));
```

```
y2 = multiply_defined(2);
y3 = multiply_defined(single(2));
```

Next, generate C code for the overloaded function `multiply_defined`. For example, at the MATLAB command line, enter:

```
codegen -O disable:inline -config:lib call_multiply_defined
```

MATLAB Coder generates C source, header, and object files for each implementation of `multiply_defined`, as highlighted. Use numeric suffixes to create unique file names.



Generated Files and Locations

The types and locations of generated files depend on the target that you specify. For all targets, if errors or warnings occur during build or if you explicitly request a report, MATLAB Coder generates reports.

Each time MATLAB Coder generates the same type of output for the same code or project, it removes the files from the previous build. If you want to preserve files from a build, copy them to a different location before starting another build.

Generated Files for MEX Targets

By default, MATLAB Coder generates the following files for MEX function (`mex`) targets.

Type of Files	Location
Platform-specific MEX files	Current folder
MEX, and C/C++ source, header, and object files	`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`*
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.

To change	Action
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 27-106.
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 27-106.
- 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 27-113.

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.

Build Process Customization

For certain applications, you might want to control aspects of the build process that occur after C/C++ source code generation but before compilation. For example, you can specify compiler or linker options. You can get and modify all the generated source files to add a copyright disclaimer. You can control the build process in a variety of ways. Customize the build process by:

- Using the function `coder.updateBuildInfo`.
- Using the methods of an `RTW.BuildInfo` object.
- Modifying the build information by using a `coder.ExternalDependency` class.
- Modifying the build information with a script or function executed by the `PostCodeGenCommand` configuration property. This script or function is called a post-code-generation command.

All of these approaches work by altering the makefile that is generated and used to build your code. As a best practice, it is recommended to use the first three approaches, `coder.updateBuildInfo`, `RTW.BuildInfo`, and `coder.ExternalDependency`. These approaches enable you to preconfigure your MATLAB code with the build information that you require. Alternatively, the post-code generation command can provide an additional, highly customizable approach, based around an independent function or script.

The `coder.ExternalDependency` class and the post-code-generation command provide access to the build information object, `buildInfo`. You can use build information methods on `buildInfo` to configure project, build, and dependency information. MATLAB Coder creates `buildInfo` from the class `RTW.BuildInfo` at the start of the build. This object is stored in a MAT-file `buildInfo.mat` and saved in the build folder.

After code generation, you can access the build information object by loading it from `buildInfo.mat`. Do not confuse the build information object with the build configuration object, `coder.BuildConfig`, which provides specific functionality for configuring build within a `coder.ExternalDependency` class.

RTW.BuildInfo Methods

To access or write data to the build information object, use `RTW.BuildInfo` methods. Using these methods you can modify:

- Compiler options
- Linker options
- Preprocessor identifier definitions
- Source files and paths
- Include files and paths
- Precompiled external libraries
- Packaging options.

  See “Package Code for Other Development Environments” on page 31-43.

To call the methods, use the syntax:

```
method_name(buildInfo,input_arg1,...,input_argN)
```

Alternatively, you can enter:

```
buildInfo.method_name(input_arg1,...,input_argN)
```

To use the build information object after code generation is complete, load the `buildInfo.mat` file from your generated code. For example:

```
load(fullfile('.','raspberrypi_generated_code','buildInfo.mat'));
packNGo(buildInfo, 'fileName','copy_to_raspberrypi');
```

coder.updateBuildInfo Function

The `coder.updateBuildInfo` function provides a convenient way to customize the build process from within your MATLAB code. For more information and examples, see the `coder.updateBuildInfo` and `RTW.BuildInfo` reference pages.

coder.ExternalDependency Class

When you are working with external code integration or you have multiple functions that use the same build information, customize the build process by using the `coder.ExternalDependency` class. The `coder.ExternalDependency` class provides access to the build information object and methods. For more information and examples, see “Develop Interface for External C/C++ Code” on page 33-12 and the `coder.ExternalDependency` reference page.

Post-Code-Generation Command

As a best practice, customize your build process by using the first two approaches, `coder.updateBuildInfo` and `coder.ExternalDependency`. A third approach that provides additional flexibility is a post-code-generation command. A post-code-generation command is a function or script executed by the `PostCodeGenCommand` configuration object property. Set the command by using your code generation configuration object (`coder.MexCodeConfig`, `coder.CodeConfig` or `coder.EmbeddedCodeConfig`).

Command Format	Result
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.

To write the post code-generation command as a script, set `PostCodeGenCommand` to the script name. You can access the project name in the variable `projectName` and the `RTW.BuildInfo` object in the variable `buildInfo`. At the command line, enter:

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

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)';
```

For example, consider the function `setbuildargs` that takes the build information object as a parameter and adds linker options by using the `addLinkFlags` method.

```
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);
```

To use this function as a post-code-generation command, create a configuration object. Use this configuration object when you generate code. For example:

```
cfg = coder.config('dll');
cfg.PostCodeGenCommand = 'setbuildargs(buildInfo)';
codegen -config cfg foo
```

To set a post-code-generation command from 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.

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.

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.ExternalDependency` | `coder.MexCodeConfig` | `coder.updateBuildInfo`

More About

- "Configure Build Settings" on page 27-13
- "Develop Interface for External C/C++ Code" on page 33-12
- "Configure Build for External C/C++ Code" on page 33-9
- "Package Code for Other Development Environments" on page 31-43

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.

Compiler and Linker Errors

When you generate a library, MEX function, or executable from MATLAB Coder, the code generator invokes the C/C++ compiler to build a binary artifact. Build errors can occur during this process. These errors can occur during the compiling stage, or the linking stage, or at other stages of the build. You can view compiling and linking errors and warnings on the **Build Logs** tab of the code generation report.

The specific error messages and warnings that appear depend on the compiler and toolchain that you use for your platform. To see the current compiler or select a different one, at the command prompt, enter:

```
mex -setup
```

Build errors can occur for many different reasons. To diagnose and fix errors, you might have to investigate the error messages listed in your compiler documentation. Following are some commonly occurring issues that can lead to build errors when you generate code.

Failure to Specify a Main Function

Specify a main function to generate a C/C++ executable. If you do not specify a main function, a build error occurs. The main function is contained in a separate main file. When you generate code, MATLAB Coder creates an example main file, but does not automatically use it for compilation. The example main function calls the generated code with mock input values. You must modify the example main or create your own main function for realistic input and output handling.

You can specify the main file as a command-line parameter to the `codegen` command, or in the MATLAB Coder app, or by using configuration parameters. For more information and examples, see:

- “Specifying main Functions for C/C++ Executables” on page 27-11
- “Configure Build Settings” on page 27-13
- “Use an Example C Main in an Application” on page 31-25
- `codegen`

If you want the code generator to automatically use the generated example main file to build an executable for test purposes, you can set the `GenerateExampleMain` property of the configuration object to `'GenerateCodeAndCompile'`. See “Incorporate Generated Code Using an Example Main Function” on page 31-23.

Failure to Specify External Code Files

If your code uses external C functions in `coder.ceval`, then you must specify the external files containing those functions or build errors can occur. You can specify the files as command-line parameters to the `codegen` command, or in the MATLAB Coder app, or by using configuration parameters. For more information and examples, see:

- “Configure Build for External C/C++ Code” on page 33-9
- “Call C/C++ Code from MATLAB Code” on page 33-2
- `coder.ceval`
- `codegen`

Errors Caused by External Code

When you introduce external code into the build process, the external code can inject its own errors. You can introduce external code through multiple channels:

- External type definitions that you create by using `coder.opaque` that are defined in external header files.
- Structure type definitions that you create by using `coder.cstructname` that are defined in external header files.
- Calls to external code by using `coder.ceval`.
- Specification of external build files to the `codegen` command.
- Inclusion of external code files by `coder.cinclude` or `coder.updateBuildInfo`.
- Inclusion of external code through the app, on the **Custom Code** tab, or through code generation configuration parameters `CustomSource` and `CustomInclude`.

This list is not exhaustive. To address errors caused by these methods, you must examine and fix the issues with the external code or decouple the external code from your MATLAB code.

See Also

More About

- “Code Generation Reports” on page 28-7

External Websites

- https://www.mathworks.com/support.html

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.

<include>my_struct_in.m</include>

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
```

```
Code generation successful.
```

View the generated C code.

```
type codegen/lib/my_struct_in/my_struct_in.c
```

```
/*
 * File: my_struct_in.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:50:33
 */

/* Include Files */
#include "my_struct_in.h"
#include "my_struct_in_types.h"

/* Function Definitions */
/*
 * Arguments    : const struct0_T *s
 *                double y[4]
 * Return Type  : void
 */
void my_struct_in(const struct0_T *s, double y[4])
{
  y[0] = s->f[0];
  y[1] = s->f[1];
  y[2] = s->f[2];
  y[3] = s->f[3];
}

/*
 * File trailer for my_struct_in.c
 *
 * [EOF]
 */
```

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
```

```
Code generation successful.
```

View the generated C code.

```
type codegen/lib/my_struct_in/my_struct_in.c
```

```
/*
 * File: my_struct_in.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:50:38
 */

/* Include Files */
#include "my_struct_in.h"
#include "my_struct_in_types.h"

/* Function Definitions */
/*
 * Arguments    : const struct0_T s
 *                double y[4]
 * Return Type  : void
 */
void my_struct_in(const struct0_T s, double y[4])
{
  y[0] = s.f[0];
  y[1] = s.f[1];
  y[2] = s.f[2];
  y[3] = s.f[3];
}

/*
 * File trailer for my_struct_in.c
 *
 * [EOF]
 */
```

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.

<include>my_struct_out.m</include>

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
```

```
Code generation successful.
```

View the generated C code.

```
type codegen/lib/my_struct_out/my_struct_out.c
```

```
/*
 * File: my_struct_out.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:50:42
 */

/* Include Files */
#include "my_struct_out.h"
#include "my_struct_out_types.h"

/* Function Definitions */
/*
 * Arguments    : const double x[4]
 *                struct0_T *s
 * Return Type  : void
 */
void my_struct_out(const double x[4], struct0_T *s)
{
  s->f[0] = x[0];
  s->f[1] = x[1];
  s->f[2] = x[2];
  s->f[3] = x[3];
}

/*
 * File trailer for my_struct_out.c
 *
 * [EOF]
 */
```

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
```

```
Code generation successful.
```

View the generated C code.

```
type codegen/lib/my_struct_out/my_struct_out.c
```

```
/*
 * File: my_struct_out.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:50:46
 */

/* Include Files */
#include "my_struct_out.h"
#include "my_struct_out_types.h"

/* Function Definitions */
/*
 * Arguments    : const double x[4]
 * Return Type  : struct0_T
 */
struct0_T my_struct_out(const double x[4])
{
  struct0_T s;
  s.f[0] = x[0];
  s.f[1] = x[1];
  s.f[2] = x[2];
  s.f[3] = x[3];
  return s;
}

/*
 * File trailer for my_struct_out.c
 *
 * [EOF]
 */
```

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.

<include>my_struct_inout.m</include>

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
```

```
Code generation successful.
```

View the generated C code.

```
type codegen/lib/my_struct_inout/my_struct_inout.c
```

```
/*
 * File: my_struct_inout.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:50:51
 */

/* Include Files */
#include "my_struct_inout.h"
#include "my_struct_inout_types.h"

/* Function Definitions */
/*
 * Arguments    : const double x[4]
 *                const struct0_T *s
 *                double y[4]
 * Return Type  : void
 */
void my_struct_inout(const double x[4], const struct0_T *s, double y[4])
{
  double b_y;
  b_y = ((s->f[0] + s->f[1]) + s->f[2]) + s->f[3];
  y[0] = x[0] + b_y;
  y[1] = x[1] + b_y;
  y[2] = x[2] + b_y;
  y[3] = x[3] + b_y;
}

/*
 * File trailer for my_struct_inout.c
```

```
 *
 * [EOF]
 */
```

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`.

See Also

More About

- "Structure Definition for Code Generation" on page 7-2

Name the C Structure Type to Use With a Global Structure Variable

This example shows how to name the C structure type to use in code generated for a global structure.

To name the C structure type to use for a structure variable, you use `coder.cstructname`. However, you cannot apply `coder.cstructname` directly to a global variable inside a function. Instead, specify the C structure type name in one of these ways:

- At the command line, use `coder.cstructname` to create a type object that names the C structure type. When you run `codegen`, specify that the global variable has that type.
- In the MATLAB® Coder™ app, after you define and initialize a global variable, specify the C structure type name in the structure properties dialog box.

You can also use these approaches to name the C structure type for a global cell array.

Write a MATLAB Function That Uses a Global Variable

Write a MATLAB® function `getmyfield` that returns field `a` of global variable `g`.

```
type getmyfield
```

```
function y = getmyfield()
% Copyright 2018 The MathWorks, Inc.
%#codegen

global g;
y = g.a;
end
```

Specify the C Structure Type Name at the Command Line

1. Define and initialize a global structure `g`.
2. Use `coder.cstructname` to create a type object `T` that has the properties of `g` and names the generated C structure type `mytype`.
3. Generate code for `getmyfield`, specifying that `g` is a global variable with the type `T`.

```
global g
g = struct('a',5);
T = coder.cstructname(g,'mytype');
codegen -config:lib -globals {'g',T} getmyfield
```

```
Code generation successful.
```

In the generated code, `g` has the type `mytype`.

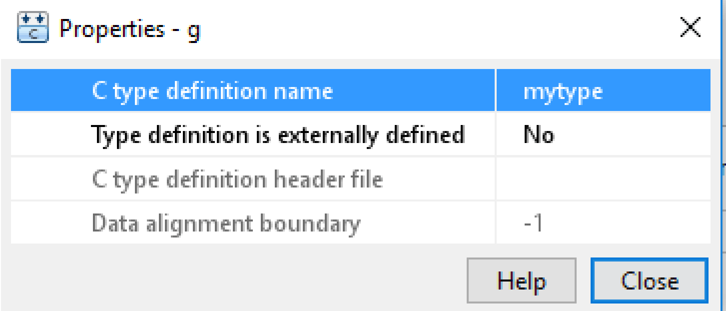
```
mytype g;
```

The generated C structure type `mytype` is:

```
typedef struct {
  double a;
} mytype;
```

Specify the C Structure Type Name in the MATLAB Coder App

1. Open the MATLAB Coder app and specify that you want to generate code for `getmyfields`.
2. On the **Define Input Types** page, Click **Add global**.
3. Click the field next to the global variable `g`. Then, click `Define Initial Value`.
4. Enter `struct('a',5)`.
5. To specify the C structure type name to use for `g`, click the gear icon.
6. In the Properties dialog box, next to **C type definition name**, enter `mytype`.



Alternatively, if you defined `g` or a type object for `g` in the workspace, you can enter `g` or the type object as the initial value.

See Also

`coder.cstructname`

More About

- “Structure Definition for Code Generation” on page 7-2
- “Generate Code for Global Data” on page 27-88
- “Specify Cell Array Inputs at the Command Line” on page 27-52

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.

<include>sysMode.m</include>

Define the enumerated type `LEDcolor`. Store it in `LEDcolor.m` on the MATLAB path.

<include>LEDcolor.m</include>

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.

<include>displayState.m</include>

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}
```

```
Code generation successful.
```

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}
```

```
Code generation successful.
```

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            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:52:31
 */

#ifndef DISPLAYSTATE_TYPES_H
```

```
#define DISPLAYSTATE_TYPES_H

/* Include Files */
#include "rtwtypes.h"

/* Type Definitions */
#ifndef enum_sysMode
#define enum_sysMode
enum sysMode
{
  OFF = 0, /* Default value */
  ON
};
#endif /* enum_sysMode */
#ifndef typedef_sysMode
#define typedef_sysMode
typedef enum sysMode sysMode;
#endif /* typedef_sysMode */

#ifndef enum_LEDcolor
#define enum_LEDcolor
enum LEDcolor
{
  GREEN = 1, /* Default value */
  RED
};
#endif /* enum_LEDcolor */
#ifndef typedef_LEDcolor
#define typedef_LEDcolor
typedef enum LEDcolor LEDcolor;
#endif /* typedef_LEDcolor */

#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)
```

See Also

More About

- "Code Generation for Enumerations" on page 14-2
- "Customize Enumerated Types in Generated Code" on page 14-7

Generate Code That Uses N-Dimensional Indexing

By default, the code generator uses one-dimensional indexing for arrays. The code generator creates one-dimensional arrays in C/C++ code for N-dimensional arrays in MATLAB code. You can use N-dimensional indexing to improve readability and adapt the interface to your generated code.

This table shows an example of the differences in the generated code with and without N-dimensional indexing.

MATLAB Code	Generated C Code (default)	Generated C Code with N-D Indexing Enabled
`A = zeros(2,4,6)`	`A[48]`	• With column-major array layout (default): `A[6][4][2]` • With row-major array layout enabled: `A[2][4][6]`

The order of the indices is reversed for N-dimensional indexing because MATLAB generates code that uses column-major array layout by default. To switch the order of the indices, you can enable row-major array layout.

Conversion of an N-dimensional array to one dimension is also called array *flattening*. In computer memory, all data is stored in terms of one-dimensional arrays. The choice of indexing does not change computation results. However, if your code has inputs or outputs that are arrays, the interface to your generated code can change.

To enable N-dimensional indexing:

- Use the `-preservearraydims` option:

  ```
  codegen foo -preservearraydims
  ```
- Set the `PreserveArrayDimensions` property for your code generation configuration object to `true`. For example:

  ```
  cfg = coder.config('lib');
  cfg.PreserveArrayDimensions = true;
  codegen foo -config cfg
  ```

To enable N-dimensional indexing from the MATLAB Coder App:

- Navigate to the **Generate Code** page in the code generation workflow.
- Open the **Generate** dialog box by clicking the **Generate** arrow .
- Click **More Settings**.
- On the **Memory** tab, select the **Preserve array dimensions** check box.

Improve Readability with N-Dimensional Indexing and Row-Major Layout

N-dimensional indexing can make it easier for you to trace your generated C/C++ code back to your MATLAB code. The code generator preserves the dimensions of the original arrays, rather than

converting arrays to one dimension. Furthermore, you can specify row-major layout to make the code appearance even more intuitive.

Consider the MATLAB function `addMatrices`, which adds two matrices, element by element:

```
function sum = addMatrices(A,B)
%#codegen
sum = coder.nullcopy(A);
for row = 1:size(A,1)
   for col = 1:size(A,2)
       sum(row,col) = A(row,col) + B(row,col);
   end
end
```

Generate code for `addMatrices` so that it operates on 2-by-4 arrays. Enable N-dimensional indexing and row-major array layout:

```
cfg = coder.config('lib');
cfg.PreserveArrayDimensions = true;
cfg.RowMajor = true;
codegen addMatrices -args {ones(2,4),ones(2,4)} -config cfg -launchreport
```

Code generation produces code with explicit two-dimensional array indexing:

```
/* N-d indexing on, row-major on */
void addMatrices(double A[2][4], double B[2][4], double sum[2][4])
{
  int row;
  int col;
  for (row = 0; row < 2; row++) {
    for (col = 0; col < 4; col++) {
      sum[row][col] = A[row][col] + B[row][col];
    }
  }
}
```

The generated code for `addMatrices` uses the same two-dimensional indexing as the original MATLAB code. You can easily analyze the generated code in comparison with the original algorithm. To understand how to use row-major layout, see “Generate Code That Uses Row-Major Array Layout” on page 37-4.

Column-Major Layout and N-Dimensional Indexing

The choice of array layout affects the appearance of N-dimensional indexing. For example, generate code for the `addMatrices` function using column-major array layout:

```
cfg.RowMajor = false;
codegen addMatrices -args {ones(2,4),ones(2,4)} -config cfg -launchreport
```

Code generation produces this C code:

```
/* N-d indexing on, row-major off */
void addMatrices(double A[4][2], double B[4][2], double sum[4][2])
{
  int row;
  int col;
  for (row = 0; row < 2; row++) {
```

```
    for (col = 0; col < 4; col++) {
      sum[col][row] = A[col][row] + B[col][row];
    }
  }
}
```

The input and output matrices in the C code are transposes of the original MATLAB matrices. To understand why, consider how arrays are represented in computer memory. The MATLAB language uses column-major layout by default, where the elements from the first (leftmost) dimension or index are contiguous in memory. C uses row-major array layout by default, where elements from the last (rightmost) dimension or index are contiguous. To preserve the original element adjacency, the code generator must reverse the order of the array dimensions.

For example, in this case, if you define the MATLAB matrix `A` as:

```
A=reshape(1:8,2,4)
```

or

```
A =
     1     3     5     7
     2     4     6     8
```

then, because MATLAB uses column-major layout, the data is internally stored in the order:

```
A(:)' =
     1     2     3     4     5     6     7     8
```

In C code, you must transpose the original data, for this example, call it `AA`:

```
AA = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
```

to attain the list of data elements with the same internal storage order. In other words, the C array must be 4-by-2. (You can obtain an equivalent storage order by defining the array as a 2-by-4, with `AA = {{1, 2, 3, 4}, {5, 6, 7, 8}}`. However, obtaining this order requires a manual reshape or rearrangement of the data.)

The choice of array layout affects only internal data representation and does not change computational or algorithmic results. To preserve the intuitive appearance of MATLAB arrays in generated code, use N-dimensional indexing with row-major array layout. Note that row-major layout can affect the efficiency of your generated code. For more information, see “Code Design for Row-Major Array Layout” on page 5-21.

Other Code Generation Considerations

Consider other aspects of N-dimensional indexing. The code generator always produces one-dimensional arrays for N-dimensional vectors, even when you specify N-dimensional indexing. For example, if you generate code for a MATLAB vector:

```
A = zeros(1,10)
```

or

```
A = zeros(1,10,1)
```

the resulting C/C++ arrays are stored as:

```
A[10]
```

N-dimensional indexing also applies to arrays and structures. For example, if you declare structures in your code as:

```
x = struct('f1', ones(2,3));
coder.cstructname(x,'myStruct1');
y = struct('f2', ones(1,6,1));
coder.cstructname(y,'myStruct2');
```

then the generated code contains the structure definitions:

```
typedef struct {
  double f1[2][3];
} myStruct1;
typedef struct {
  double f2[6];
} myStruct2;
```

Avoid linear indexing on N-dimensional arrays. Linear indexing occurs, for example, when you use the colon operator:

```
A(:)
```

To apply linear indexing, the code generator must cast an N-dimensional array into a one-dimensional array. Casting operations make your code more complex for the code generator to analyze. This increased complexity can hinder the ability of the code generator to optimize for performance.

Last, note the following:

- You can use N-dimensional indexing for arrays of any data type.
- Only fixed-size arrays, and not variable-size arrays, can use N-dimensional indexing.

See Also

`codegen` | `coder.cstructname` | `reshape`

More About

- “Generate Code That Uses Row-Major Array Layout” on page 37-4
- “Code Design for Row-Major Array Layout” on page 5-21
- “Code Generation for Variable-Size Arrays” on page 6-2
- “Preserve Variable Names in Generated Code” on page 27-38

Install OpenMP Library on macOS Platform

You can generate parallel `for`-loops on the macOS platform by using `parfor` in your MATLAB code. The code generator uses the OpenMP (Open Multiprocessing) application interface to support shared-memory, multicore code generation. To run the code generated for a `parfor`-loop outside of MATLAB, you must install an OpenMP library.

To install the OpenMP library libomp on the macOS platform, do one of the following:

- Install libomp from the LLVM download page.
  1. Navigate to the LLVM download page.
  2. Download the OpenMP source.
  3. Compile the source and install.
- Install libomp by using homebrew. At the terminal, run this command.

  ```
  brew install libomp
  ```

See Also

`parfor`

More About

- “Generate Code with Parallel for-Loops (parfor)” on page 34-31

External Websites

- https://releases.llvm.org/
- https://brew.sh/

Generate Code to Detect Edges 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.

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
```

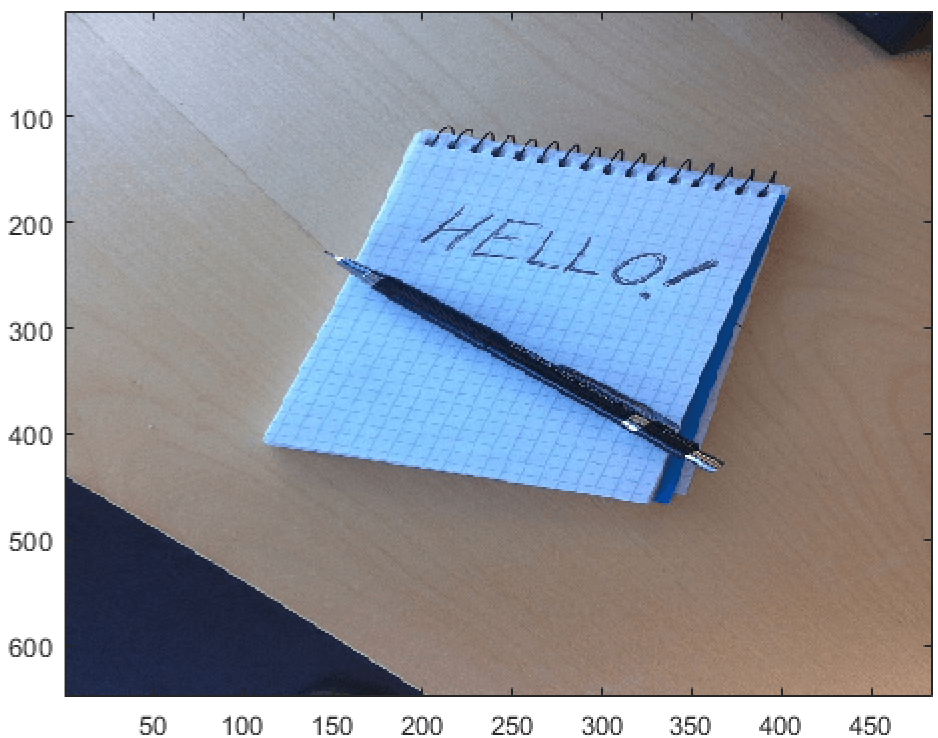
```
Code generation successful.
```

Before generating C code, you should first test the MEX function in MATLAB to ensure that it is functionally equivalent to the original MATLAB code and that no run-time errors occur. By default, `codegen` generates a MEX function named `sobel_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Read in the Original Image

Use the standard `imread` command.

```
im = imread('hello.jpg');
image(im);
```



Convert Image to a Grayscale Version

Convert the color image (shown above) to an equivalent grayscale image with normalized values (0.0 for black, 1.0 for white).

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255
```

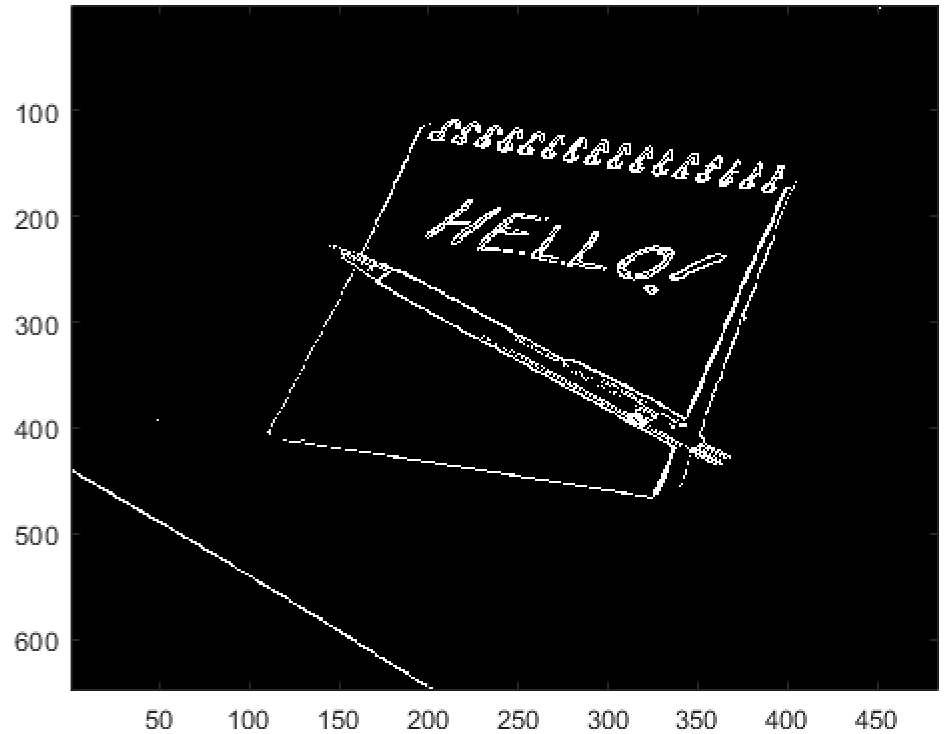
Run the MEX Function (The Sobel Filter)

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
```

```
Code generation successful.
```

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            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 13:55:47
 */

/* Include Files */
#include "sobel.h"
#include "conv2AXPYSameCMP.h"
#include "sobel_data.h"
#include "sobel_emxutil.h"
#include "sobel_initialize.h"
#include "sobel_types.h"
#include <math.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 k;
  int nx;
  if (!isInitialized_sobel) {
    sobel_initialize();
  }
  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. */
  conv2AXPYSameCMP(originalImage, H);
  b_conv2AXPYSameCMP(originalImage, V);
  nx = H->size[0] * H->size[1];
  for (k = 0; k < nx; k++) {
    H->data[k] = H->data[k] * H->data[k] + V->data[k] * V->data[k];
  }
  emxFree_real_T(&V);
  nx = H->size[0] * H->size[1];
  for (k = 0; k < nx; k++) {
    H->data[k] = sqrt(H->data[k]);
  }
  k = edgeImage->size[0] * edgeImage->size[1];
  edgeImage->size[0] = H->size[0];
  edgeImage->size[1] = H->size[1];
  emxEnsureCapacity_uint8_T(edgeImage, k);
  nx = H->size[0] * H->size[1];
  for (k = 0; k < nx; k++) {
    edgeImage->data[k] = (unsigned char)((H->data[k] > threshold) * 255U);
  }
  emxFree_real_T(&H);
}

/*
 * File trailer for sobel.c
 *
 * [EOF]
 */
```

C Code Generation for a MATLAB Kalman Filtering Algorithm

This example shows how to generate C code for a MATLAB® Kalman filter function, `kalmanfilter`, which estimates the position of a moving object based on past noisy measurements. It also shows how to generate a MEX function for this MATLAB code to increase the execution speed of the algorithm in MATLAB.

Prerequisites

There are no prerequisites for this example.

About the `kalmanfilter` Function

The `kalmanfilter` function predicts the position of a moving object based on its past values. It uses a Kalman filter estimator, a recursive adaptive filter that estimates the state of a dynamic system from a series of noisy measurements. Kalman filtering has a broad range of application in areas such as signal and image processing, control design, and computational finance.

About the Kalman Filter Estimator Algorithm

The Kalman estimator computes the position vector by computing and updating the Kalman state vector. The state vector is defined as a 6-by-1 column vector that includes position (x and y), velocity (Vx Vy), and acceleration (Ax and Ay) measurements in a 2-dimensional Cartesian space. Based on the classical laws of motion:

$$\begin{cases} X = X_0 + V_x dt \\ Y = Y_0 + V_y dt \\ V_x = V_{x0} + A_x dt \\ V_y = V_{y0} + A_y dt \end{cases}$$

The iterative formula capturing these laws are reflected in the Kalman state transition matrix "A". Note that by writing about 10 lines of MATLAB code, you can implement the Kalman estimator based on the theoretical mathematical formula found in many adaptive filtering textbooks.

```
type kalmanfilter.m
```

```
%   Copyright 2010 The MathWorks, Inc.
function y = kalmanfilter(z)
%#codegen
dt=1;
% Initialize state transition matrix
A=[ 1 0 dt 0 0 0;...     % [x  ]
       0 1 0 dt 0 0;...     % [y  ]
       0 0 1 0 dt 0;...     % [Vx]
       0 0 0 1 0 dt;...     % [Vy]
       0 0 0 0 1 0 ;...     % [Ax]
       0 0 0 0 0 1 ];       % [Ay]
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];    % Initialize measurement matrix
Q = eye(6);
R = 1000 * eye(2);
persistent x_est p_est                % Initial state conditions
if isempty(x_est)
    x_est = zeros(6, 1);             % x_est=[x,y,Vx,Vy,Ax,Ay]'
    p_est = zeros(6, 6);
end
```

```
% Predicted state and covariance
x_prd = A * x_est;
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 - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;
% Compute the estimated measurements
y = H * x_est;
end                % of the function
```

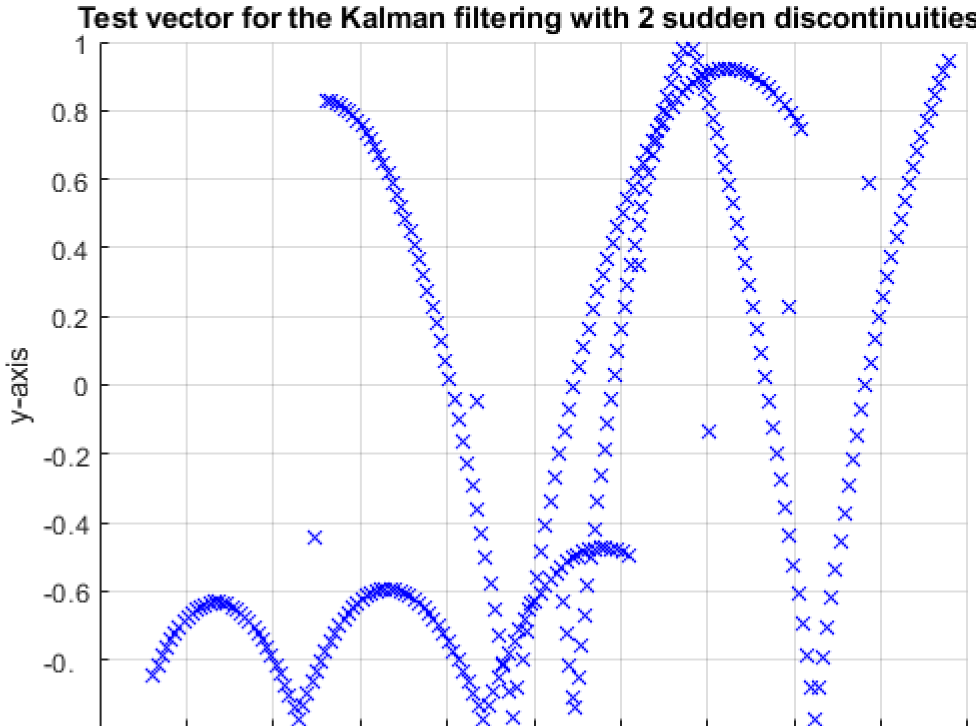
Load Test Data

The position of the object to track are recorded as x and y coordinates in a Cartesian space in a MAT file called `position_data.mat`. The following code loads the MAT file and plots the trace of the positions. The test data includes two sudden shifts or discontinuities in position which are used to check that the Kalman filter can quickly re-adjust and track the object.

```
load position_data.mat
hold; grid;
```

```
Current plot held
```

```
for idx = 1: numPts
z = position(:,idx);
plot(z(1), z(2), 'bx');
axis([-1 1 -1 1]);
end
title('Test vector for the Kalman filtering with 2 sudden discontinuities ');
xlabel('x-axis');ylabel('y-axis');
hold;
```




```
Current plot released
```

Inspect and Run the `ObjTrack` Function

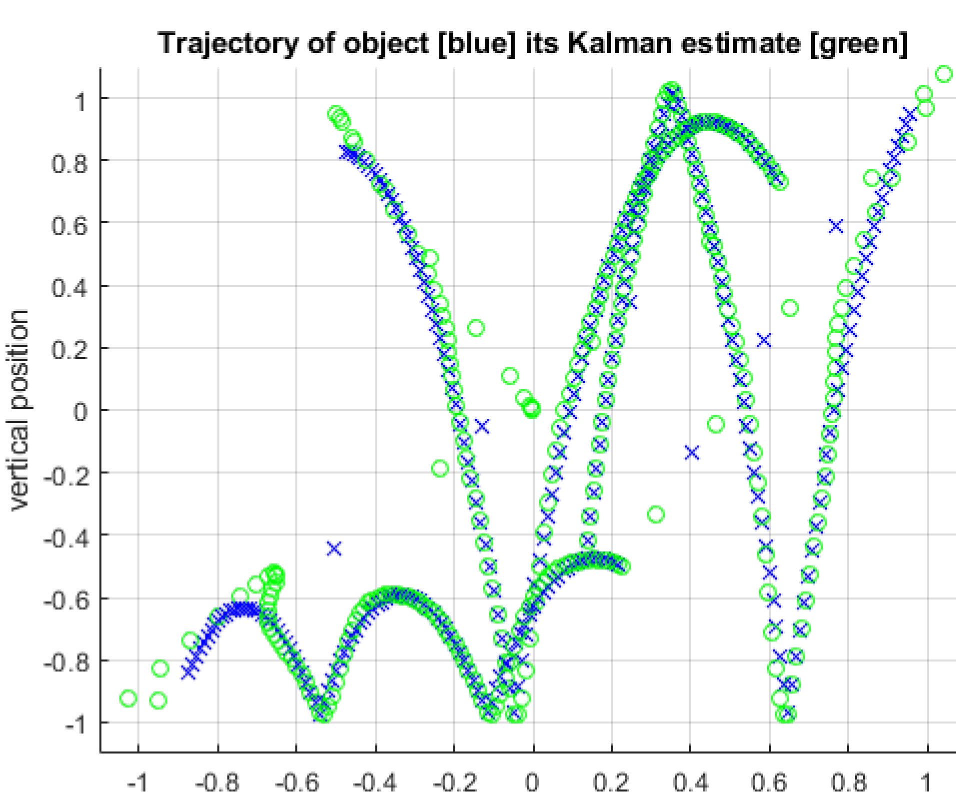
The `ObjTrack.m` function calls the Kalman filter algorithm and plots the trajectory of the object in blue and the Kalman filter estimated position in green. Initially, you see that it takes a short time for the estimated position to converge with the actual position of the object. Then, three sudden shifts in position occur. Each time the Kalman filter readjusts and tracks the object after a few iterations.

```
type ObjTrack
```

```
%   Copyright 2010 The MathWorks, Inc.
function ObjTrack(position)
%#codegen
% First, setup the figure
numPts = 300;                % Process and plot 300 samples
figure;hold;grid;             % Prepare plot window
% Main loop
for idx = 1: numPts
    z = position(:,idx);      % Get the input data
    y = kalmanfilter(z);         % Call Kalman filter to estimate the position
    plot_trajectory(z,y);     % Plot the results
end
hold;
end   % of the function
```

```
ObjTrack(position)
```

```
Current plot held
```




```
Current plot released
```

Generate C Code

The `codegen` command with the `-config:lib` option generates C code packaged as a standalone C library.

Because C uses static typing, `codegen` must determine the properties of all variables in the MATLAB files at compile time. Here, the `-args` command-line option supplies an example input so that `codegen` can infer new types based on the input types.

The `-report` option generates a compilation report that contains a summary of the compilation results and links to generated files. After compiling the MATLAB code, `codegen` provides a hyperlink to this report.

```
z = position(:,1);
codegen  -config:lib -report -c kalmanfilter.m -args {z}
```

```
Code generation successful: To view the report, open('codegen\lib\kalmanfilter\html\report.mldat
```

Inspect the Generated Code

The generated C code is in the `codegen/lib/kalmanfilter/` folder. The files are:

```
dir codegen/lib/kalmanfilter/
```

```
.                          kalmanfilter_data.h
..                         kalmanfilter_initialize.c
.gitignore                 kalmanfilter_initialize.h
_clang-format              kalmanfilter_rtw.bat
buildInfo.mat              kalmanfilter_rtw.mk
codeInfo.mat               kalmanfilter_rtw.rsp
codedescriptor.dmr         kalmanfilter_rtw_comp.rsp
compileInfo.mat            kalmanfilter_rtw_ref.rsp
defines.txt                kalmanfilter_terminate.c
examples                   kalmanfilter_terminate.h
html                       kalmanfilter_types.h
interface                  rtw_proj.tmw
kalmanfilter.c             rtwtypes.h
kalmanfilter.h             setup_msvc.bat
kalmanfilter_data.c
```

Inspect the C Code for the `kalmanfilter.c` Function

```
type codegen/lib/kalmanfilter/kalmanfilter.c
```

```
/*
 * File: kalmanfilter.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 13:54:44
 */

/* Include Files */
#include "kalmanfilter.h"
#include "kalmanfilter_data.h"
#include "kalmanfilter_initialize.h"
#include <math.h>
#include <string.h>

/* Variable Definitions */
static double x_est[6];

static double p_est[36];

/* Function Definitions */
/*
 * Arguments    : const double z[2]
 *                double y[2]
 * Return Type  : void
 */
void kalmanfilter(const double z[2], double y[2])
{
  static const short R[4] = {1000, 0, 0, 1000};
  static const signed char a[36] = {1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                    1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0,
                                    0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1};
  static const signed char iv[36] = {1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0,
                                     0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
                                     0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1};
  static const signed char c_a[12] = {1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};
  static const signed char iv1[12] = {1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0};
  double b_a[36];
  double p_prd[36];
```

```
double B[12];
double Y[12];
double x_prd[6];
double S[4];
double b_z[2];
double a21;
double a22;
double a22_tmp;
double d;
double d1;
int i;
int k;
int r1;
int r2;
signed char Q[36];
if (!isInitialized_kalmanfilter) {
  kalmanfilter_initialize();
}
/*    Copyright 2010 The MathWorks, Inc. */
/*  Initialize state transition matrix */
/*      % [x  ] */
/*      % [y  ] */
/*      % [Vx] */
/*      % [Vy] */
/*      % [Ax] */
/*  [Ay] */
/*  Initialize measurement matrix */
for (i = 0; i < 36; i++) {
  Q[i] = 0;
}
/*  Initial state conditions */
/*  Predicted state and covariance */
for (k = 0; k < 6; k++) {
  Q[k + 6 * k] = 1;
  x_prd[k] = 0.0;
  for (i = 0; i < 6; i++) {
    r1 = k + 6 * i;
    x_prd[k] += (double)a[r1] * x_est[i];
    d = 0.0;
    for (r2 = 0; r2 < 6; r2++) {
      d += (double)a[k + 6 * r2] * p_est[r2 + 6 * i];
    }
    b_a[r1] = d;
  }
}
for (i = 0; i < 6; i++) {
  for (r2 = 0; r2 < 6; r2++) {
    d = 0.0;
    for (r1 = 0; r1 < 6; r1++) {
      d += b_a[i + 6 * r1] * (double)iv[r1 + 6 * r2];
    }
    r1 = i + 6 * r2;
    p_prd[r1] = d + (double)Q[r1];
  }
}
/*  Estimation */
for (i = 0; i < 2; i++) {
  for (r2 = 0; r2 < 6; r2++) {
```

```
      d = 0.0;
      for (r1 = 0; r1 < 6; r1++) {
        d += (double)c_a[i + (r1 << 1)] * p_prd[r2 + 6 * r1];
      }
      B[i + (r2 << 1)] = d;
    }
    for (r2 = 0; r2 < 2; r2++) {
      d = 0.0;
      for (r1 = 0; r1 < 6; r1++) {
        d += B[i + (r1 << 1)] * (double)iv1[r1 + 6 * r2];
      }
      r1 = i + (r2 << 1);
      S[r1] = d + (double)R[r1];
    }
  }
  if (fabs(S[1]) > fabs(S[0])) {
    r1 = 1;
    r2 = 0;
  } else {
    r1 = 0;
    r2 = 1;
  }
  a21 = S[r2] / S[r1];
  a22_tmp = S[r1 + 2];
  a22 = S[r2 + 2] - a21 * a22_tmp;
  for (k = 0; k < 6; k++) {
    i = k << 1;
    d = B[r1 + i];
    d1 = (B[r2 + i] - d * a21) / a22;
    Y[i + 1] = d1;
    Y[i] = (d - d1 * a22_tmp) / S[r1];
  }
  for (i = 0; i < 2; i++) {
    for (r2 = 0; r2 < 6; r2++) {
      B[r2 + 6 * i] = Y[i + (r2 << 1)];
    }
  }
  /*  Estimated state and covariance */
  for (i = 0; i < 2; i++) {
    d = 0.0;
    for (r2 = 0; r2 < 6; r2++) {
      d += (double)c_a[i + (r2 << 1)] * x_prd[r2];
    }
    b_z[i] = z[i] - d;
  }
  for (i = 0; i < 6; i++) {
    d = B[i + 6];
    x_est[i] = x_prd[i] + (B[i] * b_z[0] + d * b_z[1]);
    for (r2 = 0; r2 < 6; r2++) {
      r1 = r2 << 1;
      b_a[i + 6 * r2] = B[i] * (double)c_a[r1] + d * (double)c_a[r1 + 1];
    }
    for (r2 = 0; r2 < 6; r2++) {
      d = 0.0;
      for (r1 = 0; r1 < 6; r1++) {
        d += b_a[i + 6 * r1] * p_prd[r1 + 6 * r2];
      }
      r1 = i + 6 * r2;
```

```
      p_est[r1] = p_prd[r1] - d;
    }
  }
  /*  Compute the estimated measurements */
  for (i = 0; i < 2; i++) {
    d = 0.0;
    for (r2 = 0; r2 < 6; r2++) {
      d += (double)c_a[i + (r2 << 1)] * x_est[r2];
    }
    y[i] = d;
  }
}

/*
 * Arguments    : void
 * Return Type  : void
 */
void kalmanfilter_init(void)
{
  int i;
  for (i = 0; i < 6; i++) {
    x_est[i] = 0.0;
  }
  /*  x_est=[x,y,Vx,Vy,Ax,Ay]' */
  memset(&p_est[0], 0, 36U * sizeof(double));
}

/*
 * File trailer for kalmanfilter.c
 *
 * [EOF]
 */
```

Accelerate the Execution Speed of the MATLAB Algorithm

You can accelerate the execution speed of the `kalmanfilter` function that is processing a large data set by using the `codegen` command to generate a MEX function from the MATLAB code.

Call the `kalman_loop` Function to Process Large Data Sets

First, run the Kalman algorithm with a large number of data samples in MATLAB. The `kalman_loop` function runs the `kalmanfilter` function in a loop. The number of loop iterations is equal to the second dimension of the input to the function.

```
type kalman_loop
```

```
%   Copyright 2010 The MathWorks, Inc.
function y=kalman_loop(z)
% Call Kalman estimator in the loop for large data set testing
%#codegen
[DIM, LEN]=size(z);
y=zeros(DIM,LEN);           % Initialize output
for n=1:LEN                     % Output in the loop
    y(:,n)=kalmanfilter(z(:,n));
end;
```

Baseline Execution Speed Without Compilation

Now time the MATLAB algorithm. Use the `randn` command to generate random numbers and create the input matrix `position` composed of 100,000 samples of (2x1) position vectors. Remove all MEX files from the current folder. Use the MATLAB stopwatch timer (`tic` and `toc` commands) to measure how long it takes to process these samples when running the `kalman_loop` function.

```
clear mex
delete(['*.' mexext])
position = randn(2,100000);
tic, kalman_loop(position); a=toc;
```

Generate a MEX Function for Testing

Next, generate a MEX function using the command `codegen` followed by the name of the MATLAB function `kalman_loop`. The `codegen` command generates a MEX function called `kalman_loop_mex`. You can then compare the execution speed of this MEX function with that of the original MATLAB algorithm.

```
codegen -args {position} kalman_loop.m
```

```
Code generation successful.
```

```
which kalman_loop_mex
```

```
C:\TEMP\Bdoc21a_1606923_4784\ib8756D3\10\tp55acccf3\coder-ex53054096\kalman_loop_mex.mexw64
```

Time the MEX Function

Now, time the MEX function `kalman_loop_mex`. Use the same signal `position` as before as the input, to ensure a fair comparison of the execution speed.

```
tic, kalman_loop_mex(position); b=toc;
```

Comparison of the Execution Speeds

Notice the speed execution difference using a generated MEX function.

```
display(sprintf('The speedup is %.1f times using the generated MEX over the baseline MATLAB funct
```

```
The speedup is 25.3 times using the generated MEX over the baseline MATLAB function.
```

Generate Code to Optimize Portfolio by Using Black Litterman Approach

This example shows how to generate a MEX function and C source code from MATLAB® code that performs portfolio optimization using the Black Litterman approach.

Prerequisites

There are no prerequisites for this example.

About the `hlblacklitterman` Function

The `hlblacklitterman.m` function reads in financial information regarding a portfolio and performs portfolio optimization using the Black Litterman approach.

```
type hlblacklitterman
```

```
function [er, ps, w, pw, lambda, theta] = hlblacklitterman(delta, weq, sigma, tau, P, Q, Omega)%#
% hlblacklitterman
%   This function performs the Black-Litterman blending of the prior
%   and the views into a new posterior estimate of the returns as
%   described in the paper by He and Litterman.
% Inputs
%   delta  - Risk tolerance from the equilibrium portfolio
%   weq    - Weights of the assets in the equilibrium portfolio
%   sigma  - Prior covariance matrix
%   tau    - Coefficiet of uncertainty in the prior estimate of the mean (pi)
%   P      - Pick matrix for the view(s)
%   Q      - Vector of view returns
%   Omega  - Matrix of variance of the views (diagonal)
% Outputs
%   Er     - Posterior estimate of the mean returns
%   w      - Unconstrained weights computed given the Posterior estimates
%            of the mean and covariance of returns.
%   lambda - A measure of the impact of each view on the posterior estimates.
%   theta  - A measure of the share of the prior and sample information in the
%            posterior precision.

% Reverse optimize and back out the equilibrium returns
% This is formula (12) page 6.
pi = weq * sigma * delta;
% We use tau * sigma many places so just compute it once
ts = tau * sigma;
% Compute posterior estimate of the mean
% This is a simplified version of formula (8) on page 4.
er = pi' + ts * P' * inv(P * ts * P' + Omega) * (Q - P * pi');
% We can also do it the long way to illustrate that d1 + d2 = I
d = inv(inv(ts) + P' * inv(Omega) * P);
d1 = d * inv(ts);
d2 = d * P' * inv(Omega) * P;
er2 = d1 * pi' + d2 * pinv(P) * Q;
% Compute posterior estimate of the uncertainty in the mean
% This is a simplified and combined version of formulas (9) and (15)
ps = ts - ts * P' * inv(P * ts * P' + Omega) * P * ts;
posteriorSigma = sigma + ps;
% Compute the share of the posterior precision from prior and views,
% then for each individual view so we can compare it with lambda
```

```
theta=zeros(1,2+size(P,1));
theta(1,1) = (trace(inv(ts) * ps) / size(ts,1));
theta(1,2) = (trace(P'*inv(Omega)*P* ps) / size(ts,1));
for i=1:size(P,1)
    theta(1,2+i) = (trace(P(i,:)'*inv(Omega(i,i))*P(i,:)* ps) / size(ts,1));
end
% Compute posterior weights based solely on changed covariance
w = (er' * inv(delta * posteriorSigma))';
% Compute posterior weights based on uncertainty in mean and covariance
pw = (pi * inv(delta * posteriorSigma))';
% Compute lambda value
% We solve for lambda from formula (17) page 7, rather than formula (18)
% just because it is less to type, and we've already computed w*.
lambda = pinv(P)' * (w'*(1+tau) - weq)';
end

% Black-Litterman example code for MatLab (hlblacklitterman.m)
% Copyright (c) Jay Walters, blacklitterman.org, 2008.
%
% Redistribution and use in source and binary forms,
% with or without modification, are permitted provided
% that the following conditions are met:
%
% Redistributions of source code must retain the above
% copyright notice, this list of conditions and the following
% disclaimer.
%
% Redistributions in binary form must reproduce the above
% copyright notice, this list of conditions and the following
% disclaimer in the documentation and/or other materials
% provided with the distribution.
%
% Neither the name of blacklitterman.org nor the names of its
% contributors may be used to endorse or promote products
% derived from this software without specific prior written
% permission.
%
% THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
% CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
% INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
% DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
% CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
% SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
% BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
% SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
% INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
% WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
% NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
% OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
% DAMAGE.
%
% This program uses the examples from the paper "The Intuition
% Behind Black-Litterman Model  Portfolios", by He and Litterman,
% 1999.  You can find a copy of this  paper at the following url.
%     http:%papers.ssrn.com/sol3/papers.cfm?abstract_id=334304
%
% For more details on the Black-Litterman model you can also view
```

```
% "The BlackLitterman Model: A Detailed Exploration", by this author
% at the following url.
%     http:%www.blacklitterman.org/Black-Litterman.pdf
%
```

The `%#codegen` directive indicates that the MATLAB code is intended for code generation.

Generate the MEX Function for Testing

Generate a MEX function using the `codegen` command.

```
codegen hlblacklitterman -args {0, zeros(1, 7), zeros(7,7), 0, zeros(1, 7), 0, 0}
```

```
Code generation successful.
```

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 `hlblacklitterman_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Run the MEX Function

Call the generated MEX function

```
testMex();
```

```
View 1
Country        P       mu      w*
Australia        0     4.328     1.524
Canada           0     7.576     2.095
France       -29.5     9.288    -3.948
Germany        100     11.04     35.41
Japan            0     4.506     11.05
UK           -70.5     6.953    -9.462
USA              0     8.069     58.57
q                5
omega/tau        0.0213
lambda           0.317
theta          0.0714
pr theta         0.929


View 1
Country        P       mu      w*
Australia        0     4.328     1.524
Canada           0     7.576     2.095
France       -29.5     9.288    -3.948
Germany        100     11.04     35.41
Japan            0     4.506     11.05
UK           -70.5     6.953    -9.462
USA              0     8.069     58.57
q                5
omega/tau        0.0213
lambda           0.317
theta          0.0714
pr theta         0.929

Execution Time - MATLAB function: 0.32216 seconds
Execution Time - MEX function   : 0.0053331 seconds
```

Generate C Code

```
cfg = coder.config('lib');
codegen -config cfg hlblacklitterman  -args {0, zeros(1, 7), zeros(7,7), 0, zeros(1, 7), 0, 0}
```

```
Code generation successful.
```

Using `codegen` with the specified `-config cfg` option produces a standalone C library.

Inspect the Generated Code

By default, the code generated for the library is in the folder `codegen/lib/hbblacklitterman/`.

The files are:

```
dir codegen/lib/hlblacklitterman/
```

```
.                                   hlblacklitterman_terminate.c
..                                  hlblacklitterman_terminate.h
.gitignore                          hlblacklitterman_terminate.obj
_clang-format                       hlblacklitterman_types.h
buildInfo.mat                       interface
codeInfo.mat                        inv.c
codedescriptor.dmr                  inv.h
compileInfo.mat                     inv.obj
defines.txt                         pinv.c
examples                            pinv.h
hlblacklitterman.c                  pinv.obj
hlblacklitterman.h                  rtGetInf.c
hlblacklitterman.lib                rtGetInf.h
hlblacklitterman.obj                rtGetInf.obj
hlblacklitterman_data.c             rtGetNaN.c
hlblacklitterman_data.h             rtGetNaN.h
hlblacklitterman_data.obj           rtGetNaN.obj
hlblacklitterman_initialize.c       rt_nonfinite.c
hlblacklitterman_initialize.h       rt_nonfinite.h
hlblacklitterman_initialize.obj     rt_nonfinite.obj
hlblacklitterman_rtw.bat            rtw_proj.tmw
hlblacklitterman_rtw.mk             rtwtypes.h
hlblacklitterman_rtw.rsp            setup_msvc.bat
hlblacklitterman_rtw_comp.rsp
hlblacklitterman_rtw_ref.rsp
```

Inspect the C Code for the `hlblacklitterman.c` Function

```
type codegen/lib/hlblacklitterman/hlblacklitterman.c
```

```
/*
 * File: hlblacklitterman.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:51:06
 */

/* Include Files */
#include "hlblacklitterman.h"
#include "inv.h"
#include "pinv.h"
#include "rt_nonfinite.h"
```

```
/* Function Definitions */
/*
 * hlblacklitterman
 *    This function performs the Black-Litterman blending of the prior
 *    and the views into a new posterior estimate of the returns as
 *    described in the paper by He and Litterman.
 *  Inputs
 *    delta  - Risk tolerance from the equilibrium portfolio
 *    weq    - Weights of the assets in the equilibrium portfolio
 *    sigma  - Prior covariance matrix
 *    tau    - Coefficiet of uncertainty in the prior estimate of the mean (pi)
 *    P      - Pick matrix for the view(s)
 *    Q      - Vector of view returns
 *    Omega  - Matrix of variance of the views (diagonal)
 *  Outputs
 *    Er     - Posterior estimate of the mean returns
 *    w      - Unconstrained weights computed given the Posterior estimates
 *             of the mean and covariance of returns.
 *    lambda - A measure of the impact of each view on the posterior estimates.
 *    theta  - A measure of the share of the prior and sample information in the
 *             posterior precision.
 *
 * Arguments    : double delta
 *                const double weq[7]
 *                const double sigma[49]
 *                double tau
 *                const double P[7]
 *                double Q
 *                double Omega
 *                double er[7]
 *                double ps[49]
 *                double w[7]
 *                double pw[7]
 *                double *lambda
 *                double theta[3]
 * Return Type  : void
 */
void hlblacklitterman(double delta, const double weq[7], const double sigma[49],
                      double tau, const double P[7], double Q, double Omega,
                      double er[7], double ps[49], double w[7], double pw[7],
                      double *lambda, double theta[3])
{
  double b_er_tmp[49];
  double dv[49];
  double posteriorSigma[49];
  double ts[49];
  double b_y_tmp[7];
  double er_tmp[7];
  double pi[7];
  double unusedExpr[7];
  double b;
  double b_P;
  double b_b;
  double d;
  double y_tmp;
  int i;
  int i1;
```

```
int ps_tmp;
/*  Reverse optimize and back out the equilibrium returns */
/*  This is formula (12) page 6. */
for (i = 0; i < 7; i++) {
  b = 0.0;
  for (i1 = 0; i1 < 7; i1++) {
    b += weq[i1] * sigma[i1 + 7 * i];
  }
  pi[i] = b * delta;
}
/*  We use tau * sigma many places so just compute it once */
for (i = 0; i < 49; i++) {
  ts[i] = tau * sigma[i];
}
/*  Compute posterior estimate of the mean */
/*  This is a simplified version of formula (8) on page 4. */
y_tmp = 0.0;
b_P = 0.0;
for (i = 0; i < 7; i++) {
  b = 0.0;
  b_b = 0.0;
  for (i1 = 0; i1 < 7; i1++) {
    d = P[i1];
    b += ts[i + 7 * i1] * d;
    b_b += d * ts[i1 + 7 * i];
  }
  b_y_tmp[i] = b_b;
  er_tmp[i] = b;
  b = P[i];
  y_tmp += b_b * b;
  b_P += b * pi[i];
}
b_b = 1.0 / (y_tmp + Omega);
b = Q - b_P;
for (i = 0; i < 7; i++) {
  er[i] = pi[i] + er_tmp[i] * b_b * b;
}
/*  We can also do it the long way to illustrate that d1 + d2 = I */
y_tmp = 1.0 / Omega;
pinv(P, unusedExpr);
/*  Compute posterior estimate of the uncertainty in the mean */
/*  This is a simplified and combined version of formulas (9) and (15) */
b = 0.0;
for (i = 0; i < 7; i++) {
  b += b_y_tmp[i] * P[i];
}
b_b = 1.0 / (b + Omega);
for (i = 0; i < 7; i++) {
  for (i1 = 0; i1 < 7; i1++) {
    b_er_tmp[i1 + 7 * i] = er_tmp[i1] * b_b * P[i];
  }
}
for (i = 0; i < 7; i++) {
  for (i1 = 0; i1 < 7; i1++) {
    b = 0.0;
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
      b += b_er_tmp[i + 7 * ps_tmp] * ts[ps_tmp + 7 * i1];
    }
```

```
      ps_tmp = i + 7 * i1;
      ps[ps_tmp] = ts[ps_tmp] - b;
    }
  }
  for (i = 0; i < 49; i++) {
    posteriorSigma[i] = sigma[i] + ps[i];
  }
  /*  Compute the share of the posterior precision from prior and views, */
  /*  then for each individual view so we can compare it with lambda */
  inv(ts, dv);
  for (i = 0; i < 7; i++) {
    for (i1 = 0; i1 < 7; i1++) {
      b = 0.0;
      for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        b += dv[i + 7 * ps_tmp] * ps[ps_tmp + 7 * i1];
      }
      ts[i + 7 * i1] = b;
    }
  }
  b = 0.0;
  for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    b += ts[ps_tmp + 7 * ps_tmp];
  }
  theta[0] = b / 7.0;
  for (i = 0; i < 7; i++) {
    for (i1 = 0; i1 < 7; i1++) {
      b_er_tmp[i1 + 7 * i] = P[i1] * y_tmp * P[i];
    }
  }
  for (i = 0; i < 7; i++) {
    for (i1 = 0; i1 < 7; i1++) {
      b = 0.0;
      for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        b += b_er_tmp[i + 7 * ps_tmp] * ps[ps_tmp + 7 * i1];
      }
      ts[i + 7 * i1] = b;
    }
  }
  b = 0.0;
  for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    b += ts[ps_tmp + 7 * ps_tmp];
  }
  theta[1] = b / 7.0;
  for (i = 0; i < 7; i++) {
    for (i1 = 0; i1 < 7; i1++) {
      b_er_tmp[i1 + 7 * i] = P[i1] * y_tmp * P[i];
    }
  }
  for (i = 0; i < 7; i++) {
    for (i1 = 0; i1 < 7; i1++) {
      b = 0.0;
      for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        b += b_er_tmp[i + 7 * ps_tmp] * ps[ps_tmp + 7 * i1];
      }
      ts[i + 7 * i1] = b;
    }
  }
  b = 0.0;
```

```
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
      b += ts[ps_tmp + 7 * ps_tmp];
    }
    theta[2] = b / 7.0;
    /*  Compute posterior weights based solely on changed covariance */
    for (i = 0; i < 49; i++) {
      b_er_tmp[i] = delta * posteriorSigma[i];
    }
    inv(b_er_tmp, dv);
    for (i = 0; i < 7; i++) {
      b = 0.0;
      for (i1 = 0; i1 < 7; i1++) {
        b += er[i1] * dv[i1 + 7 * i];
      }
      w[i] = b;
    }
    /*  Compute posterior weights based on uncertainty in mean and covariance */
    for (i = 0; i < 49; i++) {
      posteriorSigma[i] *= delta;
    }
    inv(posteriorSigma, dv);
    for (i = 0; i < 7; i++) {
      b = 0.0;
      for (i1 = 0; i1 < 7; i1++) {
        b += pi[i1] * dv[i1 + 7 * i];
      }
      pw[i] = b;
    }
    /*  Compute lambda value */
    /*  We solve for lambda from formula (17) page 7, rather than formula (18) */
    /*  just because it is less to type, and we've already computed w*. */
    pinv(P, er_tmp);
    *lambda = 0.0;
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
      *lambda += er_tmp[ps_tmp] * (w[ps_tmp] * (tau + 1.0) - weq[ps_tmp]);
    }
  }

  /*
   * File trailer for hlblacklitterman.c
   *
   * [EOF]
   */
```

Generate Code for Persistent Variables

This example shows how to generate a MEX function from a MATLAB® function, `compute_average`, that uses persistent variables. It illustrates that you must clear the state of persistent variables before using the function to compute the average of a new set of values.

This example also shows how to initialize and terminate the state of the persistent variables for the same MATLAB function in standalone generated code. You must clear the state of persistent variables in the generated code before using the function to compute the average of a new set of values.

Prerequisites

There are no prerequisites for this example.

About the `compute_average` Function

The `compute_average.m` function uses two persistent variables, the accumulated sum and the number of values added so far, so that you can call the function with one value at a time.

```
type compute_average
```

```
% y = compute_average(x)
% This function takes an input scalar value 'x' and returns the average
% value so far.
function y = compute_average(x) %#codegen
assert(isa(x,'double')); % Input is scalar double

% Declare two persistent variables 'sum' and 'cnt'.
persistent sum cnt;

% Upon the first call we need to initialize the variables.
if isempty(sum)
    sum = 0;
    cnt = 0;
end

% Compute the accumulated sum and the number of values so far.
sum = sum + x;
cnt = cnt + 1;

% Return the current average.
y = sum / cnt;
```

The `%#codegen` directive indicates that the MATLAB code is intended for code generation.

Generate the MEX Function

First, generate a MEX function using the command `codegen` followed by the name of the MATLAB file to compile.

```
codegen compute_average
```

```
Code generation successful.
```

By default, `codegen` generates a MEX function named `hello_world_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Run the MEX Function

(10 + 20 + 100) / 3 = 43.3333

```
compute_average_mex(10)
```

```
ans = 10
```

```
compute_average_mex(20)
```

```
ans = 15
```

```
compute_average_mex(100)
```

```
ans = 43.3333
```

Clear the Internal State of Persistent Variables

Clear the persistent variables by using the `clear mex` command.

```
clear mex
```

Run the MEX Function Again to Calculate the Average of a Different Set of Values

(10 + 20 + 30 + 40) / 4 = 25

```
compute_average_mex(10)
```

```
ans = 10
```

```
compute_average_mex(20)
```

```
ans = 15
```

```
compute_average_mex(30)
```

```
ans = 20
```

```
compute_average_mex(40)
```

```
ans = 25
```

Clear the Internal State of Persistent Variables in Standalone Generated Code

The states of persistent variables in standalone generated code are cleared by calling the initiate and terminate functions in the main function.These functions are generated by the code generator. These files are in the `codegen` directory.

You can edit the example main file, `main.c` to invoke the initiate and terminate functions. For example:

```
type main.c
```

```
/*
 * File: main.c
 */
/* Include Files */
#include "main.h"
#include "compute_average.h"
#include "compute_average_terminate.h"
```

```
#include "compute_average_initialize.h"

/* Function Declarations */
static double argInit_real_T(void);
static void main_compute_average(void);

/* Function Definitions */
/*
 * Arguments    : void
 * Return Type  : double
 */
static double argInit_real_T(void)
{
  return 0.0;
}

/*
 * Arguments    : void
 * Return Type  : void
 */
static void main_compute_average(void)
{
  double y;

  /* Initialize function 'compute_average' input arguments. */
  /* Call the entry-point 'compute_average'. */
  y = compute_average(argInit_real_T());
}

/*
 * Arguments    : int argc
 *                const char * const argv[]
 * Return Type  : int
 */
int main(int argc, const char * const argv[])
{
  (void)argc;
  (void)argv;

  /* Initialize the entry-point function. */
  compute_average_initiatlize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_compute_average();

  /* Terminate the application. */
  compute_average_terminate();

  /*Once the application is terminated, the state of the persistent variables is cleared. */

  /* Re-initialize the entry-point function. */
  compute_average_initialize();

  /* You can run the application for a new set of values.*/
  main_compute_average();

  /* Terminate the application after your process is complete.*/
```

```
  compute_average_terminate();

  return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */
```

As you can see, the `main.c` file has been edited to call the terminate function, `compute_average_terminate()` to clear the state of the persistent variables. A new set of computations is run by calling `compute_average_initialize()` and `main_compute_average()` with a new set of values.

Generate Code for Structure Arrays

This example shows how to write a MATLAB® function that uses structure arrays so that it is suitable for code generation. For code generation, you must first create a scalar template version of the structure before growing it into an array. The code generation inference engine uses the type of this scalar value as the base type of the array.

Prerequisites

There are no prerequisites for this example.

About the `struct_array` Function

The `struct_array.m` file uses a structure array.

```
type struct_array
```

```
% y = struct_array(n)
% Take an input scalar number 'n' which will designate the size of the
% structure array return.
function y = struct_array(n) %#codegen

%   Copyright 2010-2013 The MathWorks, Inc.

assert(isa(n,'double')); % Input is scalar double

% To create a structure array you start to define the base scalar element
% first. Typically, we initialize all the fields with "dummy" (or zero)
% values so the type/shape of all its contents are well defined.
s.x = 0;
s.y = 0;
s.vx = 0;
s.vy = 0;

% To create a structure array of fixed size you can do this in multiple
% ways. One example is to use the library function 'repmat' which takes a
% scalar element and repeats it to its desired size.
arr1 = repmat(s, 3, 5); % Creates a 3x5 matrix of structure 's'

% At this point you can now modify the fields of this structure array.
arr1(2,3).x = 10;
arr1(2,3).y = 20;
arr1(2,4).x = 5;
arr1(2,4).y = 7;

% Another way of creating a structure array of fixed size is to use the
% concatenation operator.
arr2 = [s s s; s s s; s s s; s s s; s s s];

% If two variables agree on base type and shape you can copy one structure
% array to the other using standard assignment.
arr2 = arr1;

% To create a structure array of variable size with a known upper bound can
% be done in multiple ways as well. Again, we can use repmat for this, but
% this time we will add a constraint to the (non constant) input variable.
% This guarantees that the input 'n' of this function is less than or equal to 10.
```

```
assert(n <= 10);

% Create a row vector with at most 10 elements of structures based on 's'
arr3 = repmat(s, 1, n);

% Or we can use a for-loop with the concatenation operator. The compiler is
% unable to analyze that 'arr4' will be at most 10 elements big, so we
% add a hint on 'arr4' using coder.varsize. This will specify that the
% dimensions of 'arr4' is exactly one row with at most 10 columns. Look at
% the documentation for coder.varsize for further information.
coder.varsize('arr4', [1 10]);
arr4 = repmat(s, 1, 0);
for i = 1:n
    arr4 = [arr4 s];
end

% Let the top-level function return 'arr4'.
y = arr4;
```

In MATLAB, when building up a structure array, you would typically just add fields as you go. For example, s(1).x = 10; s(2).y = 20; This "dynamic" style of building structures is not supported for code generation. One reason is that it is possible in MATLAB to have different structure fields for two different elements of a structure array, which conflicts with the more static approach of type inference. Therefore, you need to fully specify the base scalar element first, and then grow a structure array from this fully specified element. This method guarantees that two elements of a structure array always share the same type (fields).

Generate the MEX Function

Generate a MEX function using the command `codegen` followed by the name of the MATLAB file to compile.

```
codegen struct_array
```

```
Code generation successful.
```

By default, `codegen` generates a MEX function named `struct_array_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Run the MEX Function

```
struct_array_mex(10)
```

```
ans=1×10 struct array with fields:
    x
    y
    vx
    vy
```

Add Custom Toolchains to MATLAB® Coder™ Build Process

This example shows how to register and use a toolchain to compile an executable. This example uses Intel® Compiler, but the concepts and API shown below can be used for any toolchain. The registered toolchain can be selected from a list of toolchains and a makefile will be generated to build the code using that toolchain.

About the coderrand Function

In this example, you generate code for the `coderrand` function. This MATLAB® function simply generates a random scalar value from the standard uniform distribution on the open interval (0,1).

```
type coderrand
```

```
function y = coderrand %#codegen

%   Copyright 2012 The MathWorks, Inc.

y = rand();
```

Toolchain Info

A toolchain is a collection of tools that is required for compiling and linking generated code for a specified platform. A toolchain has multiple tools, such as a compiler, linker and archiver. Each of these tools can take multiple options, which can be grouped into configurations like Faster Builds, Faster Runs, Debug. A toolchain object describes the basic information of the toolchain. The toolchain object has methods to describe all of the above. The object can be saved into a MATLAB file and shared across installations.

This example uses the toolchain definition file `intel_tc.m`.

```
tc = intel_tc
```

```
tc =
############################################################################
# Toolchain Name: Intel v14 | nmake makefile (64-bit Windows)
# Supported Toolchain Version: 14
# Toolchain Specification Format Version: 2021a
# Toolchain Specification Revision: 1.0
############################################################################

# Supported Host Platform = win64
# Supported Languages = C/C++

# ---------------
# Setup/Cleanup
# ---------------
MATLAB Setup: (none)
MATLAB Cleanup: (none)
Shell Setup:
    call %ICPP_COMPILER14%\bin\compilervars.bat intel64
Shell Cleanup: (none)

# ------------
# Attributes
# ------------
RequiresBatchFile        = true
```

```
RequiresCommandFile      = true
TransformPathsWithSpaces = true


# ---------------------------------------------------------------------
# Macros intrinsic to the toolchain or assumed to be defined elsewhere
# ---------------------------------------------------------------------
# ldebug
# conflags
# cflags

# --------
# MACROS
# --------
MW_EXTERNLIB_DIR    = $(MATLAB_ROOT)\extern\lib\win64\microsoft
MW_LIB_DIR          = $(MATLAB_ROOT)\lib\win64
CFLAGS_ADDITIONAL   = -D_CRT_SECURE_NO_WARNINGS
CPPFLAGS_ADDITIONAL = -EHs -D_CRT_SECURE_NO_WARNINGS
LIBS_TOOLCHAIN      = $(conlibs)
CVARSFLAG           =

###########################################################################
# Build Tool: Intel C Compiler
###########################################################################

Language              : 'C'
OptionsRegistry       : {'C Compiler', 'CFLAGS'}
InputFileExtensions   : {Source}
OutputFileExtensions  : {Object}
DerivedFileExtensions : {}
SupportedOutputs      : {*}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
CC = icl
CC_PATH =

# -----------
# Directives
# -----------
CompileFlag        =
Debug              = -Zi
ErrorPattern       =
FileNamePattern    =
FileSeparator      = \
Include            =
IncludeSearchPath  = -I
LineNumberPattern  =
OutputFlag         = -Fo
PreprocessFile     =
PreprocessorDefine = -D
WarningPattern     =

# ----------------
# File Extensions
# ----------------
```

```
Header = .h
Object = .obj
Source = .c

##########################################################################
# Build Tool: Intel C/C++ Linker
##########################################################################

Language              : 'C'
OptionsRegistry       : {'Linker', 'LDFLAGS', 'Shared Library Linker', 'SHAREDLIB_LDFLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {'Executable', 'Shared Library'}
DerivedFileExtensions : {}
SupportedOutputs      : {coder.make.enum.BuildOutput.EXECUTABLE, coder.make.enum.BuildOutput.SHAR
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
LD = xilink
LD_PATH =

# ------------
# Directives
# ------------
Debug                    =
FileSeparator            = \
Library                  = -L
LibrarySearchPath        = -I
LibrarySearchPathRuntime =
OutputFlag               = -out:

# -----------------
# File Extensions
# -----------------
Executable     = .exe
Shared Library = .dll

##########################################################################
# Build Tool: Intel C++ Compiler
##########################################################################

Language              : 'C++'
OptionsRegistry       : {'C++ Compiler', 'CPPFLAGS'}
InputFileExtensions   : {Source}
OutputFileExtensions  : {Object}
DerivedFileExtensions : {}
SupportedOutputs      : {*}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
CPP = icl
CPP_PATH =

# ------------
# Directives
```

```
# ------------
CompileFlag        =
Debug              = -Zi
ErrorPattern       =
FileNamePattern    =
FileSeparator      = \
Include            =
IncludeSearchPath  = -I
LineNumberPattern  =
OutputFlag         = -Fo
PreprocessFile     =
PreprocessorDefine = -D
WarningPattern     =

# -----------------
# File Extensions
# -----------------
Header = .hpp
Object = .obj
Source = .cpp

###########################################################################
# Build Tool: Intel C/C++ Linker
###########################################################################

Language               : 'C++'
OptionsRegistry        : {'C++ Linker', 'CPP_LDFLAGS', 'C++ Shared Library Linker', 'CPP_SHAREDLIB
InputFileExtensions    : {}
OutputFileExtensions   : {'Executable', 'Shared Library'}
DerivedFileExtensions  : {}
SupportedOutputs       : {coder.make.enum.BuildOutput.EXECUTABLE, coder.make.enum.BuildOutput.SHAR
CommandPattern         : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
CPP_LD = xilink
CPP_LD_PATH =

# ------------
# Directives
# ------------
Debug                    =
FileSeparator            = \
Library                  = -L
LibrarySearchPath        = -I
LibrarySearchPathRuntime =
OutputFlag               = -out:

# -----------------
# File Extensions
# -----------------
Executable     = .exe
Shared Library = .dll

###########################################################################
# Build Tool: Intel C/C++ Archiver
###########################################################################
```

```
Language              : 'C'
OptionsRegistry       : {'Archiver', 'ARFLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {Static Library}
DerivedFileExtensions : {}
SupportedOutputs      : {coder.make.enum.BuildOutput.STATIC_LIBRARY}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<| |>OUTPUT_FLAG<||>OUTPUT<|'

# ---------
# Command
# ---------
AR = xilib
AR_PATH =

# -----------
# Directives
# -----------
Debug            =
FileSeparator    = \
LibrarySearchPath =
OutputFlag       = -out:

# ----------------
# File Extensions
# ----------------
Static Library = .lib

##########################################################################
# Build Tool: Download
##########################################################################

Language              : ''
OptionsRegistry       : {'Download', 'DOWNLOAD_FLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {coder.make.enum.BuildOutput.EXECUTABLE}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<|'

# ---------
# Command
# ---------
DOWNLOAD =
DOWNLOAD_PATH =

# -----------
# Directives
# -----------
(none)

# ----------------
# File Extensions
# ----------------
(none)

##########################################################################
# Build Tool: Execute
```

```
##########################################################################

Language              : ''
OptionsRegistry       : {'Execute', 'EXECUTE_FLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {coder.make.enum.BuildOutput.EXECUTABLE}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<|'

# ---------
# Command
# ---------
EXECUTE = $(PRODUCT)
EXECUTE_PATH =

# -----------
# Directives
# -----------
(none)

# ----------------
# File Extensions
# ----------------
(none)

##########################################################################
# Build Tool: NMAKE Utility
##########################################################################

Language              : ''
OptionsRegistry       : {'Make Tool', 'MAKE_FLAGS'}
InputFileExtensions   : {}
OutputFileExtensions  : {}
DerivedFileExtensions : {}
SupportedOutputs      : {*}
CommandPattern        : '|>TOOL<| |>TOOL_OPTIONS<|'

# ---------
# Command
# ---------
MAKE = nmake
MAKE_PATH =

# -----------
# Directives
# -----------
Comment                = #
DeleteCommand          = @del
DisplayCommand         = @echo
FileSeparator          = \
ImpliedFirstDependency = $<
ImpliedTarget          = $@
IncludeFile            = !include
LineContinuation       = \
MoveCommand            = @ren
ReferencePattern       = \$\($1\)
RunScriptCommand       = @cmd /C
```

```
# -----------------
# File Extensions
# -----------------
Makefile = .mk

##########################################################################
# Build Configuration : Faster Runs
# Description         : Minimize run time
##########################################################################

ARFLAGS              = /nologo
CFLAGS               = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /O2
CPPFLAGS             = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /O2
CPP_LDFLAGS          = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN)
CPP_SHAREDLIB_LDFLAGS =
DOWNLOAD_FLAGS       =
EXECUTE_FLAGS        =
LDFLAGS              = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN)
MEX_CPPFLAGS         =
MEX_CPPLDFLAGS       =
MEX_CFLAGS           =
MEX_LDFLAGS          =
MAKE_FLAGS           = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS    = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE)

##########################################################################
# Build Configuration : Faster Builds
# Description         : Minimize compilation and linking time
##########################################################################

ARFLAGS              = /nologo
CFLAGS               = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /Od
CPPFLAGS             = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /Od
CPP_LDFLAGS          = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN)
CPP_SHAREDLIB_LDFLAGS =
DOWNLOAD_FLAGS       =
EXECUTE_FLAGS        =
LDFLAGS              = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN)
MEX_CPPFLAGS         =
MEX_CPPLDFLAGS       =
MEX_CFLAGS           =
MEX_LDFLAGS          =
MAKE_FLAGS           = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS    = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE)

##########################################################################
# Build Configuration : Debug
# Description         : Build with debug information
##########################################################################

ARFLAGS              = /nologo $(ARDEBUG)
CFLAGS               = $(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) /c /Od $(CDEBUG)
CPPFLAGS             = $(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL) /c /Od $(CPPDEBUG)
CPP_LDFLAGS          = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) $(CPPLDDEBUG)
CPP_SHAREDLIB_LDFLAGS =
DOWNLOAD_FLAGS       =
EXECUTE_FLAGS        =
```

```
LDFLAGS              = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) $(LDDEBUG)
MEX_CPPFLAGS         =
MEX_CPPLDFLAGS       =
MEX_CFLAGS           =
MEX_LDFLAGS          =
MAKE_FLAGS           = -f $(MAKEFILE)
SHAREDLIB_LDFLAGS    = $(ldebug) $(conflags) $(LIBS_TOOLCHAIN) -dll -def:$(DEF_FILE) $(LDDEBUG)
```

```
save intel_tc tc
```

Registering a Toolchain

Toolchains are registered through `RTW.TargetRegistry`. To register the toolchain, you can also use `rtwTargetInfo` which will be loaded by the system automatically.

```
copyfile myRtwTargetInfoCustom.txt rtwTargetInfo.m
type rtwTargetInfo
```

```
function rtwTargetInfo(tr)
%RTWTARGETINFO Registration file for custom toolchains.

% Copyright 2012-2016 The MathWorks, Inc.

tr.registerTargetInfo(@loc_createToolchain);

end

% -------------------------------------------------------------------------
% Create the ToolchainInfoRegistry entries
% -------------------------------------------------------------------------
function config = loc_createToolchain

config(1)                        = coder.make.ToolchainInfoRegistry;
config(1).Name                   = 'Intel v14 | nmake makefile (64-bit Windows)';
config(1).FileName               = fullfile(fileparts(mfilename('fullpath')), 'intel_tc.mat');
config(1).TargetHWDeviceType     = {'*'};
config(1).Platform               =  {computer('arch')};

end
```

Now, you can reset the `TargetRegistry` to pick up the new `rtwTargetInfo`.

```
RTW.TargetRegistry.getInstance('reset');
```

Choosing the Toolchain

You can now create the config object that is configured to create an executable using the new toolchain.

```
cfg = coder.config('exe');
cfg.CustomSource = 'coderrand_main.c';
cfg.CustomInclude = pwd;
cfg.Toolchain = 'Intel v14';
```

If you do not have the Intel compilers installed, you can use the following command to generate the code and makefile only.

```
cfg.GenCodeOnly = true;
```

Run the `codegen` to generate the code and makefile that uses the new toolchain.

```
codegen -config cfg coderrand
```

```
Code generation successful.
```

Once the `codegen` is finished, and you had Intel compilers installed, you can use `system('coderrand.exe')` to run the executable.

Cleanup

You can reset the `TargetRegistry` to remove the toolchain that you registered above.

```
delete ./rtwTargetInfo.m
RTW.TargetRegistry.getInstance('reset');
```

Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type

This example shows how to generate a standalone C++ library from a MATLAB® function that performs Sobel edge detection of images by using half-precision floating point numbers. The Sobel edge algorithm accepts an image that is represented as a matrix and returns an image emphasizing the high spatial frequency regions that correspond to its edges. This example also shows how to test the genenrated code by using a MEX function.

Sobel Edge Detection Algorithm

In the Sobel edge detection algorithm, a 2-D spatial gradient operation is performed on a grayscale image. This operation emphasizes the high spatial frequency regions that correspond to the edges in the image.

```
type sobelEdgeDetectionAlg
```

```
function edgeImg = sobelEdgeDetectionAlg(img,thresh)  %#codegen
%sobelEdgeDetection Example MATLAB function for edge detection.
% Copyright 2018 The MathWorks, Inc.

kern = half([1 2 1; 0 0 0; -1 -2 -1]);

% Finding horizontal and vertical gradients.
h = conv2(img(:,:,2),kern,'same');
v = conv2(img(:,:,2),kern','same');

% Finding magnitude of the gradients.
e = sqrt(h.*h + v.*v);

% Threshold the edges
edgeImg = uint8((e > thresh) * 240);

end
```

The Sobel edge algorithm computes the horizontal gradient `h` and the vertical gradient `v` of the input image by using two orthogonal filter kernels `maskX` and `maskY`. After the filtering operation, the algorithm computes the gradient magnitude and applies a threhold to find the regions of the image that correspond to the edges.

Read Images and Pack Data Into RGBA Packed Column Major Order

Use the `imread` function to read the images. `imread` represents the RGB channels of an images with integers, one for each pixel. The integers range from 0 to 255. Simply casting inputs to half type might result in overflow during convolutions. To avoid this issue, scale the images to values between 0 and 1.

```
im = imread('peppers.png');
figure();
image(im);
imPacked = half(im)/255;
thresh = half(100)/255;
```

Generate MEX

Generate a C++ MEX function for the `sobelEdgeDetectionAlg` function by using the `codegen` command.

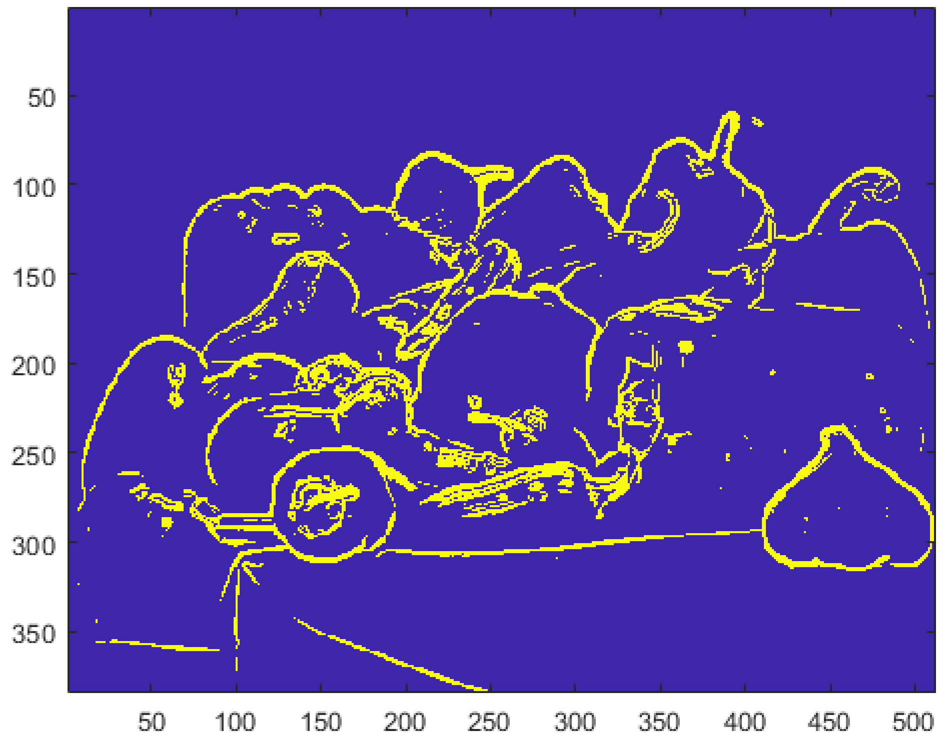
```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.GenerateReport = true;
codegen -config cfg -args {imPacked,thresh} sobelEdgeDetectionAlg
```

```
Code generation successful: To view the report, open('codegen\mex\sobelEdgeDetectionAlg\html\repo
```

Run Generated MEX and Display Detected Edge

Before generating C++ code, you must first test the MEX function inside MATLAB environment to make sure 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 `sobelEdgeDetectionAlg_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

```
out_disp = sobelEdgeDetectionAlg_mex(imPacked,thresh);
figure();
imagesc(out_disp);
```



Generate Static C++ Library

Use the `codegen` command to produces a C++ static library. By default, the generated library is located in the folder `codegen/lib/sobelEdgeDetectionAlg/`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenerateReport = true;
codegen -config cfg -args {imPacked,thresh} sobelEdgeDetectionAlg;
```

```
Code generation successful: To view the report, open('codegen\lib\sobelEdgeDetectionAlg\html\repo
```

Inspect the Generated Function

```
type codegen/lib/sobelEdgeDetectionAlg/sobelEdgeDetectionAlg.cpp
```

```
//
// File: sobelEdgeDetectionAlg.cpp
//
// MATLAB Coder version            : 5.2
// C/C++ source code generated on  : 23-Feb-2021 17:25:57
//

// Include Files
#include "sobelEdgeDetectionAlg.h"
#include "conv2MovingWindowSameCM.h"
```

```
#include "rtwhalf.h"
#include "sobelEdgeDetectionAlg_data.h"
#include "sobelEdgeDetectionAlg_initialize.h"
#include <cmath>

// Function Definitions
//
// sobelEdgeDetection Example MATLAB function for edge detection.
//  Copyright 2018 The MathWorks, Inc.
//
// Arguments    : const real16_T img[589824]
//                real16_T thresh
//                unsigned char edgeImg[196608]
// Return Type  : void
//
void sobelEdgeDetectionAlg(const real16_T img[589824], real16_T thresh,
                           unsigned char edgeImg[196608])
{
  static const real16_T hv[9]{real16_T(1.0F), real16_T(0.0F), real16_T(-1.0F),
                              real16_T(2.0F), real16_T(0.0F), real16_T(-2.0F),
                              real16_T(1.0F), real16_T(0.0F), real16_T(-1.0F)};
  static const real16_T hv1[9]{
      real16_T(1.0F),  real16_T(2.0F),  real16_T(1.0F),
      real16_T(0.0F),  real16_T(0.0F),  real16_T(0.0F),
      real16_T(-1.0F), real16_T(-2.0F), real16_T(-1.0F)};
  static real16_T h[196608];
  static real16_T v[196608];
  if (!isInitialized_sobelEdgeDetectionAlg) {
    sobelEdgeDetectionAlg_initialize();
  }
  //  Finding horizontal and vertical gradients.
  coder::conv2MovingWindowSameCM(*(real16_T(*)[196608]) & img[196608], hv, h);
  coder::conv2MovingWindowSameCM(*(real16_T(*)[196608]) & img[196608], hv1, v);
  //  Finding magnitude of the gradients.
  //  Threshold the edges
  for (int k{0}; k < 196608; k++) {
    real16_T b_h;
    real16_T h1;
    b_h = h[k];
    h1 = v[k];
    b_h = static_cast<real16_T>(
        std::sqrt(static_cast<float>(b_h * b_h + h1 * h1)));
    h[k] = b_h;
    edgeImg[k] = static_cast<unsigned char>((b_h > thresh) * 240U);
  }
}

//
// File trailer for sobelEdgeDetectionAlg.cpp
//
// [EOF]
//
```

See Also

`codegen` | `coder.config` | `half`

More About

- “Floating-Point Numbers” (Fixed-Point Designer)

Build Process Support for Folder Names with Spaces or Special Characters

Folder Names with Spaces

On a Windows system, the code generator maps a drive corresponding to the MATLAB installation folder for either of these conditions:

- The `matlabroot` folder is a UNC location.
- The path the `matlabroot` folder contains spaces, and the system has no alternative name support.

These folder paths can contain spaces:

- The path to your MATLAB installation folder (`matlabroot`). For example, `C:\Program Files\MATLAB\R2015b`
- The path to the current working folder where you start the build (`pwd`). For example, `C:\Users\username\Documents\My Work`.
- The path to the installation folder for a compiler that the build process uses.

If your work environment includes one or more of the preceding scenarios, use the following support mechanisms for the build process:

- If you are using the toolchain approach to build generated code, the system support for spaces in folder names influences toolchain operation:

  - For Linux systems and Windows systems with 8.3 name creation enabled, the toolchain manages spaces in folder names by using alternative names from the operating system. The toolchain uses the `TransformPathsWithSpaces` attribute to manage these names.

    ```
    addAttribute(toolchainObject, 'TransformPathsWithSpaces', true);
    ```

    The security permissions of drives and folders can determine whether the toolchain transforms the path. For example, if the path contains a folder with a security configuration that forbids 8.3 path transformations, the toolchain does not transform the path and the build process produces a warning.
  - For Windows systems with 8.3 name creation disabled, the toolchain manages spaces in folder names by mapping a network drive using a batch file (.bat). This operation requires adding the `RequiresBatchFile` attribute to the toolchain definition.

    ```
    addAttribute(toolchainObject, 'RequiresBatchFile', true);
    ```

  When developing a toolchain for a Windows system, set both attributes. For more information about the toolchain attributes, see `addAttribute`.

When there is an issue with support for creation of alternate names (short names), build errors can occur on Windows. If a build generates an error message similar to the following message, see “Troubleshooting Errors When Folder Names Have Spaces” on page 27-181.

```
NMAKE : fatal error U1073: don't know how to make ' ...
```

When using operating system commands, such as `system` or `dos`, enclose paths that specify executable files or command parameters in double quotes (`" "`). For example:

```
system('dir "D:\Applications\Common Files"')
```

This table provides a summary of build folder support and limitations for Windows.

<table>
<tr><th>Build Process Folders</th><th>Approach for Paths with UNC or Spaces</th><th>Support for Windows</th></tr>
<tr><td><code>matlabroot</code> folder

The <code>matlabroot</code> value is derived from the MATLAB installation location.</td><td>During a build, a UNC location such as:

<code>\\networkdrive\matlab\R20xxb</code>

could be remapped as:

<code>T:\</code>

During a build on a Windows system with short file name (8.3) support (default for Windows using NTFS), the build process uses the Windows API <code>getShortPathName()</code> for the folder location.

During a build on a Windows system without short file name (8.3) support (systems using ReFS or using NTFS with 8.3 support disabled), a location with spaces in the path such as:

<code>C:\Program Files\MATLAB\R20xxb</code>

could be remapped as:

<code>T:\R20xxb</code></td><td>Build process folder support available independent of file system (NTFS or ReFS) or file system configuration for short file name support.

Limitations:

On systems that require drive mapping for the installation location, the build process requires that a drive letter is available for mapping.

On systems without short file name (8.3) support (using ReFS or using NTFS with 8.3 support disabled), the final folder in the installation location cannot contain spaces. For example, a final folder name:

<code>C:\Program Files\MATLAB\R20xxb sp1</code>

is not supported.</td></tr>
<tr><td rowspan="2">Code generation folder

Custom code source file locations—among others, these locations include folders specified by a Code Replacement Library</td><td>For UNC locations, build process temporarily maps a drive by using the shell commands <code>pushd</code> and <code>popd</code>.</td><td>Build process folder support is available independent of file system (NTFS or ReFS) or file system configuration for short path name support.</td></tr>
<tr><td>For paths with spaces, build process uses the Windows short path name (8.3) by using the Windows API:

<code>getShortPathName()</code></td><td>Build process folder support depends on NTFS file system and requires Windows default support. Registry sets value of 2 or 0 for:

<code>NtfsDisable8dot3NameCreation</code>

Limitations: Build process does not support spaces in the path to these folders for:

• NTFS file system with short path name support disabled
• ReFS file system (this file system does not support short path names)</td></tr>
</table>

Folder Names with Special Characters

If a build-related folder path contains a Japanese (multibyte) character where the final byte is equal to the 5C hexadecimal character, the build process might produce an error. The make and compiler tools might incorrectly interpret the final byte as the '\' (backslash) character.

Troubleshooting Errors When Folder Names Have Spaces

On Windows, when there is an issue with support for creation of short file names, build process errors can occur. When this issue affects a build, you see an error message similar to:

```
NMAKE : fatal error U1073: don't know how to make 'C:\Work\My'
```

This message can occur if a space in the folder name (C:\Work\My Models) prevents the build process from finding a file to build. For descriptions of the build-related folders that are sensitive to a space in the folder name or path, see "Folder Names with Spaces" on page 27-179.

To avoid issues from folder names with spaces when Windows short file name support for file names is disabled, do not use paths with spaces. For example, install third-party software to paths without spaces. Do not use paths with spaces for folders containing your models, source files, or libraries.

An issue can occur with builds that use folder names with spaces, because it is possible to disable Windows alternate name support. The build process uses this alternate name support on Windows systems. There are many terms for this file, folder, and path alternate name support:

- 8.3 name
- DOS path
- short file name (SFN, ShortFileName)
- long name alias
- Windows path alias

Verify the type of file system that the drive uses. In Windows Explorer, right-click the drive icon and select properties.

- If the file system is ReFS (Resilient File System), it is an issue. The ReFS does not provide short file name support. Except for the MATLAB installation folder, the build process does not support folder names with spaces for the ReFS file system. If your work environment requires short file name support for the build folder or for additional external code folders, do not use ReFS.
- If the file system is NTFS (New Technology File System), it is possible that the build error is related to a registry setting incompatibility. Continue with troubleshooting steps.

The error could stem from an issue with short file name support on a system using NTFS. Check the Windows registry setting that enables the creation of short names for files, folders, and paths.

1. Open the Windows command prompt, running as administrator. For example, from the Windows Start menu, type cmd, right-click the cmd.exe icon, and select Run as administrator.
2. Change to the windows\system32 folder and query the NtfsDisable8dot3NameCreation status by typing:

   ```
   > fsutil 8dot3name query
   ```
3. If the registry state of NtfsDisable8dot3NameCreation is not 2, the default (Volume level setting), change the value to 2 by typing:

```
> fsutil 8dot3name set 2
```

For more information about enabling creation of short names. See https://technet.microsoft.com/en-us/library/ff621566.aspx.

Changing the registry setting enables creation of short names only for files and folders that are created after the change.

4. To create short names for files created while short name creation was disabled, at the Windows command line, use the `fsutil` utility.

   To set the short name, the syntax is:

   ```
   > fsutil file setshortname <FileName> <ShortName>
   ```

   For example, to create the short name `PROGRA~1` for the long name `C:\Program Files`, type:

   ```
   > fsutil file setshortname "C:\Program Files" PROGRA~1
   ```

   The `C:\Program Files` folder name is in quotations because it has spaces.
5. To verify that the short name was created, use the `dir` command with `/x` option to show short names.

   ```
   > dir C:\ /x
   ```

See Also

`addAttribute`

External Websites

- MATLAB Answers: “Why is the build process failing ...?”
- https://technet.microsoft.com/en-us/library/cc788058.aspx
- https://technet.microsoft.com/en-us/library/cc788058.aspx

Verify Generated C/C++ Code

- “Tracing Generated C/C++ Code to MATLAB Source Code” on page 28-2
- “Code Generation Reports” on page 28-7
- “Access Code Generation Report Information Programmatically” on page 28-13
- “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20
- “Example: Generate Standalone C Code That Detects and Reports Run-Time Errors” on page 28-24
- “Testing Code Generated from MATLAB Code” on page 28-26
- “Unit Test Generated Code with MATLAB Coder” on page 28-27
- “Unit Test External C Code with MATLAB Coder” on page 28-33
- “Calculate Number of Lines of Code by Using Report Information Object” on page 28-43

Tracing Generated C/C++ Code to MATLAB Source Code

In this section...

"Generate Traceability Tags" on page 28-2
"Format of Traceability Tags" on page 28-2
"Location of Comments in Generated Code" on page 28-2
"Traceability Tag Limitations" on page 28-6

Tracing the generated C/C++ code to the original MATLAB source code helps you to:

- Understand how the generated code implements your algorithm.
- Evaluate the quality of the generated code.

You can trace by using one of these methods:

- Configure MATLAB Coder to generate code that includes the MATLAB source code as comments. In the comments, a traceability tag immediately precedes each line of source code. The traceability tag provides details about the location of the source code. If you have Embedded Coder, in the code generation report, the traceability tags link to the corresponding MATLAB source code.
- With Embedded Coder, produce a code generation report that includes interactive traceability. Interactive tracing in the report helps you to visualize the mapping between the MATLAB source code and the generated C/C++ code. See "Interactively Trace Between MATLAB Code and Generated C/C++ Code" (Embedded Coder).

Generate Traceability Tags

To produce traceability tags in the generated code, enable generation of MATLAB source code as comments.

- In the MATLAB Coder app, set **MATLAB source code as comments** to `Yes`.
- In a code generation configuration object, set `MATLABSourceComments` to `true`.

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, this 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); */
```

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 Tag Limitations

- You cannot include MATLAB source code as comments for:
  - MathWorks toolbox functions
  - P-code
- The appearance or location of comments can vary:
  - 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.

See Also

More About

- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Include Comments in Generated C/C++ Code” (Embedded Coder)
- “Code Generation Reports” on page 28-7

Code Generation Reports

In this section...

"Report Generation" on page 28-7
"Report Location" on page 28-8
"Errors and Warnings" on page 28-8
"Files and Functions" on page 28-8
"MATLAB Source" on page 28-9
"MATLAB Variables" on page 28-10
"Tracing Code" on page 28-11
"Code Insights" on page 28-11
"Additional Reports" on page 28-12
"Report Limitations" on page 28-12

MATLAB Coder produces a code generation report that helps you to:

- Debug code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated C/C++ code.
- Trace between MATLAB source code and generated C/C++ code.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.
- Identify potential issues in the generated code.
- Access additional reports available with Embedded Coder.

Report Generation

When you enable report generation or when an error occurs, the code generator produces a code generation report. To control production and opening of a code generation report, use app settings, `codegen` options, or configuration object properties.

In the MATLAB Coder app:

- To generate a report, set **Always create a report** to `Yes`.
- If you want the app to open the report for you, set **Automatically launch a report if one is generated** to `Yes`.

At the command line, use `codegen` options:

- To generate a report, use the `-report` option.
- To generate and open a report, use the `-launchreport` option.

Alternatively, use configuration object properties:

- To generate a report, set `GenerateReport` to `true`.
- If you want `codegen` to open the report for you, set `LaunchReport` to `true`.

Report Location

The code generation report is named `report.mldatx`. It is located in the `html` subfolder of the code generation output folder. If you have MATLAB R2018a or later, you can open the `report.mldatx` file by double-clicking it.

Errors and Warnings

View code generation error, warning, and information messages on the **All Messages** tab. To highlight the source code for an error or warning, click the message. It is a best practice to address the first message because subsequent errors and warnings can be related to the first message.

View compilation and linking errors and warnings on the **Build Logs** tab. The code generator detects compilation warnings only for MEX output or if you use a supported compiler for other types of output. See .`https://www.mathworks.com/support/compilers/current_release/`.

Files and Functions

The report lists MATLAB source functions and generated files. In the **MATLAB Source** pane, the **Function List** view organizes functions according to the containing file. To visualize functions according to the call structure, use the **Call Tree** view.

To view a function in the code pane of the report, click the function in the list. Clicking a function opens the file that contains the function. To edit the selected file in the MATLAB Editor, click **Edit in MATLAB** or click a line number in the code pane.

If you have Embedded Coder and generate the report with traceability enabled, to view the source code and generated code next to each other in the code pane, click **Trace Code**. You can interactively trace between the source code and the generated code. See "Interactively Trace Between MATLAB Code and Generated C/C++ Code" (Embedded Coder).

If you want to move the generated files for standalone code (library or executable) to another development environment, you can put them into a zip file by clicking **Package Code**.

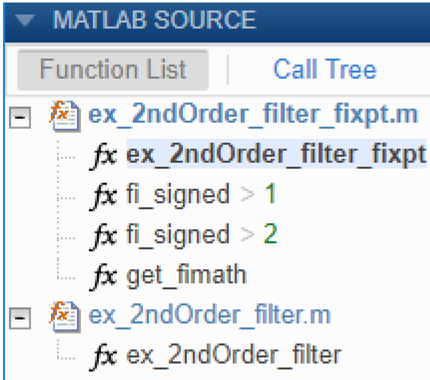
Specialized Functions or Classes

When a function is called with different types of inputs or a class uses different types for its properties, the code generator produces specializations. In the **MATLAB Source** pane, numbered functions (or classes) indicate specializations. For example:

fx fcn > 1
fx fcn > 2

Functions List After Fixed-Point Conversion

If you convert floating-point MATLAB code to fixed-point MATLAB code, and then generate fixed-point C/C++ code, the **MATLAB Source** pane lists the original MATLAB functions and the fixed-point MATLAB functions. For example:



MATLAB Source

To view a MATLAB function in the code pane, click the name of the function in the **MATLAB Source** pane. In the code pane, when you pause on a variable or expression, a tooltip displays information about its size, type, and complexity. Additionally, syntax highlighting helps you to identify MATLAB syntax elements and certain code generation attributes, such as whether a function is extrinsic or whether an argument is constant.

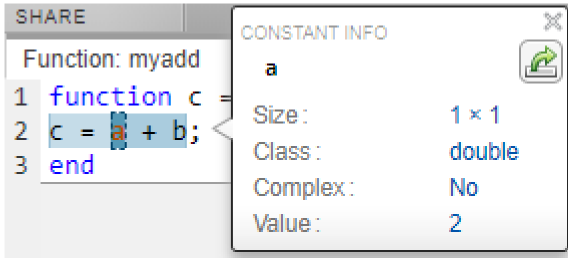
Extrinsic Functions

The report identifies an extrinsic function with purple text. The tooltip indicates that the function is extrinsic.

```
Function: callMyExtrinsic
1 function z = callMyExtrinsic(a,b)
2 %#codegen
3 coder.extrinsic('myExtr
4 z = 0;
5 z = myExtrinsic(a,b);
6 disp(z);
7 end
8

EXPRESSION INFO
myExtrinsic(a,b)
Size :    1 × 1
Class :   mxArray
myExtrinsic is an extrinsic function.
```

Constant Arguments

Orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The tooltip includes the constant value.



Knowing the value of a constant argument helps you to understand the generated function signatures. It also helps you to see when code generation creates function specializations for different constant argument values.

To export the value to a variable in the workspace, click the Export icon .

MATLAB Variables

The **Variables** tab provides information about the variables for the selected MATLAB function. To select a function, click the function in the **MATLAB Source** pane.

The variables table shows:

- Class, size, and complexity
- Properties of fixed-point types
- Whether an array is sparse
- Array layout

This information helps you to debug errors, such as type mismatch errors, and to understand how the code generator propagates types and represents data in the generated code.

Visual Indicators on the Variables Tab

This table describes the symbols, badges, and other indicators in the variables table.

Column in the Variables Table	Indicator	Description
Name	expander	Variable has elements or properties that you can see by clicking the expander.
Name	`{:}`	Heterogeneous cell array (all elements have the same properties).
Name	`{n}`	`n`th element of a heterogeneous cell array.
Class	`v > n`	`v` is reused with a different class, size, and complexity. The number `n` identifies a reuse with a unique set of properties. When you pause on a renamed variable, the report highlights only the instances of this variable that share the class, size, and complexity. See “Reuse the Same Variable with Different Properties” on page 4-9.
Size	`:n`	Variable-size array with an upper bound of `n`.

Column in the Variables Table	Indicator	Description
Size	`:?`	Variable-size array with no upper bound.
Size	italics	Variable-size array whose dimensions do not change size during execution.
Class	`sparse` prefix	Sparse array.
Class	`complex` prefix	Complex number.
Class		Fixed-point type. To see the fixed-point properties, click the badge.

Array Layout Indicators on the Variables Tab

This table describes the badges that indicate array layout in the variables table.

Badge	Description
	Row-major array layout.
	Column-major array layout.
	A mixture of row-major and column-major layouts.

See “Row-Major and Column-Major Array Layouts” on page 37-2.

Tracing Code

You can trace between MATLAB source code and generated C/C++ code by using one of these methods:

- Interactively visualize the mapping between the MATLAB code and the generated code. To access interactive tracing, in the report, click **Trace Code**.

  The **Trace Code** button is enabled only if you have Embedded Coder and you enabled code traceability when you generated code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).
- Include source code as comments in the generated C/C++ code.

  In a comment, the code generator produces a tag that helps you find the corresponding MATLAB source code. If you have Embedded Coder, the tag is a link to the source code. See “Tracing Generated C/C++ Code to MATLAB Source Code” on page 28-2.

Code Insights

The code generator can detect and report issues that can potentially occur in the generated code. View the messages on the **Code Insights** tab. The issues include:

- Potential differences between the behavior of the generated code and the behavior of the MATLAB code. The report includes potential differences messages only if you enabled potential differences reporting. See “Potential Differences Reporting” on page 2-18.
- Potential data type issues in the generated code, such as single-precision and double-precision operations.

  The report includes potential data type issues only if you have Embedded Coder and you enabled potential data type issues reporting. If you have Fixed-Point Designer, the report also identifies expensive fixed-point operations. See “Highlight Potential Data Type Issues in a Report” (Embedded Coder).
- Potential row-major issues. See “Code Design for Row-Major Array Layout” on page 5-21.
- Automatic parallelization issues. See “Automatically Parallelize for Loops in Generated Code” on page 34-69.

Additional Reports

The **Summary** tab can have links to these additional reports:

- Static code metrics report (requires Embedded Coder). See “Generating a Static Code Metrics Report for Code Generated from MATLAB Code” (Embedded Coder).
- Code replacements report (requires Embedded Coder). See “Verify Code Replacement Library” (Embedded Coder).
- Fixed-point conversion report (requires Fixed-Point Designer). See “Convert MATLAB Code to Fixed-Point C Code” on page 21-5.

Report Limitations

- The entry-point summary shows the individual elements of `varargin` and `varargout`, but the variables table does not show them.
- The report does not show full information for unrolled loops. It displays data types of one arbitrary iteration.
- The report does not show information about dead code.

See Also

More About

- “Generating a Static Code Metrics Report for Code Generated from MATLAB Code” (Embedded Coder)
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Tracing Generated C/C++ Code to MATLAB Source Code” on page 28-2
- “Convert MATLAB Code to Fixed-Point C Code” on page 21-5
- “Row-Major and Column-Major Array Layouts” on page 37-2
- “Basic HDL Code Generation and FPGA Synthesis from MATLAB” (HDL Coder)
- “Generate HDL Code from MATLAB Code Using the Command Line Interface” (HDL Coder)

Access Code Generation Report Information Programmatically

You can export information about code generation to a variable in your base MATLAB workspace. This variable contains a `coder.ReportInfo` object whose properties contain this information:

- A code generation summary that includes information about code generation success, date and time, path of the output file, processor, version of MATLAB Coder, toolbox licenses checked out during code generation, toolchain, and build configuration.
- The code generation configuration object.
- The text, path, and extension of the input files.
- The text, path, and extension of the generated files.
- For all MATLAB functions and methods involved in code generation: name, specialization, file, start index, and end index.
- Code generation error, warning, and information messages.
- Code insights indicating potential issues with the generated code.
- Build logs produced during code generation.

See coder.ReportInfo Properties.

You can use the report information object to programmatically access this information about code generation. For example, you can display the code generation messages at the MATLAB command line. To perform this action, in your build script, access the property that contains these messages.

Create Report Information Object

Suppose that you want to export the code generation report information to the variable `info` in your base MATLAB workspace. Do one of the following:

- In the MATLAB Coder app, on the **Debugging** tab, set **Export report information to variable** to the variable name `info`.
- At the command line, use the `codegen` command with the `-reportinfo` option. Specify the variable name after the `-reportinfo` option.

  ```
  codegen myFunction -reportinfo info
  ```
- At the command line, set the code configuration object property `ReportInfoVarName` to the character vector `'info'`.
- Generate and open the code generation report. Click **Export Report Information**. In the dialog box, specify the variable name `info`.

Example: Create Report Information Object for Successful Code Generation

Create a report information object for a successful code generation process. Inspect the properties of this object.

1. Define the MATLAB function `foo`:

   ```
   function b = foo(a)
   c = svd(a,0);
   ```

```
b = sum(c);
end
```

Generate a MEX function for `foo`. Specify the input `a` as a variable-size matrix whose first dimension has an upper bound of `3` and second dimension has an upper bound of `5`. Export the code generation report information to the variable `info` in your base MATLAB workspace.

```
codegen -config:mex foo -args {coder.typeof(ones(1,1),[3 5],[1 1])} -reportinfo info
```

The code generator produces the MEX function `foo_mex`. The code generator also creates the report information object `info` in the base MATLAB workspace.

2. Inspect the structure of the report information object. The object has eight properties that contain information about code generation.

```
  ReportInfo with properties:

           Summary: [1×1 coder.Summary]
            Config: [1×1 coder.MexCodeConfig]
        InputFiles: [1×1 coder.CodeFile]
    GeneratedFiles: [21×1 coder.CodeFile]
         Functions: [1×1 coder.Function]
          Messages: [0×1 coder.Message]
      CodeInsights: [1×1 coder.Message]
         BuildLogs: [1×1 coder.BuildLog]
```

3. Inspect each property of `info` separately.

   - `info.Summary` is a `coder.Summary` object whose properties contain information about code generation success, code generation date and time, path of the output file, processor, toolbox licenses checked out during code generation, and version of MATLAB Coder.

     ```
       Summary with properties:

                 Success: true
                    Date: '08-May-2020 09:15:07'
              OutputFile: 'C:\coder\R2020b\License discovery\foo_mex.mexw64'
               Processor: 'Generic->MATLAB Host Computer'
                 Version: 'MATLAB Coder 5.1 (R2020b)'
         ToolboxLicenses: [1×0 string]
     ```

     If you generate standalone code, `info.Summary` also contains information about toolchain and build configuration.

   - `info.Config` is the code configuration object. In this example, because you generated a MEX function for `foo`, it is a `coder.MexCodeConfig` object.
   - `info.InputFiles` is an array of `coder.CodeFile` objects. Each element of the array contains the text, path, and extension of a code generation input file. In this example, the array has just one element because there is only one input file `foo.m`.

     ```
       CodeFile with properties:

              Text: 'function b = foo(a)←↵b = svd(a,0);←↵end←↵'
              Path: 'C:\coder\R2019a\Report Info Object\foo.m'
         Extension: '.m'
     ```

   - `info.GeneratedFiles` is an array of `coder.CodeFile` objects. Each element of the array contains the text, path, and extension of a generated file. In this example, it is a `21`-by-`1` array because there are `25` generated files.

```
  21×1 CodeFile array with properties:

    Text
    Path
    Extension
```

- `info.Functions` is an array of `coder.Function` objects. Each element of the array contains the following information about a MATLAB function or method:

  - Name and specialization.
  - The `coder.CodeFile` object for the input file that contains the function or method. This object is also contained in `info.InputFiles`.
  - The start and end index of the function or the method in the text of the file.

  In this example, `info.Functions` has one element because there is only one MATLAB function in the input file `foo.m`.

```
  Function with properties:

              Name: 'foo'
    Specialization: 0
              File: [1×1 coder.CodeFile]
        StartIndex: 1
          EndIndex: 52
```

- `info.Messages` is an array of `coder.Message` objects that contain the code generation error, warning, and information messages. In this example, there are no such messages. So, this property is an empty array.

```
  0×1 Message array with properties:

    Identifier
    Type
    Text
    File
    StartIndex
    EndIndex
```

- `info.CodeInsights` is an array of `coder.Message` objects that contain the code insights. These insights are messages about potential issues in the generated code such as potential differences from MATLAB code and potential row-major array layout issues. These messages also appear in the code generation report **Code Insights** tab. Each element of the array contains the following information about one code insight:

  - The identifier and the type of the message.
  - The text of the message.
  - The category and the subcategory that the message belongs to.
  - The `coder.File` or `coder.CodeFile` object for the input file that produced the message.
  - The start and end index of the part of the file text that produced the message.

  In this example, there is one code insight.

```
  Message with properties:

    Identifier: 'Coder:potentialDifferences:autoDimIncompatibility'
          Type: 'Info'
```

```
        Text: 'In the generated code, the dimension to operate along is selected automat
    Category: 'PotentialDifferencesFromMATLAB'
        File: [1×1 coder.CodeFile]
  StartIndex: 41
    EndIndex: 46
```

To index into the text of the file, use the `StartIndex` and `EndIndex` properties.

```
info.CodeInsights.File.Text(41:46)
```

This command displays the part of the file text that produced the code insight.

```
'sum(c)'
```

- `info.BuildLogs` is an array of `coder.BuildLog` objects that contain the build logs produced during code generation. The build logs contain compilation and linking errors and warnings. The same build logs also appear in the code generation report **Build Logs** tab. Each element of the array contains the type and the text of one build log. In this example, there is one build log of type `'Target'`.

Example: Create Report Information Object for Successful Code Generation That Checks Out Toolbox Licenses

Create a report information object for a code generation process that checks out toolbox licenses. Inspect the properties of this object.

1. Define the MATLAB function `bar` that calls the functions `iqr` and `haart`.

   ```
   function [u,v,w] = bar(x) %#codegen
   u = iqr(x);
   [v,w] = haart(x);
   end
   ```

   Generate C source code for `bar`. Specify the type of the input argument as a `1`-by-`100` row vector of doubles. Export the code generation report information to the variable `info` in your base MATLAB workspace.

   ```
   codegen -c bar -args {zeros(1,100)} -reportinfo info
   ```

2. Code generation succeeds. Inspect the `info.Summary.ToolboxLicenses` property.

   ```
     1×2 string array

       "statistics_toolbox"    "wavelet_toolbox"
   ```

   This property shows that the Statistics and Machine Learning Toolbox™ and Wavelet Toolbox™ licenses were checked out during code generation.

   Note If you generate MEX code, these licenses are checked out again when you load the MEX function.

   If you generate static library or dynamically linked library, the toolbox licenses are checked out only during code generation. The code generator does not write license checkouts into generated standalone code.

Example: Create Report Information Object for Failed Code Generation

Create a report information object for a code generation process that fails. Inspect the properties of this object.

1. Define the MATLAB function `foo`:

```
function b = foo(a)
b = svd(a,0);
end
```

   Generate a MEX function for `foo`. Specify the input `a` as a string scalar. Export the code generation report information to the variable `info` in your base MATLAB workspace.

```
codegen -config:mex foo -args {"A string scalar"} -reportinfo info
```

   Code generation fails because a string scalar is not a valid input for the MATLAB function `svd`. The code generator creates the report information object `info` in the base MATLAB workspace.

2. Inspect the `info.Summary` and `info.Messages` properties.

   - `info.Summary` indicates that code generation has failed.

```
Summary with properties:

           Success: false
              Date: '08-May-2020 10:20:35'
        OutputFile: 'C:\coder\R2020b\License discovery\codegen\mex\foo'
         Processor: 'Generic->MATLAB Host Computer'
           Version: 'MATLAB Coder 5.1 (R2020b)'
   ToolboxLicenses: [1×0 string]
```

   - `info.Messages` is an array of `coder.Message` objects that contain the code generation error, warning, and information messages. Each element of the array contains the following information about one message:

     - The identifier and the type of the message.
     - The text of the message.
     - The `coder.CodeFile` object for the input file that caused the message.
     - The start and end index of the part of the file text that caused the message.

     In this example, there are two error messages. So, `info.Messages` is a `2`-by-`1` array.

```
2×1 Message array with properties:

  Identifier
  Type
  Text
  File
  StartIndex
  EndIndex
```

     View the first element of the array `info.Messages(1)`.

```
Message with properties:

  Identifier: 'Coder:toolbox:unsupportedClass'
        Type: 'Error'
```

```
          Text: 'Function 'svd' is not defined for values of class 'string'.'
          File: [1×1 coder.CodeFile]
    StartIndex: 26
      EndIndex: 33
```

Use the StartIndex and EndIndex properties to index into the text of the file.

```
info.Messages(1).File.Text(26:33)
```

This command displays the part of the file text that caused the error message.

```
'svd(a,0)'
```

Inspect Code Manually

To manually inspect the text of the input files, the line and column numbers corresponding to the StartIndex and EndIndex values are useful. Use the getLineColumn function to obtain this information. This function returns two structures that contain the line and column numbers corresponding to StartIndex and EndIndex respectively.

In the preceding example, to manually inspect the part of foo.m that caused the first error message, display the text of the file.

```
info.Messages(1).File.Text
```

The text of the file is displayed as:

```
'function b = foo(a)
 b = svd(a,0);
 end
 '
```

Access the line and column numbers of the part of the text that caused the first error message.

```
[startLoc,endLoc] = getLineColumn(info.messages(1))
```

The output is:

```
startLoc = 

  struct with fields:

      Line: 2
    Column: 5


endLoc = 

  struct with fields:

      Line: 2
    Column: 12
```

These locations correspond to the beginning and the end of the function call 'svd(a,0)' in the text of foo.m.

Transferring Code Configuration Objects to a New MATLAB Session

Suppose that you create a report information object `info` in a MATLAB session, and then use it in another MATLAB session. If `info.Config` is a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), the following behavior might occur:

- If the MATLAB host computer for the second session does not have the hardware board specified in the `info.Config.Hardware` property installed on it, the configuration parameter `info.Config.Hardware` reverts to its default value. The default value is `[]`.
- If the MATLAB host computer for the second session does not have the toolchain specified in the `info.Config.Toolchain` property installed on it, the configuration parameter `info.Config.Toolchain` reverts to its default value. The default value is `'Automatically locate an installed toolchain'`.

See Also

coder.BuildLog Properties | coder.CodeFile Properties | coder.File Properties | coder.Function Properties | coder.Message Properties | coder.Method Properties | coder.ReportInfo Properties | coder.Summary Properties | `getLineColumn`

More About

- "Code Generation Reports" on page 28-7

Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors

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 standalone libraries and executables that detect and report run-time errors, such as out-of-bounds array indexing.

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 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.

Generated C Code vs. Generated C++ Code

If your target language is C, the generated code uses `fprintf` to write error messages to `stderr`. Then, the code 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.

If your target language is C++, the generated code throws `std::runtime_error` exceptions for the run-time errors. When you call the generated C++ entry-point functions, you can catch and handle these exceptions by using a `try-catch` block in your external C++ code.

However, for runtime error checks inside parallel regions (either `parfor` loops or automatically parallelized `for` loops), the generated C++ code does not throw an exception. In such situations, the generated code uses `fprintf` to write error messages to `stderr`, and then uses `abort` to terminate the application. To learn more about automatic parallelization, see “Automatically Parallelize for Loops in Generated Code” on page 34-69.

Limitations

Run-time error detection and reporting in standalone code has these limitations:

- 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.

- If the generated C code terminates, the error detection and reporting software does not release resources, such as allocated memory. The generated C++ code does not have this limitation. If the generated C++ code terminates, allocated memory and other resources are released.
- In standalone code, the function `error` displays a message that indicates that an error occurred. To see the actual message specified by `error`, you must generate and run a MEX function.
- 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.

Example: Compare Generated C and C++ Code That Include Run-Time Checks

In this example, you compare the run-time behavior of generated C and C++ code for a MATLAB® function that calculates the square root of its input argument. The generated code can accept only nonnegative real values and produces a run-time error for negative inputs:

- The generated C code uses `fprintf` to write the error message to `stderr`. Then, the code uses `abort` to terminate the application.
- The the generated C++ code throws a `std::runtime_error` exception for this run-time error. In the C++ main function that you write to call the generated function, you catch and handle this exception by using a `try-catch` block.

Define MATLAB Function

Define the MATLAB function `errorCheckExample` in a separate file. This function calculates the square root of its input argument:

```
type errorCheckExample
```

```
function y = errorCheckExample(x)
y = sqrt(x);
end
```

Generate C Library and Executable

Generate a dynamically linked C library for `errorCheckExample` that accepts a double scalar input. Use a code configuration object with the `RuntimeChecks` parameter set to `true`. Also, use the `-d` option to name the code generation folder as `codegen_c_dll`.

```
cfg = coder.config('dll');
cfg.RuntimeChecks = true;
codegen -config cfg errorCheckExample -args 1 -d codegan_c_dll -report
```

```
Code generation successful: To view the report, open('codegan_c_dll\html\report.mldatx').
```

Open the code generation report and inspect the file `errorCheckExample.c`. The C function generated for your MATLAB function has the signature `double errorCheckExample(double x)`. To calculate the square root, `errorCheckExample` invokes the `sqrt` library function which calculates only real square roots. So, `errorCheckExample` can accept only positive inputs. For negative inputs, `errorCheckExample` calls the generated utility function `rtErrorWithMessageID` that uses `fprintf` to write an error message to `stderr` and then uses `abort` to terminate the application.

```
static void rtErrorWithMessageID(const int b, const char *c,
                                 const char *aFcnName, int aLineNum)
```

```
{
  fprintf(stderr,
          "Domain error. To compute complex results from real x, use "
          "\'%.*s(complex(x))\'.",
          b, c);
  fprintf(stderr, "\n");
  fprintf(stderr, "Error in %s (line %d)", aFcnName, aLineNum);
  fprintf(stderr, "\n");
  fflush(stderr);
  abort();
}
```

When generating library code, the code generator also produces example main files `main.h` and `main.c` in the `examples` subfolder of the build folder. The supporting C files `main_runtime_check.h` and `main_runtime_check.c` are modified versions of these example files. The modified `main` function invokes `errorCheckExample(-4)`, which produces a run-time error.

Run these commands to generate a C executable using the modified main files. Name the code generation folder `codegen_c_exe`. Name the executable file `errorCheckExample_c` by using the `-o` option with the `codegen` command.

```
cfg = coder.config('exe');
cfg.RuntimeChecks = true;
cfg.CustomSource = 'main_runtime_check.c';
cfg.CustomInclude = pwd;

codegen -config cfg main_runtime_check.c main_runtime_check.h errorCheckExample -args 1 -o errorC
```

```
Code generation successful.
```

Run the generated executable. Observe that it prints the error message that is hard-coded in the utility function `rtErrorWithMessageID`.

```
if isunix
    system('./errorCheckExample_c');
elseif ispc
    system('errorCheckExample_c.exe');
else
    disp('Platform is not supported');
end
```

```
Domain error. To compute complex results from real x, use 'sqrt(complex(x))'.
Error in sqrt (line 13)
```

Generate C++ Library and Executable

Generate a dynamically linked C++ library for `errorCheckExample` that accepts a scalar double input. Use a code configuration object with the `RuntimeChecks` parameter set to `true`. Also, use the `-d` option to name the code generation folder as `codegen_cpp_dll`.

```
cfg = coder.config('dll');
cfg.RuntimeChecks = true;
codegen -config cfg -lang:c++ errorCheckExample -args 1 -d codegen_cpp_dll -report
```

```
Code generation successful: To view the report, open('codegen_cpp_dll\html\report.mldatx').
```

Open the code generation report and inspect the file `errorCheckExample.cpp`. Similar to the C function generated in the previous section, `errorCheckExample` can accept only positive inputs. For

negative inputs, `errorCheckExample` calls the utility function `rtErrorWithMessageID`. But in this case, the utility function throws a `std:runtime_error` exception that you can catch and handle in your hand-written `main` function.

```
static void rtErrorWithMessageID(const char *b, const char *aFcnName,
                                 int aLineNum)
{
  std::stringstream outStream;
  ((outStream << "Domain error. To compute complex results from real x, use \'")
   << b)
      << "(complex(x))\'.";
  outStream << "\n";
  ((((outStream << "Error in ") << aFcnName) << " (line ") << aLineNum) << ")";
  throw std::runtime_error(outStream.str());
}
```

When generating library code, the code generator also produces example main files `main.h` and `main.c` in the `examples` subfolder of the build folder. The supporting C++ files `main_runtime_check.hpp` and `main_runtime_check.cpp` are modified versions of these example files. The modified `main()` function invokes `errorCheckExample(-4)` inside a `try-catch` block. The block catches the exception and prints a modified message by prepending the string `"Caught excaption: "` to the message that the caught exception contains.

Run these commands to generate a C++ executable using the modified main files. Name the code generation folder `codegen_cpp_exe`. Name the executable file `errorCheckExample_cpp`.

```
cfg = coder.config('exe');
cfg.RuntimeChecks = true;
cfg.CustomSource = 'main_runtime_check.cpp';
cfg.CustomInclude = pwd;

codegen -config cfg -lang:c++ main_runtime_check.cpp main_runtime_check.hpp errorCheckExample -a

Code generation successful.
```

Run the generated executable. Observe that it prints the modified error message.

```
if isunix
    system('./errorCheckExample_cpp');
elseif ispc
    system('errorCheckExample_cpp.exe');
else
    disp('Platform is not supported');
end

Caught exception: Domain error. To compute complex results from real x, use 'sqrt(complex(x))'.
Error in sqrt (line 13)
```

See Also

`codegen` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

More About

- “Why Test MEX Functions in MATLAB?” on page 26-2

Example: Generate Standalone C Code That Detects and Reports Run-Time Errors

This example shows how to generate C libraries or executables that detect and report run-time errors such as out-of-bounds array indexing. If the generated C code detects an error, it reports a message and terminates the program. This allows you to 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) %#codegen
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 = 1;
```

Generate code.

```
codegen -config cfg getelement -args {S}
```

To open the code generation report, click the **View report** link.

In the list of generated files, click `getelement.c`.

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 "getelement.h"
#include "getelement_rtwutil.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/* Variable Definitions */
static rtBoundsCheckInfo emlrtBCI = { 1,/* iFirst */
  4,                                    /* iLast */
  2,                                    /* lineNo */
  5,                                    /* colNo */
  "S.A",                                /* aName */
  "getelement",                         /* fName */
```

```
  "C:\\Users\\username\\Documents\\MATLAB\\runtime-error-ex\\getelement.m",/* pName */
  0                                    /* checkKind */
};

static rtDoubleCheckInfo emlrtDCI = { 2,/* lineNo */
  5,                                   /* colNo */
  "getelement",                        /* fName */
  "C:\\Users\\username\\Documents\\MATLAB\\runtime-error-ex\\getelement.m",/* pName */
  1                                    /* checkKind */
};

/* Function Definitions */
double getelement(const struct0_T *S)
{
  int i;
  if (S->u != (int)floor(S->u)) {
    rtIntegerError(S->u, &emlrtDCI);
  }

  i = (int)S->u;
  if ((i < 1) || (i > 4)) {
    rtDynamicBoundsError(i, 1, 4, &emlrtBCI);
  }

  return S->A[i - 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.

See Also

More About

- "Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors" on page 28-20

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.

See Also

More About

- “Verify MEX Functions in the MATLAB Coder App” on page 26-7
- “Verify MEX Functions at the Command Line” on page 26-8
- “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 28-27
- “Unit Test External C Code with MATLAB Coder” on page 28-33

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”.

```
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.

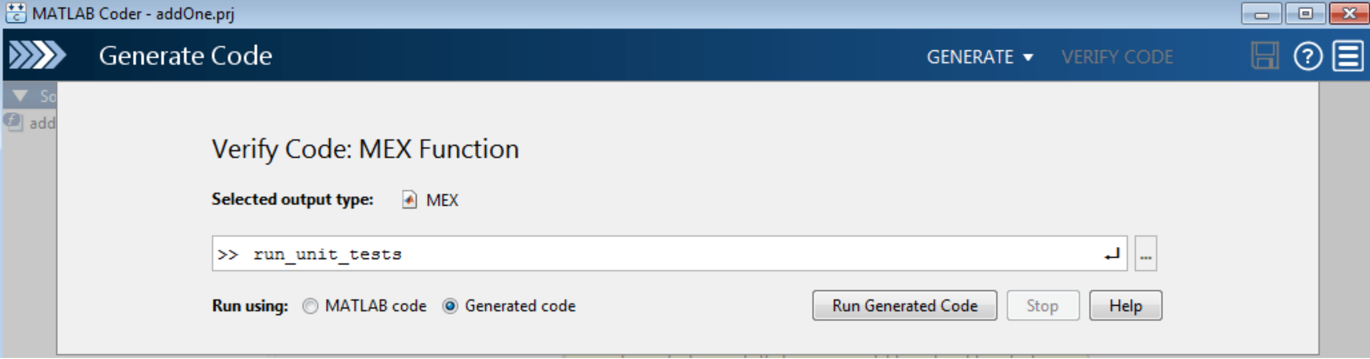
To generate the MEX function, on the **Generate Code** page:

1. For **Build type**, specify `MEX`.
2. Click **Generate**.

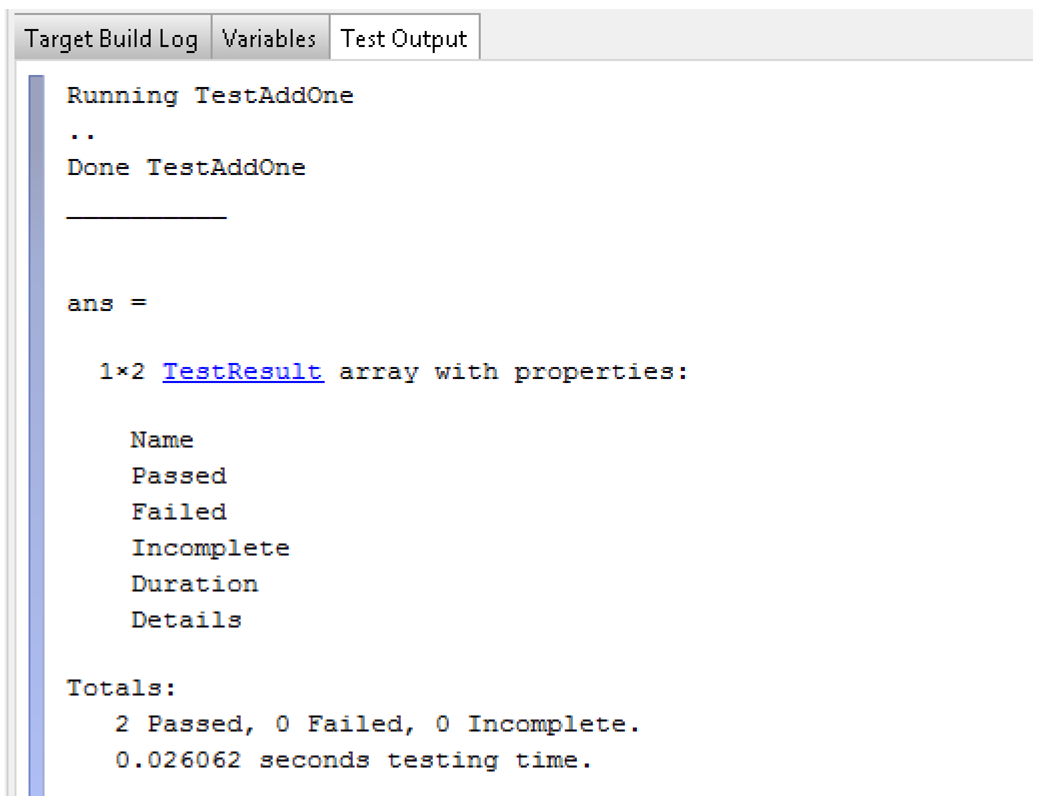
Run the unit tests on the generated MEX.

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**.

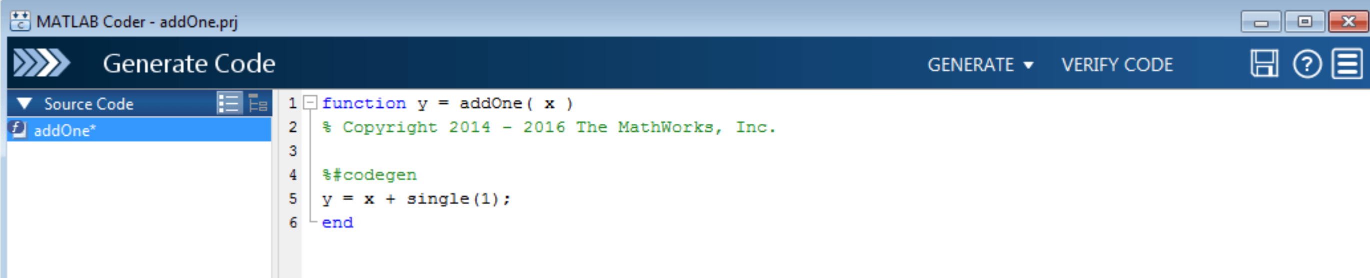


The app displays the test output on the **Test Output** tab. The unit tests pass.



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`.



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.

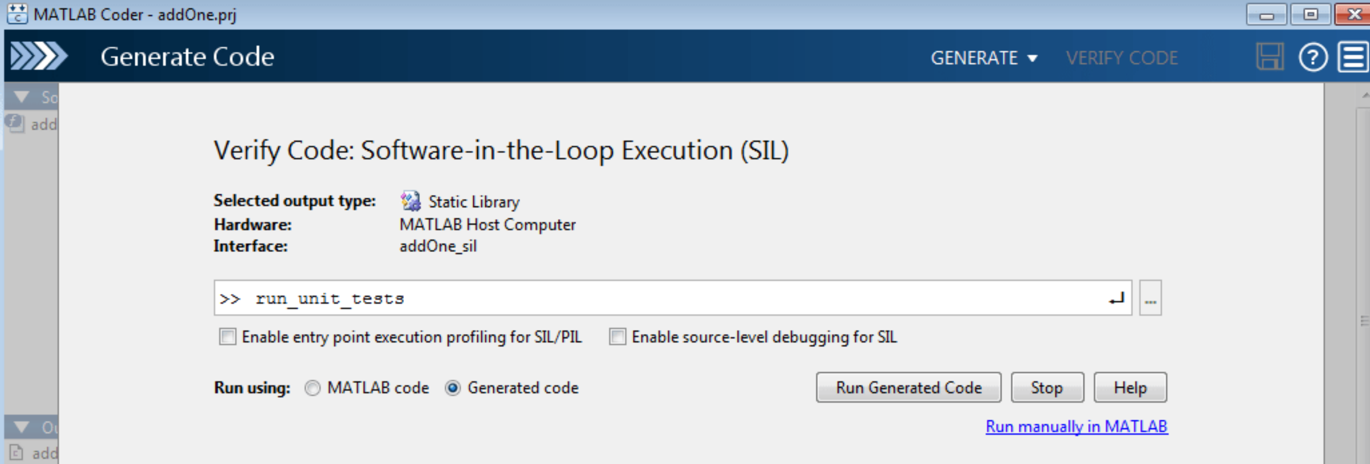
Generate a library for `addOne`. For example, generate a static library.

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**.



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}
```

```
Code generation successful.
```

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 = 

  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
```

```
0.040987 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`

More About

- “Author Class-Based Unit Tests in 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 28-33

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".

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(C) ']'])
        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.

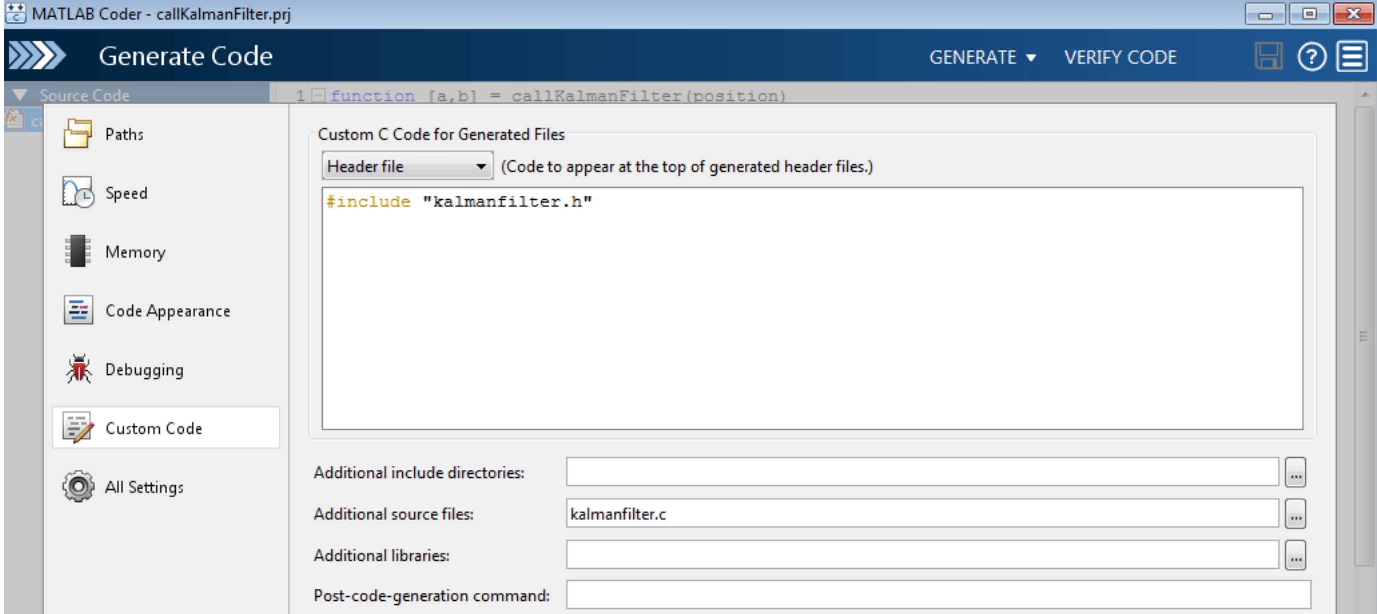
The unit tests load the variable `position` from `position.mat` and pass `position` to `callKalmanFilter`. Therefore, the input to `callKalmanFilter` must have the properties that `position` has. In the MATLAB workspace, if you load `position.mat`, you see that `position` is a 2-by-310 array of doubles.

- 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:

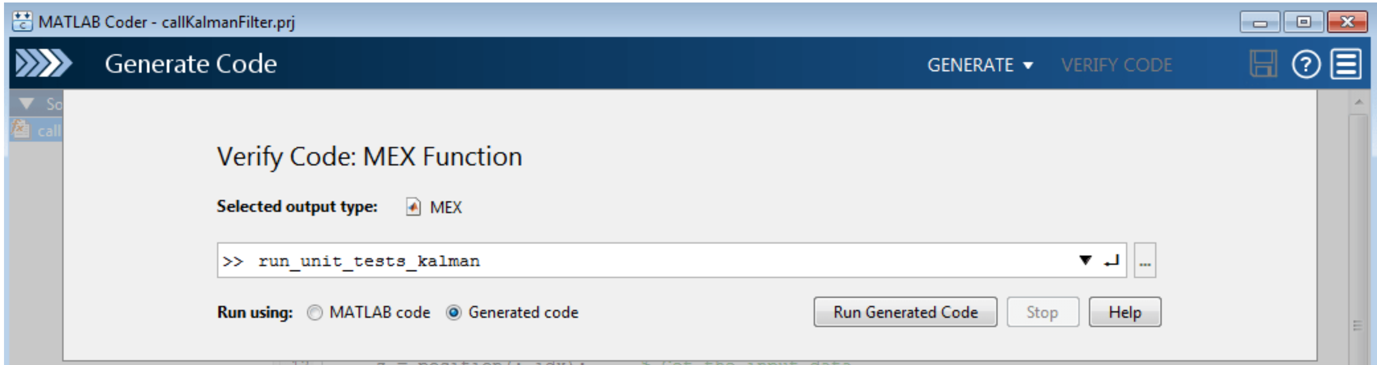
- Under **Custom C Code for Generated Files**, select **Header file**. In the custom code field, enter `#include "kalmanfilter.h"`.
- In the **Additional source files** field, enter `kalmanfilter.c`.



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**.

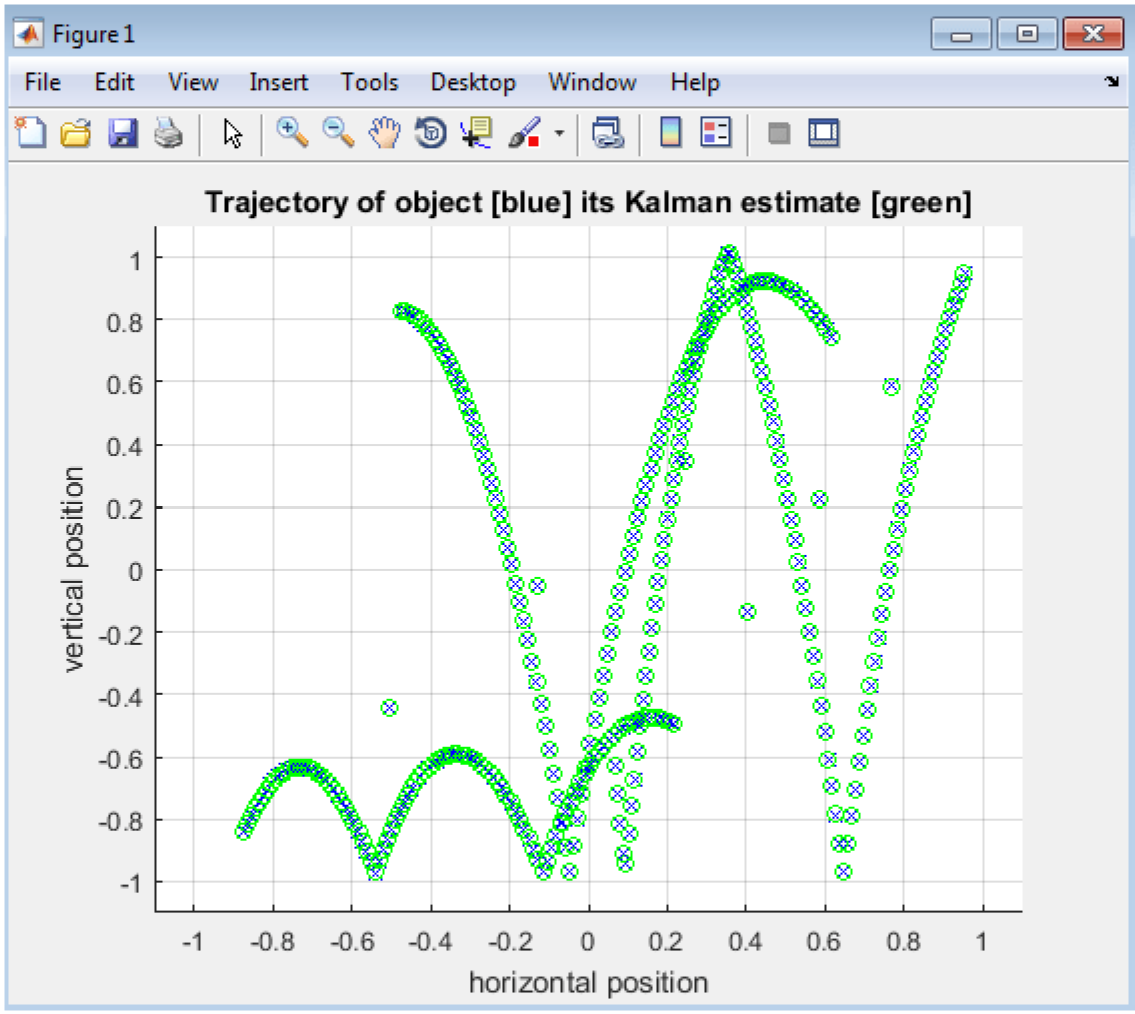


When the app runs the test file, it replaces calls to `callKalmanFilter` in the unit test with calls to `callKalmanFilter_mex`. The unit tests run on the MEX function instead of the original MATLAB function.

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**.

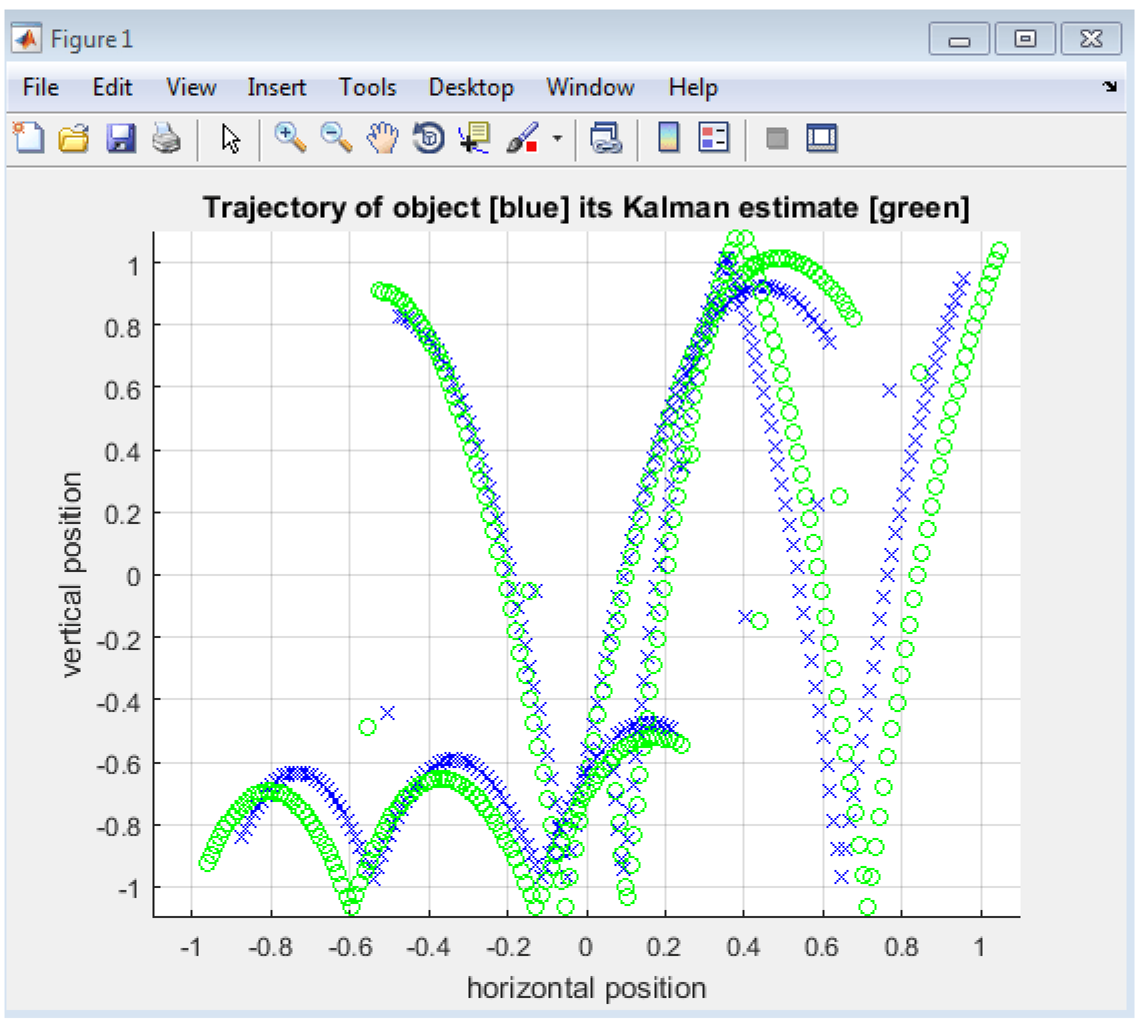
To run the unit tests:

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`.

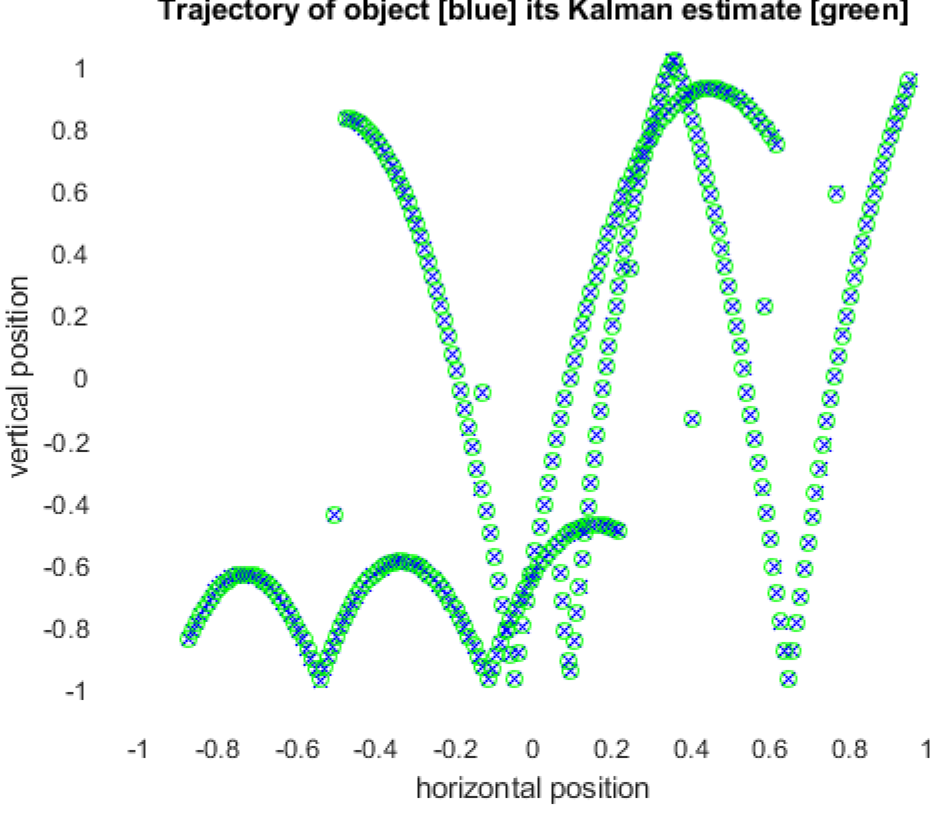
```
codegen -config cfg callKalmanFilter -args position
```

```
Code generation successful.
```

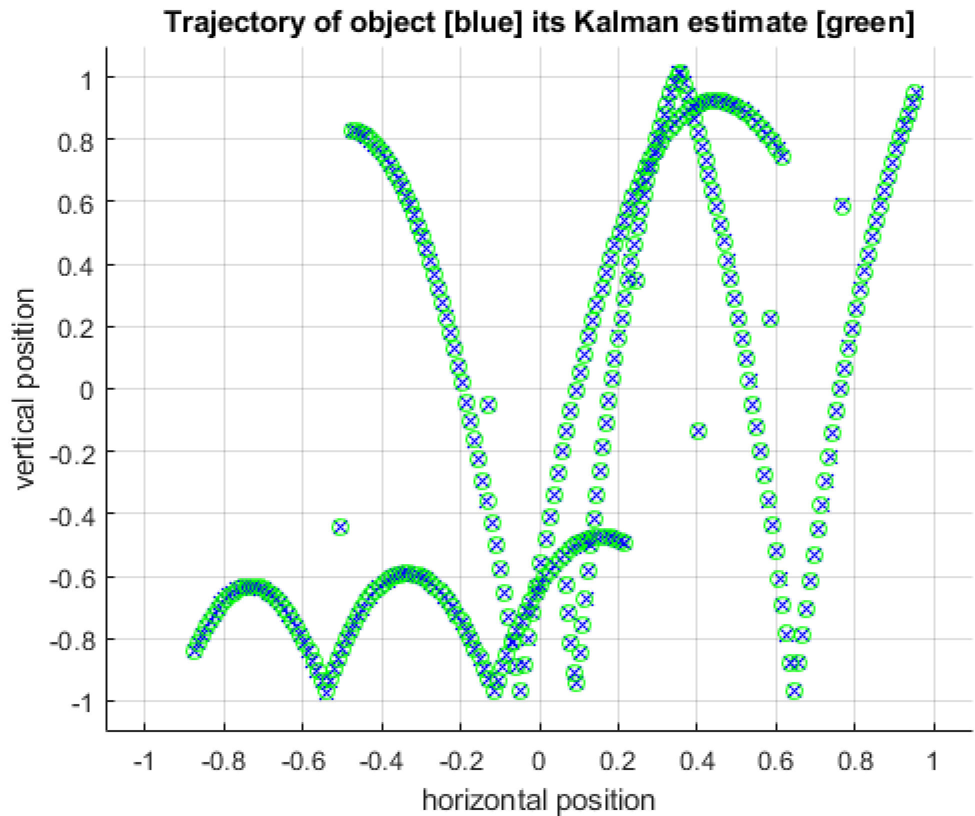
Run the units tests on the MEX function. Specify that the test file is `run_unit_tests_kalman` and that the function is `callKalmanfilter`. When `coder.runTest` runs the test file, it replaces calls to `callKalmanFilter` in the unit test with calls to `callKalmanFilter_mex`. The unit tests run on the 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 = 

  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   21.1278 seconds testing time.
```



Trajectory of object [blue] its Kalman estimate [green]
vertical position
horizontal position
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1



See Also

`coder.runTest`

More About

- “Author Class-Based Unit Tests in 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 28-27

Calculate Number of Lines of Code by Using Report Information Object

This example shows how to calculate the number of lines in the source code and the generated code by using the report information object. For more information on the report information object, see “Access Code Generation Report Information Programmatically” on page 28-13.

Setup

Add example files to path.

```
path = fullfile(matlabroot, 'examples', 'coder', 'main');
addpath(path);
```

The MATLAB Code

In this example, you generate code for the MATLAB function `dijkstra`. This function calculates the lengths of the shortest paths from a node to every other node in a graph by using Dijkstra's algorithm.

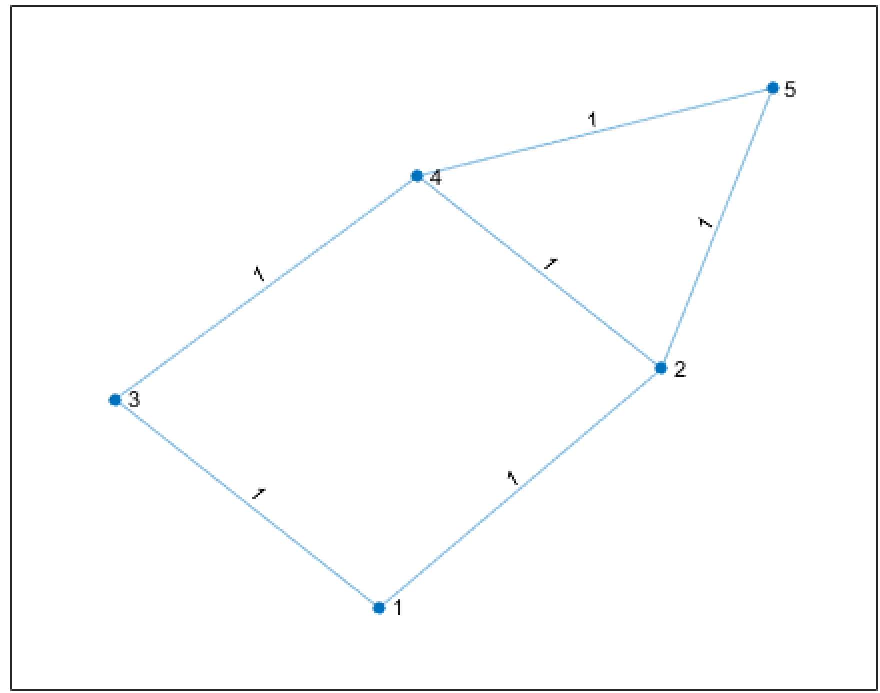
```
type dijkstra
```

```
% DIJKSTRA Find length of shortest path between nodes in a graph
%
% D = dijkstra(A, p)
% Takes a graph represented by its adjacency matrix 'A' along with a node
% 'p' as input and returns a vector 'D' containing the length of the
% shortest path from 'p' to all other nodes in the graph.

% Copyright 2018 The MathWorks, Inc.
function D = dijkstra(A, p) %#codegen

    narginchk(2,2);

    [m, n] = size(A);

    % Assertions to make sure inputs are valid
    assert(m == n, "Input adjacency matrix for graph must be a square matrix");
    assert(rem(p, 1) == 0 && p <= m && p > 0, "Input src must be a node in the graph");

    % Initialization
    max = realmax;
    D = repmat(max, 1, m);
    D(p) = 0;
    visited = false(1, m);

    for i = 1:m
        % Select next node to visit
        min = max;
        u = -1;
        for v = 1:n
            if ~visited(v) && D(v) <= min
                min = D(v);
                u = v;
            end
        end
```

```
        % Mark selected node as visited
        visited(u) = true;

        %{
          Update distances of nodes adjacent to selected node that are yet
          to be visited
        %}
        for v = 1:n
            if(~visited(v) && A(u, v) ~= 0 && D(u) ~= max)
                distVal = D(u) + A(u, v);
                if distVal < D(v)
                    D(v) = distVal;
                end
            end
        end
    end
end
```

Specify the adjacency matrix A for a graph and a node p where the traversal of the graph begins. Plot the graph. Call dijkstra to compute the shortest distance from p to every node in the graph and display these distances.

```
% Sample adjacency Matrix for graph with 5 nodes
A = [
        0 1 1 0 0;
        1 0 0 1 1;
        1 0 0 1 0;
        0 1 1 0 1;
        0 1 0 1 0
    ];

% Plot the graph to see how it looks like
G = graph(A, 'omitselfloops');
plot(G, 'EdgeLabel', G.Edges.Weight)
```




```
% Source node from where graph traversal begins
p = randi(size(A, 1));

% Calculate shortest distance from 'p' to every other node in graph G
D = dijkstra(A, p);

for i=1:numel(D)
    fprintf("Length of shortest path from %d to %d is %d. \n", p, i, D(i));
end
```

```
Length of shortest path from 5 to 1 is 2.
Length of shortest path from 5 to 2 is 1.
Length of shortest path from 5 to 3 is 2.
Length of shortest path from 5 to 4 is 1.
Length of shortest path from 5 to 5 is 0.
```

Export Information about Code Generation

The report information object provides programmatic access to information about code generation. The properties of this object provide information about code generation settings, input files, generated files, and code generation error, warning, and information messages.

To export the report information object to a variable in your base MATLAB workspace, include the `-reportinfo` option with the name of the variable while running the codegen command. In this example, you export the code generation report information to the variable `info`.

```
codegen -c dijkstra -args {A, p} -reportinfo info
```

```
Code generation successful.
```

Calculate Number of Lines of Code

The `loc` function takes a report information object as input and returns two outputs that contain the number of lines in the source code and the generated code, respectively. This function excludes blank lines and the lines containing comments while computing the number of lines of code.

```
type loc
```

```
% LOC Calculate total lines of source and generated code in a codegen run
%
% [i, o] = loc(r)
% Takes a report information object 'r' as input, and produces two
% outputs - 'i' and 'o' containing the total lines of code in the source
% MATLAB files and generated files respectively.

% Copyright 2018 The MathWorks, Inc.
function [i, o] = loc(r)
    narginchk(1,1);

    % Assert that input is a report information object.
    assert(isa(r, 'coder.ReportInfo'), 'Input must be of type coder.ReportInfo');

    % Fetch source and generated files from the report information object.
    sourceFiles = r.InputFiles;
    generatedFiles = r.GeneratedFiles;

    % Count lines of code in source and generated files. Blank lines, and
    % comments are not counted.
    i = countLines(sourceFiles, true);
    o = countLines(generatedFiles, false);
end

function count = countLines(files, isSource)
    count = 0;
    for i=1:numel(files)
        f = files(i);
        if isprop(f, 'Text')
            lines = splitlines(f.Text);
            for j=1:numel(lines)
                line = strtrim(lines{j});
                if ~isempty(line) && ~isComment(line, isSource)
                    count = count + 1;
                end
            end
            clear isComment; % clear persistent variables
        end
    end
end

function result = isComment(line, isSource)
    persistent inBlockComment;
    if isempty(inBlockComment)
        inBlockComment = false;
    end
    if isSource
        result = (startsWith(line, "%") || inBlockComment);
```

```
        if line == "%{" || line == "%}"
            inBlockComment = (line ~= "%}");
        end
    else
        result = (startsWith(line, "/") || inBlockComment);
        if startsWith(line, "/*") || endsWith(line, "*/")
            inBlockComment = ~endsWith(line, "*/");
        end
    end
end
```

Call `loc` with the report information object `info` as input. Display the number of lines of code in the source files and the generated files.

```
info = evalin('base', 'info');
[nLocIn, nLocOut] = loc(info);
fprintf('Lines of code in source MATLAB file(s): %d', nLocIn);
```

```
Lines of code in source MATLAB file(s): 29
```

```
fprintf('Lines of code in generated file(s): %d', nLocOut);
```

```
Lines of code in generated file(s): 546
```

Cleanup

Remove example files from path.

```
rmpath(path);
```

See Also

coder.ReportInfo Properties

More About

- “Access Code Generation Report Information Programmatically” on page 28-13

Code Replacement for MATLAB Code

- “What Is Code Replacement?” on page 29-2
- “Choose a Code Replacement Library” on page 29-6
- “Replace Code Generated from MATLAB Code” on page 29-8

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 libraries that MathWorks provides and that you create and register by using the Embedded Coder product. The list of available libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you have created and registered libraries, using the Embedded Coder product.

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.

Code Replacement Libraries

A code replacement library consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A code replacement table contains one or more code replacement entries, with each entry representing a potential replacement for a function or operator. Each entry maps a conceptual representation of a function or operator to an implementation representation and priority.

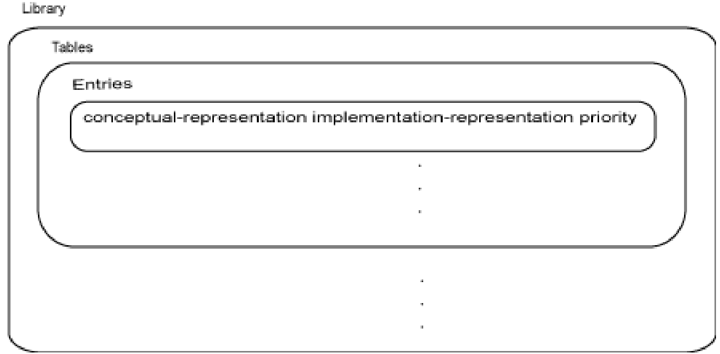


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies 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.
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 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.

See Also

Related Examples

- “Choose a Code Replacement Library” on page 29-6
- “Replace Code Generated from MATLAB Code” on page 29-8

Choose a Code Replacement Library

In this section...
“About Choosing a Code Replacement Library” on page 29-6
“Explore Available Code Replacement Libraries” on page 29-6
“Explore Code Replacement Library Contents” on page 29-6

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 29-2.
   - See “Explore Available Code Replacement Libraries” on page 29-6.
2. Explore the contents of the library. See “Explore Code Replacement Library Contents” on page 29-6.

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).

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.

See Also

Related Examples

- “What Is Code Replacement?” on page 29-2
- “Replace Code Generated from MATLAB Code” on page 29-8

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 Replacement Library” (Embedded Coder).

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.

See Also

Related Examples

- “What Is Code Replacement?” on page 29-2
- “Choose a Code Replacement Library” on page 29-6
- “Configure Build Settings” on page 27-13

Custom Toolchain Registration

- "Custom Toolchain Registration" on page 30-2
- "About coder.make.ToolchainInfo" on page 30-5
- "Create and Edit Toolchain Definition File" on page 30-7
- "Toolchain Definition File with Commentary" on page 30-8
- "Create and Validate ToolchainInfo Object" on page 30-13
- "Register the Custom Toolchain" on page 30-14
- "Use the Custom Toolchain" on page 30-16
- "Troubleshooting Custom Toolchain Validation" on page 30-17
- "Prevent Circular Data Dependencies with One-Pass or Single-Pass Linkers" on page 30-20
- "Build 32-bit DLL on 64-bit Windows® Platform Using MSVC Toolchain" on page 30-21

Custom Toolchain Registration

In this section...

"What Is a Custom Toolchain?" on page 30-2
"What Is a Factory Toolchain?" on page 30-2
"What is a Toolchain Definition?" on page 30-2
"Key Terms" on page 30-3
"Typical Workflow" on page 30-3

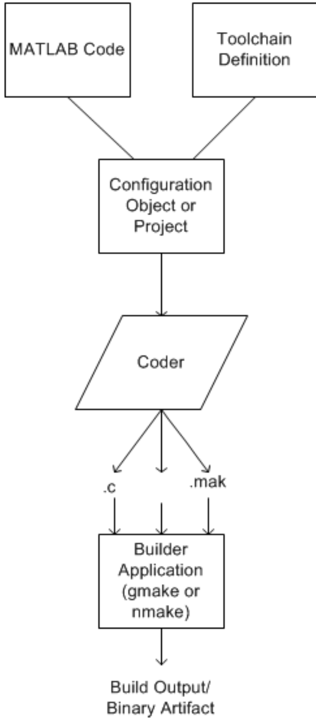
What Is a Custom Toolchain?

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 https://www.mathworks.com/support/compilers/.

What is a Toolchain Definition?



A toolchain definition provides MATLAB Coder software with information about the software build tools, such as the compiler, linker, archiver. MATLAB Coder software uses this information, along with a configuration object or project, to build the generated code. This approach can be used when generating static libraries, dynamic libraries, and executables. MEX-file generation uses a different approach. To specify which compiler to use for MEX-function generation, see “Setting Up the C or C++ Compiler”.

MATLAB Coder software comes with a set of registered factory toolchain definitions. You can create and register custom toolchain definitions. You can customize and manage toolchain definitions. You can share custom toolchain definitions with others running MATLAB Coder software.

If you install toolchain software for one of the factory toolchains, MATLAB Coder can automatically detect and use the toolchain software. For more information about factory toolchains in MATLAB Coder software, see https://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 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 30-7
   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 30-13

   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 30-14

   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 30-16

   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 “Add Custom Toolchains to MATLAB® Coder™ Build Process” on page 27-165.

For more information about the `ToolchainInfo` object, see “About coder.make.ToolchainInfo” on page 30-5.

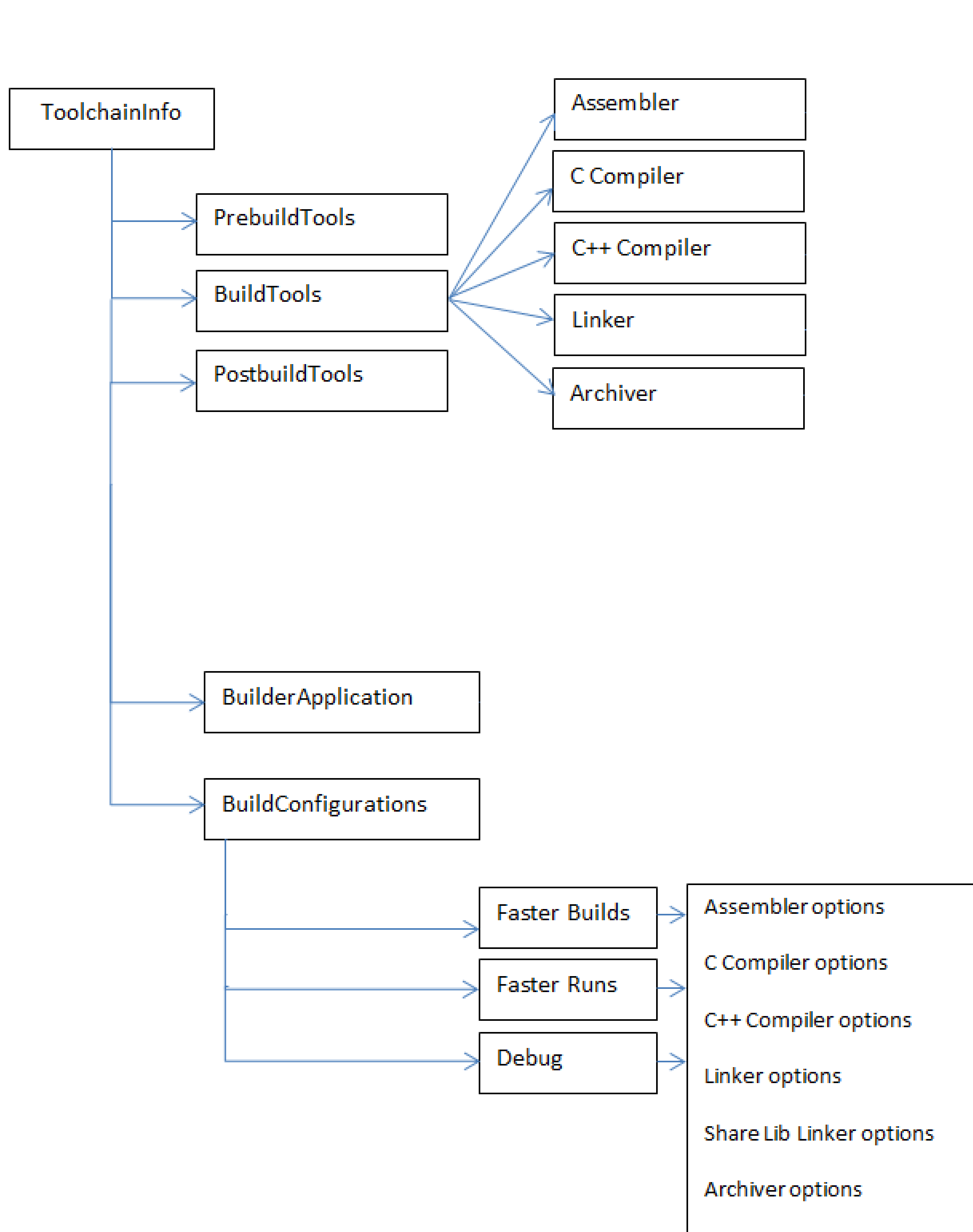
About coder.make.ToolchainInfo

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
Assembler
C Compiler
C++ Compiler
Linker
Archiver
BuilderApplication
BuildConfigurations
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 30-3.

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 development computer environment, and previously-registered custom toolchains.
2. Create the folder of example files by opening the “Add Custom Toolchains to MATLAB® Coder™ Build Process” on page 27-165 example.
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 30-8.

   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 30-13

Toolchain Definition File with Commentary

In this section...

"Steps Involved in Writing a Toolchain Definition File" on page 30-8
"Write a Function That Creates a ToolchainInfo Object" on page 30-8
"Setup" on page 30-9
"Macros" on page 30-9
"C Compiler" on page 30-9
"C++ Compiler" on page 30-10
"Linker" on page 30-10
"Archiver" on page 30-11
"Builder" on page 30-11
"Build Configurations" on page 30-11

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 "Add Custom Toolchains to MATLAB® Coder™ Build Process" on page 27-165. For more information about the workflow, see "Typical Workflow" on page 30-3.

Write a Function That Creates a ToolchainInfo Object

```
function tc = intel_tc
% INTEL_TC Creates an Intel v12.1 ToolchainInfo object.
% This can be used as a template to add other toolchains on Windows.

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','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 30-7.

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 30-14.

For more information, see “Troubleshooting Custom Toolchain Validation” on page 30-17.

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 30-13.

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”.
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 30-16.

See Also

More About

- “Add Custom Toolchains to MATLAB® Coder™ Build Process” on page 27-165

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 30-14.

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)'
   ```
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 30-2.

Troubleshooting Custom Toolchain Validation

In this section...

"Build Tool Command Path Incorrect" on page 30-17
"Build Tool Not in System Path" on page 30-17
"Tool Path Does Not Exist" on page 30-18
"Path Incompatible with Builder or Build Tool" on page 30-18
"Unsupported Platform" on page 30-18
"Toolchain is Not installed" on page 30-18
"Project or Configuration Is Using the Template Makefile" on page 30-19

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 `''`.

Path Incompatible with Builder or Build Tool

If the file separator character in the build tool path (for example `'/'` or `'\'`) is not compatible with the builder application, validation can display:

```
Path 'toolpath' does not exist.
```

To correct this issue, check that the file separators in the toolchain definition match the `'FileSeparator'` accepted by the `tc.BuilderApplication` when the specified path is used by the make file. Then, update the value of `coder.make.BuildTool.setPath` in the toolchain definition file.

Most toolchains and build tools (LCC being a notable exception) recognize `'/'` as a file separator. To get your custom toolchain definitions to behave as expected, try using `'/'` as the file separator.

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 30-16.

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.

Note Support for template makefiles (TMF) will be removed in a future release.

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);
  ```

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. The build process applies the grouping directives to model reference libraries `$(MODELREF_LIBS)` and user libraries `$(LIBS)`.

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');
```

Build 32-bit DLL on 64-bit Windows® Platform Using MSVC Toolchain

Register and use a Microsoft® Visual C/C++ (MSVC) toolchain running on a 64-bit Windows® platform to compile a 32-bit dynamic link library (DLL). This example uses a Microsoft® compiler. However, the concepts and programming interface apply for other toolchains. Once you register the toolchain, you can select it from a list of toolchains, and the code generator generates a makefile to build the code by using that toolchain. A toolchain consists of several tools, such as a compiler, linker, and archiver with multiple different configuration options. The toolchain compiles, links, and runs code on a specified platform. To access the files that this example uses, click **Open Script**.

Check Platform and Determine MSVC Version

This code checks that the platform is supported and that you have a supported version of Microsoft® Visual C/C++. The `my_msvc_32bit_tc.m` toolchain definition can use the Microsoft® Visual Studio versions 9.0, 10.0, 11.0, 12.0, 14.0, or 15.0.

If you are not using a Windows® platform, or if you do not have a supported version of Microsoft® Visual C/C++, the example generates only code and a makefile, without running the generated makefile.

```
VersionNumbers = {'14.0'}; % Placeholder value
if ~ispc
    supportedCompilerInstalled = false;
else
    installed_compilers = mex.getCompilerConfigurations('C', 'Installed');
    MSVC_InstalledVersions = regexp({installed_compilers.Name}, 'Microsoft Visual C\+\+ 20\d\d')
    MSVC_InstalledVersions = cellfun(@(a)~isempty(a), MSVC_InstalledVersions);
    if ~any(MSVC_InstalledVersions)
        supportedCompilerInstalled = false;
    else
        VersionNumbers = {installed_compilers(MSVC_InstalledVersions).Version}';
        supportedCompilerInstalled = true;
    end
end
```

Function for the Dynamic Link Library

The example function for the dynamic link library, `myMatlabFunction.m`, multiplies a number by two.

```
function y = myMatlabFunction(u)
% myMatlabFunction: Returns twice its input.
% Copyright 2017 The MathWorks, Inc.

%#codegen
assert(isa(u, 'double'), 'The input must be a "double".');
assert(all([1, 1] == size( u )), 'The input must be a scalar.');

y = double(u + u);
```

Create and Configure an MSVC Toolchain

The `my_msvc_32bit_tc.m` toolchain definition function takes in an argument containing the Visual Studio version number. In this example, the commands that create and configure this toolchain are:

```
tc = my_msvc_32bit_tc(VersionNumbers{end});
save my_msvc_32bit_tc tc;
```

```
Executing "H:\Examples\coder-ex19875030\my_msvc_32bit_tc"...
Executed "H:\Examples\coder-ex19875030\my_msvc_32bit_tc".
```

Register the Toolchain

Before the code generator can use a toolchain for the build process, the `RTW.TargetRegistry` must contain the toolchain registration. This registration can come from any `rtwTargetInfo.m` file on the MATLAB path. MATLAB will load a new registration if the `RTW.TargetRegistry` is reset.

Create the `rtwTargetInfo.m` file from the corresponding text file `myRtwTargetInfo.txt`.

```
function myRtwTargetInfo(tr)
%RTWTARGETINFO Registration file for custom toolchains.

% Copyright 2012-2017 The MathWorks, Inc.

tr.registerTargetInfo(@createToolchainRegistryFor32BitMSVCToolchain);

end

% -------------------------------------------------------------------------
% Create the ToolchainInfoRegistry entries
% -------------------------------------------------------------------------
function config = createToolchainRegistryFor32BitMSVCToolchain

config(1)                       = coder.make.ToolchainInfoRegistry;
config(1).Name                  = 'Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)'
config(1).FileName              = fullfile(fileparts(mfilename('fullpath')), 'my_msvc_32bit_tc.ma
config(1).TargetHWDeviceType    = {'Intel->x86-32 (Windows32)','AMD->x86-32 (Windows32)','Generic
config(1).Platform              =  {'win64'};

end
```

```
copyfile myRtwTargetInfo.txt rtwTargetInfo.m
RTW.TargetRegistry.getInstance('reset');
```

Create Code Generation Configuration Object

To generate the 32-bit dynamic link library (DLL), create a `'dll'` code generation configuration object. Specifying `'dll'` directs the linker (a build tool in the toolchain) to use "Shared Library" linker commands.

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

Configure Code Generation for 32-bit Hardware

To successfully generate code that is compatible with 32-bit hardware, the generated code must use the correct underlying C types (for example, `int`, `signed char`, and others). These types are the

basis for `typedef` statements for sized types (for example, `uint8`, `int16`, and others). Set the configuration with the command:

```
cfg.HardwareImplementation.ProdHWDeviceType = ...
    'Generic->Unspecified (assume 32-bit Generic)';
```

Configure Code Generation to Use the 32-bit Toolchain

Set the name of the `Toolchain` property to match the `Name` that you specify in the `rtwTargetInfo.m` file.

```
cfg.Toolchain = ...
    'Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)';
```

Select Verbose Status Reporting

To provide confirmation of compiler flags that the toolchain uses to build the DLL, select verbose status reporting.

```
cfg.Verbose = true;
```

Determine Whether to Generate Code Only

When the Microsoft® compilers are not installed, the code generator generates only code and the makefile. When the supported compilers are installed, the code generator builds the 32-bit binary file.

```
if supportedCompilerInstalled
    cfg.GenCodeOnly = false;
else
    cfg.GenCodeOnly = true;
end
```

Generate Code and Build a DLL

To use the toolchain for code generation and build the DLL (if build is enabled), at the command prompt, enter:

```
codegen -config cfg myMatlabFunction -args { double(1.0) };
```

```
### Using toolchain: Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)
### Creating 'H:\Examples\coder-ex19875030\codegen\dll\myMatlabFunction\myMatlabFunction_rtw.mk'
### Building 'myMatlabFunction': nmake  -f myMatlabFunction_rtw.mk all

H:\Examples\coder-ex19875030\codegen\dll\myMatlabFunction>set "VSCMD_START_DIR=H:\Examples\coder

H:\Examples\coder-ex19875030\codegen\dll\myMatlabFunction>call "C:\Program Files (x86)\Microsoft
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.0.26730.12
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64_x86'

Microsoft (R) Program Maintenance Utility Version 14.11.25507.1
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
myMatlabFunction_initialize.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
```

```
myMatlabFunction_terminate.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
myMatlabFunction.c
### Creating dynamic library ".\myMatlabFunction.dll" ...
    link /MACHINE:X86 /DEBUG /DEBUGTYPE:cv  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib msws
   Creating library .\myMatlabFunction.lib and object .\myMatlabFunction.exp
### Created: .\myMatlabFunction.dll
### Successfully generated all binary outputs.
```

Build and Run an Executable

If you have a supported version of the compiler installed, you can build the 32-bit executable by using a C main function. You can use the executable to test that the generated code works as expected.

```
cfge = coder.config('exe');
cfge.CustomInclude = pwd;
cfge.CustomSource = 'myMatlabFunction_main.c';
cfge.GenCodeOnly = cfg.GenCodeOnly;
cfge.Verbose = true;
cfge.Toolchain = ...
    'Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)';
codegen -config cfge myMatlabFunction -args { double(1.0) };
if supportedCompilerInstalled
    pause(5); %wait for EXE to get generated
    system('myMatlabFunction 3.1416'); % Expected output: myMatlabFunction(3.1416) = 6.2832
end
```

```
### Using toolchain: Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)
### Creating 'H:\Examples\coder-ex19875030\codegen\exe\myMatlabFunction\myMatlabFunction_rtw.mk'
### Building 'myMatlabFunction': nmake  -f myMatlabFunction_rtw.mk all

H:\Examples\coder-ex19875030\codegen\exe\myMatlabFunction>set "VSCMD_START_DIR=H:\Examples\coder

H:\Examples\coder-ex19875030\codegen\exe\myMatlabFunction>call "C:\Program Files (x86)\Microsoft
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.0.26730.12
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64_x86'

Microsoft (R) Program Maintenance Utility Version 14.11.25507.1
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
myMatlabFunction_initialize.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
myMatlabFunction_terminate.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
myMatlabFunction.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS  /Od /Oy-  -DMODEL=myMat
myMatlabFunction_main.c
### Creating standalone executable "H:\Examples\coder-ex19875030\myMatlabFunction.exe" ...
    link /MACHINE:X86 /DEBUG /DEBUGTYPE:cv  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib msws
### Created: H:\Examples\coder-ex19875030\myMatlabFunction.exe
### Successfully generated all binary outputs.
myMatlabFunction(3.1416) = 6.2832
```

Optional Step: Unregister the toolchain

To unregister the toolchain, enter:

```
delete ./rtwTargetInfo.m
RTW.TargetRegistry.getInstance('reset');
```

See Also

More About

- “Add Custom Toolchains to MATLAB® Coder™ Build Process” on page 27-165

31

Deploying Generated Code

- “C Compiler Considerations for Signed Integer Overflows” on page 31-2
- “Use C Arrays in the Generated Function Interfaces” on page 31-3
- “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces” on page 31-15
- “Use a Dynamic Library in a Microsoft Visual Studio Project” on page 31-20
- “Incorporate Generated Code Using an Example Main Function” on page 31-23
- “Use an Example C Main in an Application” on page 31-25
- “Package Code for Other Development Environments” on page 31-43
- “Structure of Generated Example C/C++ Main Function” on page 31-47
- “Troubleshoot Failures in Deployed Code” on page 31-51
- “Using Dynamic Memory Allocation for an Atoms Simulation” on page 31-52
- “Register New Hardware Devices” on page 31-58
- “Deploy Generated C Code to External Hardware: Raspberry Pi Examples” on page 31-64
- “Deploy Generated Code” on page 31-71

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).

See Also

More About

- “Setting Up the C or C++ Compiler”
- Supported and Compatible Compilers

Use C Arrays in the Generated Function Interfaces

In most cases, when you generate code for a MATLAB function that accepts or returns an array, the generated C/C++ function interface contains an array. To use the generated function interfaces, learn how the generated C/C++ arrays are defined and constructed. In particular, learn to use the `emxArray` data structure that is generated to represent dynamically allocated arrays.

When you generate C/C++ code, an example main file is created that shows how to use arrays with the generated function code. You can use the example main as a template or starting point for your own application.

Implementation of Arrays in the Generated C/C++ Code

The code generator produces C/C++ array definitions that depend on the array element type and whether the array uses static or dynamic memory allocation. The two kinds of memory allocation for an array require two different implementations:

- For an array whose size is bounded within a predefined threshold, the generated C/C++ definition consists of a pointer to memory and an integer that stores the total number of array elements, the array size. The memory for this array comes from the program stack and is statically allocated.
- For an array whose size is unknown and unbounded at compile time, or whose bound exceeds a predefined threshold, the generated C/C++ definition consists of a data structure called an `emxArray`. When an `emxArray` is created, intermediate storage bounds are set based on the current array size. During program execution, as intermediate storage bounds are exceeded, the generated code appropriates additional memory space from the heap and adds it to the `emxArray` storage. The memory for this array is dynamically allocated.

By default, arrays that are bounded within a threshold size do not use dynamic allocation in the generated code. Alternatively, you can disable dynamic memory allocation and change the dynamic memory allocation threshold. See “Control Memory Allocation for Variable-Size Arrays” on page 6-4.

This table lists a few typical cases for array representation in the generated code.

Algorithm Description and Array Size	MATLAB Function	Generated C Function Interface

Place ones onto a fixed-size 1-by-500 row vector.
 	
Fixed-size, bounded within threshold.
	`function B = create_vec0 %#codegen`
	`B = zeros(1,500);`
	`j = 1;`
	`for i = 1:500`
	`    if round(rand)`
	`        B(1,j) = 1;`
	`        j = j + 1;`
	`    end`
	`end`
		`void create_vec0(double B[500])`

Push ones onto a variable-size row vector bounded at 300 elements.
 	
Variable-size, bounded within threshold.
	`function B = create_vec %#codegen`
	`B = zeros(1,0);`
	`coder.varsize('B',[1 300],[0 1]);`
	`for i = 1:500`
	`    if round(rand)`
	`        B = [1 B];`
	`    end`
	`end`
		`void create_vec(double B_data[], int B_`

Algorithm Description and Array Size	MATLAB Function	Generated C Function Interface

Push ones onto a variable-size row vector bounded at 30,000 elements.
 	
Variable-size, not bounded within threshold.
	function B = create_vec2 %#codegen
	B = zeros(1,0);
	coder.varsize('B',[1 30000],[0 1]);
	for i = 1:500
	    if round(rand)
	        B = [1 B];
	    end
	end
		void create_vec2(emxArray_real_T *B)

Create an array with size determined by an unbounded integer input. Unknown and unbounded at compile time.	function y = create_vec3(n) %#codegen y = int8(ones(1,n));	void create_vec3(int n, emxArray_int8_T

The emxArray Dynamic Data Structure Definition

In the generated C/C++ code, the emxArray data structure definition depends on the data type of the elements that it stores. The general definition takes the form:

```
struct emxArray_<name>
{
    <type> *data;
    int *size;
    int allocatedSize;
    int numDimensions;
    boolean_T canFreeData;
};
```

In the definition, <type> indicates a data type and <name> indicates a name used to identify the emxArray structure. The code generator chooses <name> based on the types defined for MEX code generation, as listed in “Mapping MATLAB Types to Types in Generated Code” on page 33-15.

As an example, consider the emxArray definition generated for the function create_vec2. The <name> is emxArray_real_T and the <type> is double.

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

Do not seek to predict the entries for <type> and <name> prior to code generation. Instead, after code generation is complete, inspect the file <*myFunction*>_types.h from the code generation report. <*myFunction*> is the name of your entry-point function.

The generated code can also define the emxArray structure by using typedef statements, as in these examples.

```
typedef struct {
  emxArray_real_T *f1;
```

```
} cell_wrap_0;

typedef struct {
  cell_wrap_0 *data;
  int *size;
  int allocatedSize;
  int numDimensions;
  boolean_T canFreeData;
} emxArray_cell_wrap_0;
```

This table describes the `emxArray` structure fields.

Field	Description
`<type> *data`	Pointer to an array of elements of type `<type>`.
`int *size`	Pointer to a size vector. The i-th element of the size vector stores the length of the i-th dimension of the array.
`int allocatedSize`	Number of memory elements allocated for the array. If the array size changes, the generated code reallocates memory based on the new size.
`int numDimensions`	Length of the size vector. The number of dimensions you can access without crossing into unallocated or unused memory.
`boolean_T canFreeData`	Boolean flag indicating how to deallocate memory. Used only by the internal `emxArray` processing routines. • `true` – The generated code deallocates memory on its own. • `false` – The program that instantiates the `emxArray` must manually deallocate the memory pointed to by `data`.

Utility Functions for Interacting with emxArray Data

To create and interact with the `emxArray` data in your C/C++ code, the code generator exports a set of C/C++ helper functions with a user-friendly API. Use these functions to ensure that you properly initialize and destroy `emxArray` data types. To use these functions, insert an include statement for the generated header file `<`*`myFunction`*`>_emxAPI.h` in your C code. `<`*`myFunction`*`>` is the name of your entry-point function. Other functions produced by the code generator that operate on `emxArray` data, defined in `<`*`myFunction`*`>_emxutil.h`, are not intended for manual use.

The example main file generated by default for `lib`, `dll`, and `exe` code includes calls to the `emxArray` API functions. The example main code initializes the `emxArray` data to generic zero values. To use actual data inputs and values, modify the example main or create your own main file. For more information on using a main function, see “Incorporate Generated Code Using an Example Main Function” on page 31-23.

This table shows the list of exported `emxArray` API functions. Some of the API functions accept the initial number of rows, columns, or dimensions for the `emxArray` data. Each dimension can grow to accommodate new data as needed.

emxArray Helper Function	Description
`emxArray_<name> *emxCreate_<name>(int rows, int cols)`	Creates a pointer to a two-dimensional `emxArray`, with data elements initialized to zero. Allocates new memory for the data.
`emxArray_<name> *emxCreateND_<name>(int numDimensions, int *size)`	Creates a pointer to an N-dimensional `emxArray`, with data elements initialized to zero. Allocates new memory for the data.
`emxArray_<name> *emxCreateWrapper_<name>(<type> *data, int rows, int cols)`	Creates a pointer to a two-dimensional `emxArray`. Uses data and memory you provide and wraps it into the `emxArray` data structure. Sets `canFreeData` to `false` to prevent inadvertent freeing of user memory.
`emxArray_<name> *emxCreateWrapperND_<name>(<type> *data, int numDimensions, int *size)`	Creates a pointer to an N-dimensional `emxArray`. Uses data and memory you provide and wraps it into the `emxArray` data structure. Sets `canFreeData` to `false` to prevent inadvertent freeing of user memory.
`void emxInitArray_<name>(emxArray_<name> **pEmxArray, int numDimensions)`	Allocates memory for a double pointer to an `emxArray`.
`void emxDestroyArray_<name>(emxArray_<name> *emxArray)`	Frees dynamic memory allocated by the `emxCreate` or `emxInitArray` functions.

The code generator exports the `emxArray` API functions only for arrays that are entry-point function arguments or that are used by functions called by `coder.ceval`.

Examples

Use the Function Interface for a Statically Allocated Array

Consider the MATLAB function `myuniquetol` from “Generate Code for Variable-Size Data” on page 27-98.

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

Generate code for myuniquetol. Use coder.typeof to specify the input types as a bounded, variable-size array and a scalar double.

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

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 whose second dimension can vary up to 100 elements. Because the maximum size of array B is bounded within the default threshold size, the code generator uses static memory allocation for the array.

The generated function interface is:

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

The function interface declares the input argument A and the output argument B. A_size contains the size of A. After the call to myuniquetol, B_size contains the size of B.

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.

This 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);
}
```

Create an emxArray by Using the emxCreate or emxInitArray Functions

The emxCreate and emxCreateND API functions create an emxArray, allocating new memory from the heap as needed. You can then use the emxArray as an input to or output from the generated code. This C code example shows how to use emxCreate. Assume that you have already generated source code for a function myFunction that uses the data type emxArray_uint32_T.

```
#include <stdio.h>
#include <stdlib.h>
#include "myFunction_emxAPI.h"
#include "myFunction.h"

int main(int argc, char *argv[])
{
    /* Create a 10-by-10 uint32_T emxArray */
    emxArray_uint32_T *pEmx = emxCreate_uint32_T(10,10);

    /* Initialize the emxArray memory, if needed */
    int k = 0;
    for (k = 0; k < 100; ++k) {
```

```
        pEmx->data[k] = (uint32_T) k;
    }

    /* Use pEmx array here; */
    /* Insert call to myFunction */

    /* Deallocate any memory allocated in pEmx */
    /* This DOES free pEmx->data */
    emxDestroyArray_uint32_T(pEmx);

    /* Unused */
    (void)argc;
    (void)argv;

    return 0;
}
```

In this example, you know the initial size of the emxArray. If you do not know the size of the array, as when you use the array to store output, you can enter the value 0 for the rows and cols fields. For example, if you do not know the number of columns, you can write:

```
emxArray_uint32_T *pEmx = emxCreate_uint32_T(10,0);
```

The data structure grows to accommodate data as needed. After your function runs, determine the output size by accessing the size and numDimensions fields.

Use the emxInitArray API function to create an array that is returned as output, for which you do not know the array size in advance. For example, to create an emxArray of two dimensions, with unknown sizes in either dimension, you can write:

```
emxArray_uint32_T *s;
emxInitArray_uint32_T(&s, 2);
```

Load Existing Data into an emxArray

The emxCreateWrapper and emxCreateWrapperND API functions enable you to load or wrap existing memory and data into an emxArray to pass the data to a generated function. This C code example shows how to use emxCreateWrapper. Assume that you have already generated source code for a function myFunction that uses the data type emxArray_uint32_T.

```
#include <stdio.h>
#include <stdlib.h>
#include "myFunction_emxAPI.h"
#include "myFunction.h"

int main(int argc, char *argv[])
{
    /* Create a 10-by-10 C array of uint32_T values */
    uint32_T x[100];
    int k = 0;
    emxArray_uint32_T *pEmx = NULL;
    for (k = 0; k < 100; k++) {
        x[k] = (uint32_T) k;
    }

    /* Load existing data into an emxArray */
    pEmx = emxCreateWrapper_uint32_T(x,10,10);
```

```
    /* Use pEmx here; */
    /* Insert call to myFunction */

    /* Deallocate any memory allocated in pEmx */
    /* This DOES NOT free pEmx->data because the wrapper function was used */
    emxDestroyArray_uint32_T(pEmx);

    /* Unused */
    (void)argc;
    (void)argv;

    return 0;
}
```

Create and Use Nested emxArray Data

This example shows how to work with generated code that contains `emxArray` data nested inside of other `emxArray` data. To use the generated code, in your main function or calling function, initialize the `emxArray` data from the bottom nodes up.

MATLAB Algorithm

This MATLAB algorithm iterates through an array of structures called `myarray`. Each structure contains a lower-level array of values. The algorithm sorts and sum the elements of the lower-level array for each `struct`.

```
% y is an array of structures of the form
% struct('values', [...], 'sorted', [...], 'sum', ... )
function y = processNestedArrays(y) %#codegen
coder.cstructname(y, 'myarray');
for i = 1:numel(y)
    y(i).sorted = sort(y(i).values);
    y(i).sum = sum(y(i).values);
end
```

Generate MEX Function for Testing

As a first step, to be able to test the algorithm, generate a MEX function. Use the `coder.typeof` function to manually specify the input as an unbounded, variable-size row vector of `structs`, which themselves contain unbounded, variable-size row vectors.

```
myarray = coder.typeof( ...
            struct('values', coder.typeof(0, [1 inf]), ...
                   'sorted', coder.typeof(0, [1 inf]), ...
                   'sum', coder.typeof(0))                , [1 inf]);
codegen -args {myarray} processNestedArrays
```

```
Code generation successful.
```

Inspect the Generated Function Interfaces

The MEX function source code contains specialized code that enables it to interface with the MATLAB runtime environment, which makes it more complex to read. To produce more simplified source code, generate library code.

```
codegen -config:lib -args {myarray} processNestedArrays -report
```

```
Code generation successful: To view the report, open('codegen\lib\processNestedArrays\html\report
```

Inspect the generated function code `processNestedArrays.c` from the code generation report. The generated example main file `main.c` shows how to call the generated function code by creating and initializing inputs with the `emxCreate` API function.

Write and Use Your Own Customized Main File to Initialize emxArray Data

Although the generated example main shows how to invoke the generated function code, it does not contain information on desired input values. Using the example main as a guide, write your own main file. Use the coding style and preferences of your choice. Specify the values of your inputs and insert pre and post-processing code as needed.

The file `processNestedArrays_main.c` shows an example. This main file uses the `emxArray` API functions to create and initialize the structure data. For both the generated example main file and this hand written main file, the code initializes the `emxArray` data at the bottom (leaf) nodes, and assigns that data to the nodes above.

```
type processNestedArrays_main.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include "processNestedArrays_emxAPI.h"
#include "processNestedArrays.h"

static void print_vector(emxArray_real_T *v)
{
    int i;
    printf("[");
    for (i = 0; i < v->size[1]; i++) {
        if (i > 0) printf(" ");
        printf("%.0f", v->data[i]);
    }
    printf("] \n");
}

int main(int argc, char *argv[])
{
    int i;
    static double values_1[] = { 5, 3, 4, 1, 2, 6 };
    static double values_2[] = { 50, 30, 40, 10, 20, 60 };
    static double values_3[] = { 42, 4711, 1234 };
    static double * values[] = { values_1, values_2, values_3 };
    static int values_len[] = { 6, 6, 3 };

    /* Setup myarray emxArrays */
    emxArray_myarray *myarr = emxCreate_myarray(1, 3); /* Create outer array */
    for (i = 0; i < 3; i++) {
        /* Setup field 'values'. Don't allocate memory; reuse the data pointer. */
        myarr->data[i].values = emxCreateWrapper_real_T(values[i], 1, values_len[i]);
        /* Initialize the 'sorted' field to the empty vector. */
        myarr->data[i].sorted = emxCreate_real_T(1, 0);
        /* Initiailize the 'sum' field. */
        myarr->data[i].sum = 0;
```

```
    }

    /* Call process function */
    processNestedArrays(myarr);

    /* Print result */
    for (i = 0; i < myarr->size[1]; i++) {
        printf("    values: "); print_vector(myarr->data[i].values);
        printf("    sorted: "); print_vector(myarr->data[i].sorted);
        printf("       sum: %.0f \n\n", myarr->data[i].sum);
    }

    /* Cleanup memory */
    emxDestroyArray_myarray(myarr);

    /* Unused */
    (void)argc;
    (void)argv;

    return 0;
}
```

Generate an Executable and Compare Results with MEX Function

Using the provided main file, you can generate a standalone executable for the algorithm.

```
codegen -config:exe -args {myarray} processNestedArrays ...
    processNestedArrays_main.c -report
```

```
Code generation successful: To view the report, open('codegen\exe\processNestedArrays\html\report
```

Declare input data for the MEX function that matches the input for the standalone executable, defined in `processNestedArrays_main.c`.

```
myarray = [struct('values', [5 3 4 1 2 6], 'sorted', zeros(1,0), 'sum', 0), ...
           struct('values', [50 30 40 10 20 60], 'sorted', zeros(1,0), 'sum', 0), ...
           struct('values', [42 4711 1234], 'sorted', zeros(1,0), 'sum', 0)];
```

Compare the MEX function results with the standalone executable results.

```
fprintf('.mex output \n----------- \n');
r = processNestedArrays_mex(myarray);
disp(r(1));
disp(r(2));
disp(r(3));

fprintf('.exe output \n----------- \n');
system('processNestedArrays');
```

```
.mex output
-----------
    values: [5 3 4 1 2 6]
    sorted: [1 2 3 4 5 6]
       sum: 21

    values: [50 30 40 10 20 60]
    sorted: [10 20 30 40 50 60]
```

```
       sum: 210

    values: [42 4711 1234]
    sorted: [42 1234 4711]
       sum: 5987

.exe output
-----------
    values: [5 3 4 1 2 6]
    sorted: [1 2 3 4 5 6]
       sum: 21

    values: [50 30 40 10 20 60]
    sorted: [10 20 30 40 50 60]
       sum: 210

    values: [42 4711 1234]
    sorted: [42 1234 4711]
       sum: 5987
```

The output results are identical.

Use emxArray_char_T Data with String Inputs

In this example, a MATLAB function changes the size of a character vector at run time. Because the final length of the vector can vary, the generated C code instantiates the vector as a dynamically sized `emxArray`. This example shows how to write a main function that uses `emxArray_char_T` with the generated function interface. Use this example as a guide for working with the `emxArray_char_T` data type.

MATLAB Algorithm

The function `replaceCats` takes a character vector as input and replaces all instances of the word 'cat' or 'Cat' with 'velociraptor' and 'Velociraptor'. Because the code generator cannot determine the output length at compile time, the generated code uses the `emxArray` data type.

```
function cstrNew = replaceCats(cstr)
%#codegen
cstrNew = replace(cstr,'cat','velociraptor');
cstrNew = replace(cstrNew,'Cat','Velociraptor');
```

Generate Source Code

To generate code for `replaceCats`, specify the input type to the function as a variable-size character array.

```
t = coder.typeof('a',[1 inf]);
codegen replaceCats -args {t} -report -config:lib
```

```
Code generation successful: To view the report, open('codegen\lib\replaceCats\html\report.mldatx
```

In the generated code, the example main file `/codegen/lib/replaceCats/examples/main.c` provides a template for writing your own main function.

Create a Main Function from the Template

Modify the main function to take character input from the command line. Use the `emxCreate` and `emxCreateWrapper` API functions to initialize your emxArray data. After you have finished writing your main source file and header file, place the modified files in the root folder.

```
type main_replaceCats.c
```

```
#include "main_replaceCats.h"
#include "replaceCats.h"
#include "replaceCats_terminate.h"
#include "replaceCats_emxAPI.h"
#include "replaceCats_initialize.h"
#include <string.h>
#include <stdio.h>

#define MAX_STRING_SZ 512

static void main_replaceCats(char *inStr)
{
  /* Create emxArray's & other variables */
  emxArray_char_T *cstr = NULL;
  emxArray_char_T *cstrFinal = NULL;
  char outStr[MAX_STRING_SZ];
  int initCols = (int) strlen(inStr);
  int finCols;

  /* Initialize input & output emxArrays */
  cstr = emxCreateWrapper_char_T(inStr, 1, initCols);
  cstrFinal = emxCreate_char_T(1, 0);

  /* Call generated code on emxArrays */
  replaceCats(cstr, cstrFinal);

  /* Write output string data with null termination */
  finCols = cstrFinal->size[0]*cstrFinal->size[1];
  if (finCols >= MAX_STRING_SZ) {
      printf("Error: Output string exceeds max size.");
      exit(-1);
  }
  memcpy(outStr, cstrFinal->data, finCols);
  outStr[finCols]=0;

  /* Print output */
  printf("\nOld C string: %s \n", inStr);
  printf(  "New C string: %s \n", outStr);

  /* Free the emxArray memory */
  emxDestroyArray_char_T(cstrFinal);
}

int main(int argc, char *argv[])
{
  if (argc != 2 ) {
      printf("Error: Must provide exactly one input string, e.g.\n");
      printf(">replaceCats \"hello cat\"\n");
      exit(-1);
```

```
  }

  replaceCats_initialize();
  main_replaceCats(argv[1]);
  replaceCats_terminate();

  return 0;
}
```

Generate Executable File

Generate executable code:

```
t = coder.typeof('a',[1 inf]);
codegen replaceCats -args {t} -config:exe main_replaceCats.c
```

```
Code generation successful.
```

Test the executable on your platform and modify your main file as needed. For example, on Windows, you get the output:

```
C:\>replaceCats.exe "The pet owner called themselves a 'Catdad'"

Old C string: The pet owner called themselves a 'Catdad'

New C string: The pet owner called themselves a 'Velociraptordad'
```

See Also

`coder.typeof` | `coder.varsize`

More About

- “Using Dynamic Memory Allocation for an Atoms Simulation” on page 31-52
- “Generate Code for Variable-Size Data” on page 27-98
- “Multidimensional Arrays”

Use Dynamically Allocated C++ Arrays in Generated Function Interfaces

In most cases, when you generate code for a MATLAB function that accepts or returns an array, there is an array at the interface of the generated C/C++ function. For an array size that is unknown at compile time, or whose bound exceeds a predefined threshold, the memory for the generated array is dynamically allocated on the heap. Otherwise, the memory of the generated array is statically allocated on the stack. See “Control Memory Allocation for Variable-Size Arrays” on page 6-4.

If you choose C++ as your target language for code generation, by default, the dynamically allocated array is implemented as a class template called `coder::array` in the generated code. To use dynamically allocated arrays in your custom C++ code that you integrate with the generated C++ functions, learn to use the `coder::array` template.

By default, the generated C++ code uses the `coder::array` template to implement dynamically allocated arrays. Instead, you can choose to generate C++ code that uses the C style `emxArray` data structure to implement dynamically allocated arrays. To generate C style `emxArray` data structures, do one of the following:

- In a code configuration object (`coder.MexCodeConfig`, `coder.CodeConfig`, or `coder.EmbeddedCodeConfig`), set the `DynamicMemoryAllocationInterface` parameter to `'C'`.
- In the MATLAB Coder app, on the **Memory** tab, set **Dynamic memory allocation interface** to `Use C style EmxArray`.

To learn more about statically allocated arrays or dynamically allocated arrays implemented by using the C style `emxArray` data structure, see “Use C Arrays in the Generated Function Interfaces” on page 31-3.

Examples of C++ Function Interfaces That Use Dynamically Allocated Arrays

This table lists two typical cases for dynamic array representation in the generated C++ code. For the definition of the `coder::array` template that implements the dynamic arrays in the generated code, see “Using the coder::array Class Template” on page 31-16.

Algorithm Description and Array Size	MATLAB Function	Generated C++ Function Interface

Push ones onto a variable-size row vector bounded at 30,000 elements.
 	
Variable-size, not bounded within threshold.
	`function B = create_vec2 %#codegen`
	`B = zeros(1,0);`
	`coder.varsize('B',[1 30000],[0 1]);`
	`for i = 1:500`
	`    if round(rand)`
	`        B = [1 B];`
	`    end`
	`end`
		`void create_vec2(coder::array<double, 2`

Algorithm Description and Array Size	MATLAB Function	Generated C++ Function Interface
Create an array with size determined by an unbounded integer input. Unknown at compile time.	`function y = create_vec3(n) %#codegen` `y = int8(ones(1,n));`	`void create_vec3(double n, coder::array`

Using the coder::array Class Template

When you generate C++ code for your MATLAB functions, the code generator produces a header file `coder_array.h` in the build folder. This header file contains the definition of the class template `array` in the namespace `coder`. The `coder::array` template implements the dynamically allocated arrays in the generated code. The declaration for this template is:

```
template <typename T, int32_T N> class array
```

The array contains elements of type `T` and has `N` dimensions. For example, to declare a two-dimensional dynamic array `myArray` that contains elements of type `int32_T` in your custom C++ code, use:

```
coder::array<int32_T, 2> myArray
```

To use dynamically allocated arrays in your custom C++ code that you want to integrate with the generated code (for example, a custom main function), include the `coder_array.h` header file in your custom `.cpp` files. This table shows the API you use to create and interact with dynamic arrays in your custom C++ code.

Action	Instructions
Declare a dynamic array `myArray` that contains elements of type `int32_T`. Set the number of dimensions of `myArray` to `2`.	Use the `coder::array` template. Specify element type and number of dimensions. `coder::array<int32_T, 2> myArray`
Allocate memory for `myArray`. Set the size of the first dimension to `1` and the second dimension to `100`.	Use the `set_size` method. `myArray.set_size(1, 100)` If the dimension of `myArray` changes later on during execution, the generated code reallocates memory based on the new size.
Access the size vector of `myArray`.	Access the `size` array, which is a data member of `myArray`. For example, to access the size of the second dimension of `myArray`, use: `myArray.size(1)`
Index into the dynamic array `myArray`.	Use the standard C++ syntax for array indexing. For example, to set the `i`-th element of `myArray` equal to `i`, use: `myArray[i] = i`

Examples

Generate C++ Code That Accepts and Returns a Variable-Size Numeric Array

Define a MATLAB function `xTest1` that accepts an array `X`, adds the scalar `A` to each of its elements, and returns the resulting array `Y`.

```
function Y = xTest1(X, A)
Y = X;
for i = 1:numel(X)
    Y(i) = X(i) + A;
end
```

You goal is to generate a C++ executable for `xTest1` that can accept and return an array of `int32_T` elements. You want the first dimension of the array to be singleton and the second dimension to be unbounded.

Define a C++ main function in the file `xTest1_main.cpp` in your current working folder.

```
#include<iostream>
#include<coder_array.h>
#include<xTest1.h>

int main(int argc, char *argv[])
{
    static_cast<void>(argc);
    static_cast<void>(argv);

    coder::array<int32_T, 2> myArray;
    myArray.set_size(1, 100);
    for (int i = 0; i < myArray.size(1); i++) {
        myArray[i] = i;
    }

    coder::array<int32_T, 2> myResult;
    xTest1(myArray, 1000, myResult);

    for (int i = 0; i < myResult.size(1); i++) {
        if (i > 0) std::cout << " ";
        std::cout << myResult[i];
        if (((i+1) % 10) == 0) std::cout << std::endl;
    }
    std::cout << std::endl;

    return 0;
}
```

This main function includes the header file `coder_array.h` that contains the `coder::array` class template definition. The main function uses the API described in the table in the previous section to perform these actions:

- Declare `myArray` and `myResult` as two-dimensional dynamic arrays of `int32_T` elements.
- Dynamically set the sizes of the two dimensions of `myArray` to `1` and `100` by using the `set_size` method.
- Access the size vector of `myResult` by using `myResult.size`.

Generate code by running this script. Replace `'C:\work'` with the path to your current working folder.

```
cfg = coder.config('exe'); cfg.TargetLang = 'C++';
cfg.CustomSource = 'xTest1_main.cpp';
cfg.CustomInclude = 'C:\work';
codegen -config cfg -args { coder.typeof(int32(0), [1 inf]), int32(0)} xTest1_main.cpp xTest1.m
```

The code generator produces an executable file `xTest1.exe` in your current working folder.

Generate C++ Code That Accepts and Returns a Variable-Size Vector of Characters

Define a MATLAB function `xStringTest` that accepts a character vector `str`, inserts `str` between the character vectors `'hello '` and `' world!'`, and returns the result. Your goal is to generate a C ++ executable from `xStringTest`.

```
function y = xStringTest(str)
assert(isa(str, 'char'));
assert(size(str,1) == 1);
assert(size(str,2) >= 0);
y = ['hello ' str ' world!'];
```

Define a C++ main function in the file `xStringTest_main.cpp` in your current working folder. This main function uses `std::vector` to declare the vector `vec` of `char_T` elements that you pass to the generated C++ function `xStringTest`.

```
#include<iostream>
#include<coder_array.h>
#include<xTest1.h>

int main(int, char *[])
{
    coder::array<char_T, 2> result;

    std::vector<char_T> vec;
    vec.resize(10);
    for (size_t i = 0; i < 10; i++) {
        vec[i] = static_cast<char_T>('A' + i);
    }

    xStringTest(vec, result);

    std::cout << "Result is " << static_cast<std::string>(result) << std::endl;

    return 0;
}
```

Generate code by running this script. Replace `'C:\work'` with your current working folder.

```
cfg = coder.config('exe'); cfg.TargetLang = 'C++';
cfg.CustomSource = 'xStringTest_main.cpp';
cfg.CustomInclude = 'C:\work';
codegen -config cfg -args {coder.typeof(char('X'), [1 inf])} xStringTest_main.cpp xStringTest.m
```

The code generator produces an executable file `xStringTest.exe` in your current working folder.

See Also

`coder.typeof` | `coder.varsize`

More About

- “Use C Arrays in the Generated Function Interfaces” on page 31-3
- “Control Memory Allocation for Variable-Size Arrays” on page 6-4

Use a Dynamic Library in a Microsoft Visual Studio Project

This example shows how to create and configure a simple Microsoft Visual Studio project that calls a dynamic library (DLL) generated by MATLAB Coder. The example uses Microsoft Visual Studio 2017. 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. Use the same version of the same compiler to generate your DLL that you use to build your Microsoft Visual Studio project. Otherwise, you can encounter linking errors.

   For this example, use the Microsoft Visual Studio 2017 compiler. To select the compiler that the code generator uses, enter `mex -setup` at the command line. For more information, see Supported and Compatible Compilers.
4. Generate a DLL for the MATLAB function `foo`. The `-args` option specifies that the input `a` is a real double.

```
codegen -config:dll foo -args {0} -report
```

   On Microsoft Windows systems, `codegen` generates a C dynamic library, `foo.dll`, and supporting files in the default folder, `C:\dll_test\codegen\dll\foo`.

Create a Microsoft Visual Studio Project

In Microsoft Visual Studio, create an Empty Project:

1. Select **File** > **New** > **Project**.
2. Select **Installed** > **Visual C++** > **General** and select **Empty project**. Enter a project name.
3. Click **OK**.

Create a main.c File That Uses the Library

Write a `main.c` file that uses `foo.dll`. The `main.c` function must:

- Include the generated header files, which contain the function prototypes for the library functions.
- Call the terminate function after calling the library function for the last time.

By default, the code generator includes a call to the initialize function at the beginning of the generated C/C++ entry-point functions. So, you do not need to call the initialize function from `main.c`. See "Use Generated Initialize and Terminate Functions" on page 27-25.

To create the file:

1. From the **Solution Explorer**, right-click the **Source Files** folder and select **Add** > **New Item**
2. Select **C++ File (.cpp)**. In the **Name** field, enter `main.c`.

3. Click **Add**.
4. Enter the code:

```
#include "foo.h"
#include "foo_terminate.h"
#include <stdio.h>

int main()
{
  printf("%f\n", foo(26));
  foo_terminate();
  getchar();
  return 0;
}
```

Configure the Platform

MATLAB Coder automatically uses a toolchain configured to build a 64-bit DLL. By default, Microsoft Visual Studio is configured to build for the Win32 platform. You must change the build platform to x64 to match the generated 64-bit DLL. In Microsoft Visual Studio:

1. Select **Build** > **Configuration Manager**.
2. Set **Active solution platform** to **x64**.

If you want to build a 32-bit DLL on a 64-bit platform, you must use a 32-bit toolchain definition. See “Build 32-bit DLL on 64-bit Windows® Platform Using MSVC Toolchain” on page 30-21.

Specify External Dependencies

To build your project, the compiler requires the associated header files. The linker requires the generated `.lib` files.

1. Highlight your project in the **Solution Explorer**, and then select **Project** > **Properties**.
2. The code generator produces types in the file `rtwtypes.h`, which includes the file `tmwtypes.h`. This file is stored in *`matlabroot`*`\extern\include`, where `matlabroot` is the root directory of the MATLAB installation. To return the root directory, enter `matlabroot` in the Command Window.

   Under **Configuration Properties** > **C/C++** > **General**, add the folders `C:\dll_test\codegen\dll\foo` and *`matlabroot`*`\extern\include` to **Additional Include Directories**. Separate the entries with a semicolon.
3. Under **Configuration Properties** > **Linker** > **Input**, add `foo.lib` to **Additional Dependencies**.
4. Under **Configuration Properties** > **Linker** > **General**, add the folder `C:\dll_test\codegen\dll\foo` to **Additional Library Directories**.

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. Verify that the output appears as you expect.

See Also

More About

- "Build 32-bit DLL on 64-bit Windows® Platform Using MSVC Toolchain" on page 30-21

Incorporate Generated Code Using an Example Main Function

In this section...

“Workflow for Using an Example Main Function” on page 31-23
“Control Example Main Generation Using the MATLAB Coder App” on page 31-23
“Control Example Main Generation Using the Command-Line Interface” on page 31-24

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 31-25.

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';
```

See Also

More About

- “Structure of Generated Example C/C++ Main Function” on page 31-47
- “Specifying main Functions for C/C++ Executables” on page 27-11

Use an Example C Main in an Application

This example shows how to build a C executable from MATLAB code that implements a simple Sobel filter to perform edge detection on images. The executable reads an image from the disk, applies the Sobel filtering algorithm, and then saves the modified image.

The example shows how to generate and modify an example main function that you can use when you build the executable.

In this section...

"Prerequisites" on page 31-25
"Create a Folder and Copy Relevant Files" on page 31-25
"Run the Sobel Filter on the Image" on page 31-27
"Generate and Test a MEX Function" on page 31-29
"Generate an Example Main Function for sobel.m" on page 31-29
"Copy the Example Main Files" on page 31-32
"Modify the Generated Example Main Function" on page 31-32
"Generate the Sobel Filter Application" on page 31-41
"Run the Sobel Filter Application" on page 31-41
"Display the Resulting Image" on page 31-42

Prerequisites

To complete this example, install the following products:

- MATLAB
- MATLAB Coder
- C compiler (for most platforms, a default C compiler is supplied with MATLAB). For a list of supported compilers, see `https://www.mathworks.com/support/compilers/current_release/`.

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

Create a Folder and Copy Relevant Files

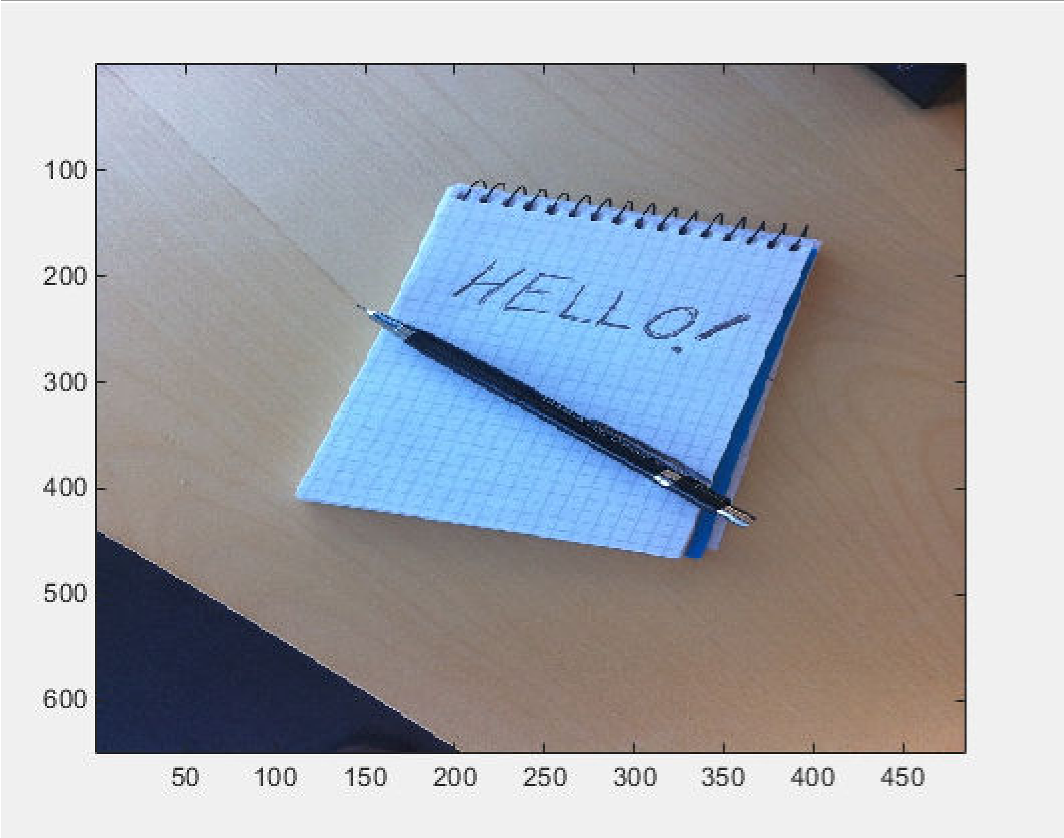
The files you use in this example are:

File Name	File Type	Description
sobel.m	Function code	MATLAB implementation of a Sobel filtering algorithm. 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'))
```

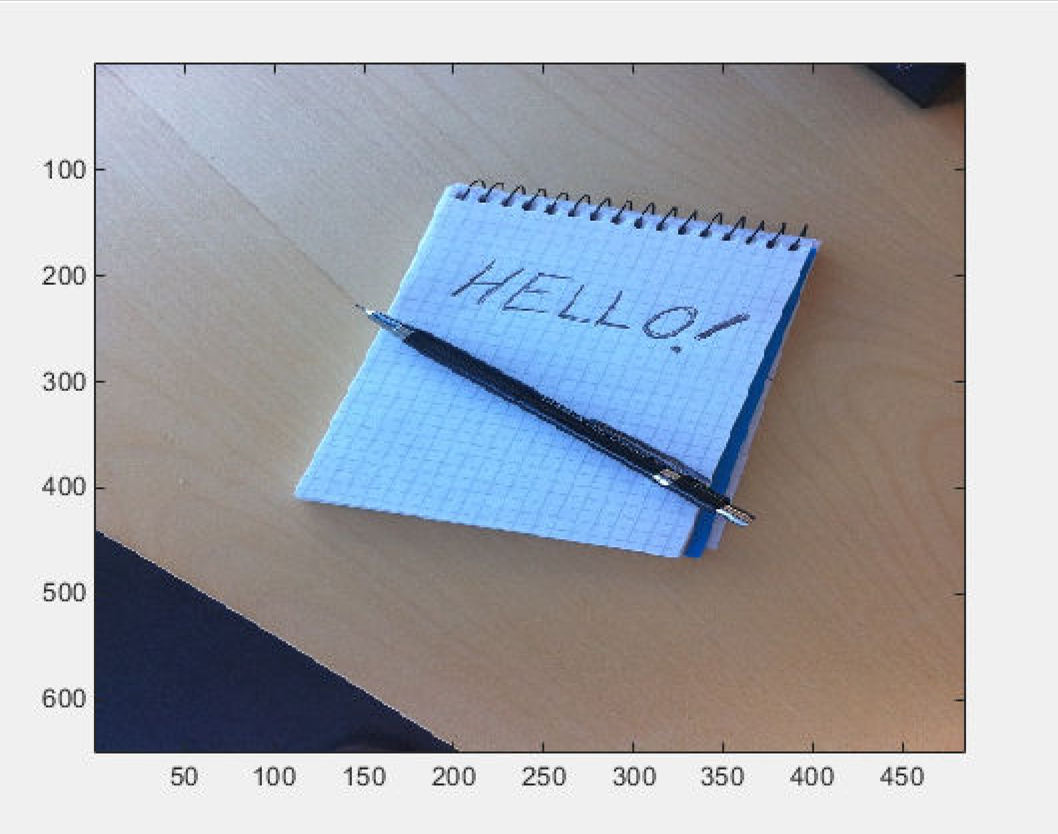
Run the Sobel Filter on the Image

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).

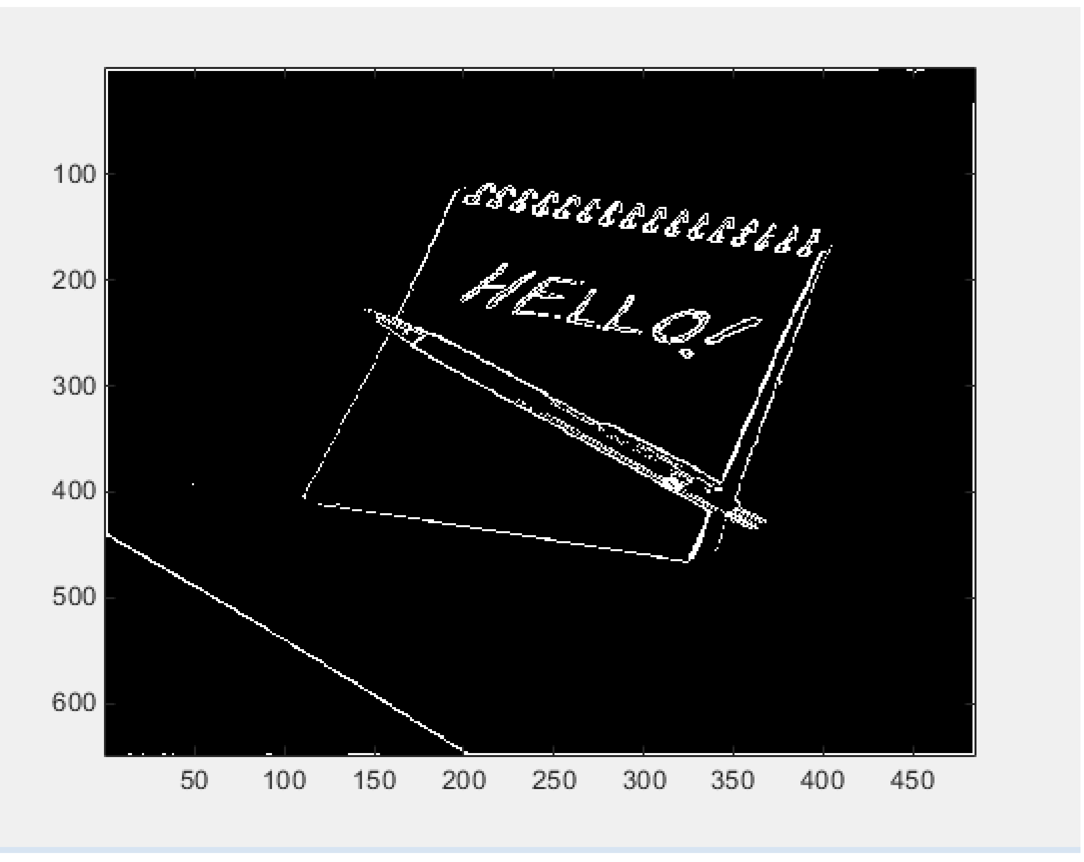
   ```
   gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
   ```

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 "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 31-32
- "Modify the Initialization Function argInit_d1024xd1024_real_T" on page 31-34
- "Write the Function saveImage" on page 31-35
- "Modify the Function main_sobel" on page 31-37
- "Modify the Function Declarations" on page 31-38
- "Modify the Include Files" on page 31-38
- "Contents of Modified File main.c" on page 31-38

The example main function declares and initializes data, including dynamically allocated data, to zero values. It calls entry-point functions with arguments set to zero values, but it does not use values returned from the entry-point functions.

The C main function must meet the requirements of your application. This example modifies the example main function to meet the requirements of the Sobel filter application.

This example modifies the file `main.c` so that the Sobel filter application:

- Reads in the grayscale image from a binary file.
- Applies the Sobel filtering algorithm.
- Saves the modified image to a binary file.

Modify the Function main

Modify the function `main` to:

- Accept the file containing the grayscale image data and a threshold value as input arguments.
- Call the function `main_sobel` with the address of the grayscale image data stream and the threshold value as input arguments.

In the function `main`:

1. Remove the declarations `void(argc)` and `(void)argv`.
2. Declare the variable `filename` to hold the name of the binary file containing the grayscale image data.

   ```
   const char *filename;
   ```
3. Declare the variable `threshold` to hold the threshold value.

   ```
   double threshold;
   ```
4. Declare the variable `fd` to hold the address of the grayscale image data that the application reads in from `filename`.

   ```
   FILE *fd;
   ```
5. Add an `if` statement that checks for three arguments.

   ```
   if (argc != 3) {
         printf("Expected 2 arguments: filename and threshold\n");
         exit(-1);
   }
   ```
6. Assign the input argument `argv[1]` for the file containing the grayscale image data to `filename`.

   ```
   filename = argv[1];
   ```
7. Assign the input argument `argv[2]` for the threshold value to `threshold`, converting the input from a string to a numeric double.

   ```
   threshold = atof(argv[2]);
   ```
8. Open the file containing the grayscale image data whose name is specified in `filename`. Assign the address of the data stream to `fd`.

   ```
   fd = fopen(filename, "rb");
   ```
9. To verify that the executable can open `filename`, write an `if`-statement that exits the program if the value of `fd` is `NULL`.

   ```
   if (fd == NULL) {
      exit(-1);
   }
   ```
10. Replace the function call for `main_sobel` by calling `main_sobel` with input arguments `fd` and `threshold`.

    ```
    main_sobel(fd, threshold);
    ```
11. Close the grayscale image file after calling `sobel_terminate`.

    ```
    fclose(fd);
    ```

Modified Function main

```
int main(int argc, const char * const argv[])
{
```

```
  const char *filename;
  double threshold;
  FILE *fd;

  if (argc != 3) {
      printf("Expected 2 arguments: filename and threshold\n");
      exit(-1);
  }

  filename = argv[1];
  threshold = atof(argv[2]);
  fd = fopen(filename, "rb");
  if (fd == NULL) {
    exit(-1);
  }
  /* Initialize the application.
     You do not need to do this more than one time. */
  sobel_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_sobel(fd, threshold);

  /* Terminate the application.
     You do not need to do this more than one time. */
  sobel_terminate();

  fclose(fd);

  return 0;
}
```

Modify the Initialization Function argInit_d1024xd1024_real_T

In the example main file, the function `argInit_d1024xd1024_real_T` creates a dynamically allocated variable-size array (emxArray) for the image that you pass to the Sobel filter. This function initializes the emxArray to a default size and the elements of the emxArray to 0. It returns the initialized emxArray.

For the Sobel filter application, modify the function to read the grayscale image data from a binary file into the emxArray.

In the function `argInit_d1024xd1024_real_T`:

1. Replace the input argument `void` with the argument `FILE *fd`. This variable points to the grayscale image data that the function reads in.

   ```
   static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
   ```
2. Change the values of the variable `iv2` to match the dimensions of the grayscale image matrix `gray`. `iv2` holds the size values for the dimensions of the emxArray that `argInit_d1024xd1024_real_T` creates.

   ```
   static int iv2[2] = { 484, 648 };
   ```

   MATLAB stores matrix data in column-major format, while C stores matrix data in row-major format. Declare the dimensions accordingly.

3 Define a variable `element` to hold the values read in from the grayscale image data.

```
double element;
```

4 Change the `for`-loop construct to read data points from the normalized image into `element` by adding an `fread` command to the inner `for`-loop.

```
fread(&element, 1, sizeof(element), fd);
```

5 Inside the `for`-loop, assign `element` as the value set for the emxArray data.

```
result->data[b_j0 + result->size[0] * b_j1] = element;
```

Modified Initialization Function argInit_d1024xd1024_real_T

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
  emxArray_real_T *result;
  static int iv2[2] = { 484, 648 };

  int b_j0;
  int b_j1;
  double element;

  /* Set the size of the array.
     Change this size to the value that the application requires. */
  result = emxCreateND_real_T(2, iv2);

  /* Loop over the array to initialize each element. */
  for (b_j0 = 0; b_j0 < result->size[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 "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 "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.

See Also

Related Examples

- “Structure of Generated Example C/C++ Main Function” on page 31-47
- “Incorporate Generated Code Using an Example Main Function” on page 31-23

Package Code for Other Development Environments

In this section...

"When to Package Code" on page 31-43
"Package Generated Code Using the MATLAB Coder App" on page 31-43
"Package Generated Code at the Command Line" on page 31-44
"Specify packNGo Options" on page 31-45

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 31-43 and "Package Generated Code at the Command Line" on page 31-44.

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 31-23.
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 31-23.

In this example, you specify only the file name. Optionally, you can specify additional packaging options. See “Specify packNGo Options” on page 31-45.

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`.

Choose a Structure for the Zip File

Before you generate and package the files, decide whether you want to package the files in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure. This approach is the simplest and might be the optimal choice.

If	Use
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	A single, flat folder structure

If	Use
The target development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code is dependent on the relative location of files	A hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of zip files. There is a primary zip file, which in turn contains the following secondary zip files:

- `mlrFiles.zip` — files in your *`matlabroot`* folder tree
- `sDirFiles.zip` — files in and under your build folder where you initiated code generation
- `otherFiles.zip` — required files not in the *`matlabroot`* or `start` folder trees

Paths for the secondary zip files are relative to the root folder of the primary zip file, maintaining the source development folder structure.

Structure of Generated Example C/C++ Main Function

In this section...

"Contents of the File main.c or main.cpp" on page 31-47
"Contents of the File main.h" on page 31-49

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 31-47
- "Function Declarations" on page 31-47
- "Argument Initialization Functions" on page 31-47
- "Entry-Point Functions" on page 31-48
- "Main Function" on page 31-48

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 each of the entry-point functions once.
- Calls the terminate function `foo_terminate`, which is named for the first MATLAB 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.

By default, the example main function does not call the initialize function foo_initialize. The code generator includes a call to the initialize function at the beginning of the generated C/C++ entry-point functions. The generated code also includes checks to make sure that the initialize function is called automatically only once, even when there are multiple entry-point functions.

You can choose to not include a call to the initialize function in the generated entry-point functions. To make this choice, do one of the following:

- In a coder.CodeConfig or coder.EmbeddedCodeConfig object, set RunInitializeFcn to false.
- In the MATLAB Coder app, on the **All Settings** tab, set **Automatically run the initialize function** to No.

If you make this choice, the example main function includes a call to the initialize function foo_initialize.

See "Use Generated Initialize and Terminate Functions" on page 27-25.

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 31-49
- "Include Files" on page 31-49
- "Function Declarations" on page 31-49

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.

See Also

Related Examples

- “Incorporate Generated Code Using an Example Main Function” on page 31-23
- “Use an Example C Main in an Application” on page 31-25

More About

- “Mapping MATLAB Types to Types in Generated Code” on page 33-15
- “Use Generated Initialize and Terminate Functions” on page 27-25

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 “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20 and “Example: Generate Standalone C Code That Detects and Reports Run-Time Errors” on page 28-24.

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.

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
```

```
Code generation successful.
```

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
```

```
Code generation successful.
```

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.c           run_atoms_emxutil.obj
..                         rt_nonfinite.h           run_atoms_initialize.c
.gitignore                 rt_nonfinite.obj         run_atoms_initialize.h
_clang-format              rtw_proj.tmw             run_atoms_initialize.obj
buildInfo.mat              rtwtypes.h               run_atoms_rtw.bat
codeInfo.mat               run_atoms.c              run_atoms_rtw.mk
codedescriptor.dmr         run_atoms.h              run_atoms_rtw.rsp
compileInfo.mat            run_atoms.lib            run_atoms_rtw_comp.rsp
defines.txt                run_atoms.obj            run_atoms_rtw_ref.rsp
examples                   run_atoms_data.c         run_atoms_terminate.c
interface                  run_atoms_data.h         run_atoms_terminate.h
rtGetInf.c                 run_atoms_data.obj       run_atoms_terminate.obj
rtGetInf.h                 run_atoms_emxAPI.c       run_atoms_types.h
rtGetInf.obj               run_atoms_emxAPI.h       setup_msvc.bat
rtGetNaN.c                 run_atoms_emxAPI.obj
rtGetNaN.h                 run_atoms_emxutil.c
rtGetNaN.obj               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)} -config
```

```
Code generation successful.
```

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

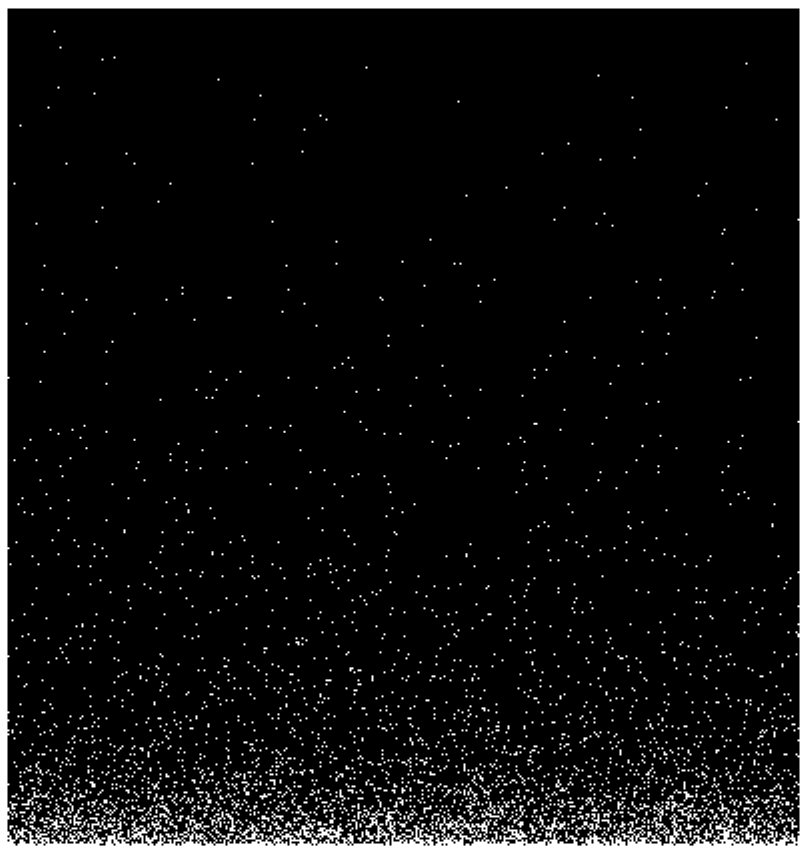
Running the executable produced `atoms_state.txt`. Now, recreate the structure array from the saved matrix:

```
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);
```



Register New Hardware Devices

On the **Hardware** tab of the MATLAB Coder app, you can specify parameters that describe target hardware and compiler properties for MATLAB software, which enables you to:

- Generate optimized code for production or test hardware.
- Directly test or deploy generated code on target hardware.

The **Hardware** tab supports a range of target hardware. To extend the range, register new hardware devices by using the `target.Processor` and `target.LanguageImplementation` classes.

Specify Hardware Implementation for New Device

To register a new hardware device:

1. Create a `target.Processor` object for the new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');
```

2. Create a `target.LanguageImplementation` object for language implementation details.

```
myLanguageImplementation = target.create('LanguageImplementation', ...
                                         'Name', 'MyProcessorImplementation');
```

3. Specify language implementation details.

```
myLanguageImplementation.Endianess = target.Endianess.Little;

myLanguageImplementation.AtomicIntegerSize = 64;
myLanguageImplementation.AtomicFloatSize = 64;
myLanguageImplementation.WordSize = 64;

myLanguageImplementation.DataTypes.Char.Size = 8;
myLanguageImplementation.DataTypes.Short.Size = 16;
myLanguageImplementation.DataTypes.Int.Size = 32;
myLanguageImplementation.DataTypes.Long.Size = 64;
myLanguageImplementation.DataTypes.LongLong.IsSupported = true;
myLanguageImplementation.DataTypes.LongLong.Size = 64;
myLanguageImplementation.DataTypes.Float.Size = 32;
myLanguageImplementation.DataTypes.Double.Size = 64;

myLanguageImplementation.DataTypes.Pointer.Size = 32;

myLanguageImplementation.DataTypes.SizeT.Size = 64;
myLanguageImplementation.DataTypes.PtrDiffT.Size = 64;
```

4. Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myLanguageImplementation;
```

5. Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

On the **Hardware** tab, you see the new device.

Specify Hardware Implementation That Persists Over MATLAB Sessions

By default, when you add the target object to the internal database, the target data is available only for the current MATLAB session. You can specify target data persistence over MATLAB sessions.

1 Create a `target.Processor` object for a new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');

existingImplementation = target.get('LanguageImplementation', ...
                                    'ARM Compatible-ARM Cortex');
myProc.LanguageImplementations = existingImplementation;
```

2 Add the `target.Processor` object to an internal database and specify persistence of target data over MATLAB sessions.

```
objectsAdded = target.add(myProc, 'UserInstall', true);
```

3 You can remove the object from the internal database.

```
target.remove(objectsAdded);
```

Create Hardware Implementation by Modifying Existing Implementation

If an existing hardware implementation contains most of the values that you want in a new hardware implementation, you can quickly create the new implementation by creating and modifying a copy of the existing implementation.

1 Create a `target.Processor` object for the new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');
```

2 Create a `target.LanguageImplementation` object that copies an existing language implementation.

```
myCopiedImplementation = target.create('LanguageImplementation', ...
                                       'Name', 'MyCopiedImplementation', ...
                                       'Copy', 'Atmel-AVR');
```

3 Specify the required language implementation details. For example, byte ordering.

```
myCopiedImplementation.Endianess = target.Endianess.Big;
```

4 Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myCopiedImplementation;
```

5 Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

Create Hardware Implementation by Reusing Existing Implementation

If your hardware device requires the same hardware implementation as an existing implementation, you can reuse the existing implementation.

1 Create a `target.Processor` object for the new hardware device.

```
myProc = target.create('Processor', ...
                       'Name', 'MyProcessor', ...
                       'Manufacturer', 'MyManufacturer');
```

2 Retrieve the existing implementation by using the identifier for the device vendor and type, for example, `'ARM Compatible-ARM Cortex'`.

```
existingImplementation = target.get('LanguageImplementation', ...
                                    'ARM Compatible-ARM Cortex');
```

3 Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = existingImplementation;
```

4 Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

Validate Hardware Device Data

To validate the data integrity of target objects, use the `IsValid` property or the `validate` method of the `target.Object` base class.

Consider an example where you create a `target.Processor` object and associate an existing language implementation with the object.

```
myProcessor = target.create('Processor');
myProcessor.LanguageImplementations = target.get('LanguageImplementation', ...
                                                 'ARM Compatible-ARM Cortex');
```

To validate the created object, run `myProcessor.IsValid` or `myProcessor.validate()`.

```
myProcessor.IsValid
```

```
ans =
  logical
  0
```

```
myProcessor.validate()
```

```
Error using target.Processor/validate
Target data validation failed.
* Undefined property "Name" in "Processor" object.
* Undefined identifier in "Processor" object.
```

The validation fails because these `target.Processor` properties are not specified:

- `Name` — Processor name
- `Id` — Object identifier

You can specify a processor name, which also specifies the object identifier.

```
myProcessor.Name = 'MyProcessor';
```

Check the validity of `myProcessor`.

```
myProcessor.IsValid
```

```
ans =
  logical
  1
```

The validity of the object is established.

Note When you use the `target.add` function to register a target object, the software also checks the validity of the object.

Export Hardware Device Data

You can share previously created hardware device data across computers and users.

For this example, specify a hardware device and add it to an internal database.

```
myProc = target.create('Processor', ...
                    'Name', 'MyProcessor', ...
                    'Manufacturer', 'MyManufacturer');
existingImplementation = target.get('LanguageImplementation', ...
                                 'ARM Compatible-ARM Cortex');
myProc.LanguageImplementations = existingImplementation;

objectsAdded = target.add(myProc);
```

To create a function for sharing the hardware device data, run:

```
target.export(myProc, 'FileName', 'exportMyProcFunction')
```

The `target.export` function creates `exportMyProcFunction.m` in the current working folder.

```
function registeredObjects = exportMyProcFunction(varargin)
% This function was generated using target data export.

    % Create target.Processor "MyManufacturer-MyProcessor"
    processor = target.create("Processor");
    processor.LanguageImplementations(1) = ...
      target.get("LanguageImplementation", "ARM Compatible-ARM Cortex");
    processor.Manufacturer = "MyManufacturer";
    processor.Name = "MyProcessor";

    % Add the target objects to MATLAB memory
    registeredObjects = target.add(processor, varargin{:});
```

Now, you can use the generated function to share the hardware device data in your database across computers and users. For example, on another computer, run this command.

```
objectsAdded = exportMyProcFunction;
```

The generated function recreates the `target.Processor` object, `MyManufacturer-MyProcessor`, and adds it to an internal database.

Create Alternative Identifier for Target Object

To create alternative identifiers for target objects, use the `target.Alias` class.

For example, if a `target.Processor` object has a long class identifier, you can create a `target.Alias` object that provides a short identifier for the `target.Processor` object.

1. Retrieve the `target.Processor` object.

   ```
   proccesorObj = target.get('Processor', ...
                             'Analog Devices-ADSP-CM40x (ARM Cortex-M)');
   ```
2. Use the `target.create` function to create a `target.Alias` object.

   ```
   aliasProcessorObj = target.create('Alias');
   ```
3. Use `target.Alias` object properties to specify the alternative identifier and original target object.

   ```
   aliasProcessorObj.Name = 'myShortName';
   aliasProcessorObj.For = proccesorObj;
   ```
4. Add the `target.Alias` object to an internal database.

   ```
   target.add(aliasProcessorObj);
   ```
5. To retrieve the original `target.Processor` object, run:

   ```
   target.get('Processor', 'myShortName');
   ```

Upgrade Data Definitions for Hardware Devices

To upgrade existing hardware device definitions that are specified through `rtwTargetInfo.m` files, use the `target.upgrade` function.

rtwTargetInfo.m File

Suppose you have the hardware device definition in an `rtwTargetInfo.m` file:

```
function rtwTargetInfo(tr)

    % Add registration function handle to the Target Registry
    tr.registerTargetInfo(@loc_register_hardware);
end

function hw = loc_register_hardware
    hw = RTW.HWDeviceRegistry;
    hw.Vendor = 'MyManufacturer';
    hw.Type = 'MyDevice';
    hw.Alias = {};
    hw.Platform = {'Prod', 'Target'};
    hw.setWordSizes([8 16 32 64 64 64 64 64 64 64 64]);
    hw.Endianess = 'Little';
    hw.IntDivRoundTo = 'Zero';
    hw.ShiftRightIntArith = true;
    hw.LargestAtomicInteger = 'Long';
    hw.LargestAtomicFloat = 'Double';
end
```

To upgrade the data definitions contained in the file, run:

```
target.upgrade('rtwTargetInfo', 'myPathTo/rtwTargetInfo.m');
```

In the current folder, the function creates this `registerUpgradedTargets.m` file:

```
function processor = registerUpgradedTargets(varargin)
% This function was generated using target data export.

    % Create target.LanguageImplementation 'MyManufacturer-MyDevice'
    languageimplementation = target.create('LanguageImplementation');
    languageimplementation.AtomicFloatSize = 64;
    languageimplementation.AtomicIntegerSize = 64;
    languageimplementation.DataTypes.Char.Size = 8;
    languageimplementation.DataTypes.Double.Size = 64;
    languageimplementation.DataTypes.Float.Size = 64;
    languageimplementation.DataTypes.Half.IsSupported = false;
    languageimplementation.DataTypes.Half.Size = 16;
    languageimplementation.DataTypes.Int.Size = 32;
    languageimplementation.DataTypes.Long.Size = 64;
    languageimplementation.DataTypes.LongLong.IsSupported = false;
    languageimplementation.DataTypes.LongLong.Size = 64;
    languageimplementation.DataTypes.Pointer.Size = 64;
    languageimplementation.DataTypes.PtrDiffT.Size = 64;
    languageimplementation.DataTypes.Short.Size = 16;
    languageimplementation.DataTypes.SizeT.Size = 64;
    languageimplementation.Name = 'MyManufacturer-MyDevice';
    languageimplementation.WordSize = 64;

    % Create target.Processor 'MyManufacturer-MyDevice'
    processor = target.create('Processor');
    processor.LanguageImplementations(1) = languageimplementation;
    processor.Manufacturer = 'MyManufacturer';
    processor.Name = 'MyDevice';

    % Add the target objects to MATLAB memory
    target.add(processor, varargin{:});
end
```

To register the hardware device with MATLAB, run:

```
registerUpgradedTargets()
```

If you want the registration to persist across MATLAB sessions, run:

```
registerUpgradedTargets('UserInstall', true)
```

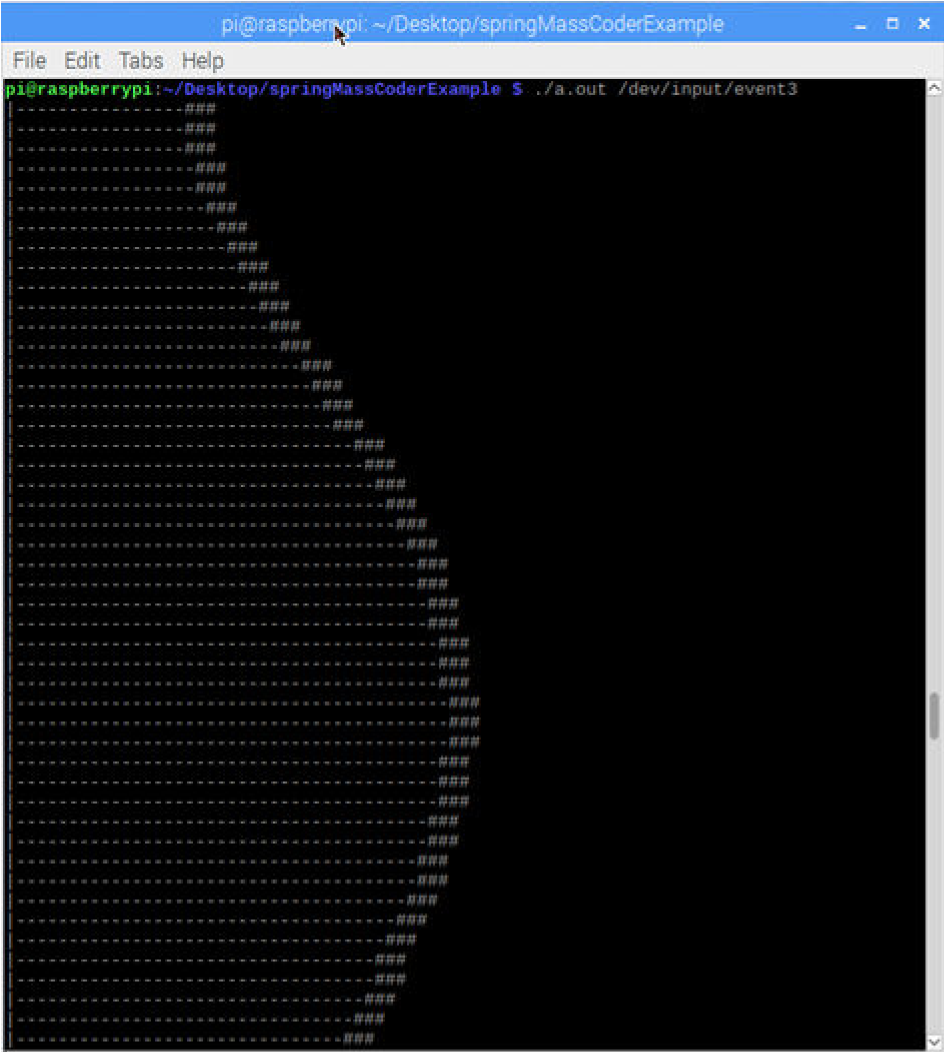
See Also

`target.LanguageImplementation` | `target.Processor`

Deploy Generated C Code to External Hardware: Raspberry Pi Examples

Use MATLAB Coder to generate C code for basic or advanced MATLAB algorithms, and then deploy the algorithms to external hardware platforms. These examples show deployment scenarios for the Raspberry Pi™ platform. You can use platforms such as the Raspberry Pi to prototype a more complex or larger scale deployment workflow. The Raspberry Pi runs a 32-bit Linux operating system environment on an ARM® processor.

This figure shows the generated code for a spring mass damper algorithm running on a Raspberry Pi. The C main function for the algorithm incorporates USB mouse input by using the Linux device file interface.



Prerequisites

- Raspberry Pi Model 3 B+. Older models of Raspberry Pi hardware might exhibit reduced performance.

- Network access or another file transfer mechanism, such as a microSD card reader.
- Remote desktop connection or an external monitor.
- Wired USB mouse for low-latency input.

Hardware Implementation Parameters

When generating code for external hardware, the code generator requires knowledge of the hardware-platform settings, as specified by a `coder.HardwareImplementation` object. This object contains implementation parameters that affect assumptions made by the code generator and that are important for achieving expected behavior.

For example, consider a Raspberry Pi running 32-bit Linux and an 8-bit Arduino® platform. The C compiler on the Arduino assigns an `int` 16 bits, whereas for the Raspberry Pi, the C compiler assigns 32 bits. If you generate C code that uses `int` variable declarations, then running the same code on both platforms can cause different integer overflow behavior.

Because of these platform differences, set the hardware implementation parameters specifically for whichever platform you expect to run the generated code on. By default, the parameters are set for the MATLAB host platform. You can use the MATLAB Coder app or command line to set hardware parameters, or you can use a Hardware Support Package.

Set Parameters by Using App and Command Line

To set the hardware implementation parameters via `coder.HardwareImplementation`, open your configuration object in the MATLAB Coder app. For example:

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

Configure the implementation parameters for a Raspberry Pi. Click the **Hardware** pane and select the **Device vendor** as `ARM Compatible` and **Device type** as `ARM Cortex`. This selection is equivalent to entering:

```
cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible->ARM Cortex';
```

Setting the `ProdHWDeviceType` parameter triggers the appropriate settings for all the other `coder.HardwareImplementation` parameters.

Set Parameters by Using Hardware Support Package

If you have access to the MATLAB Support Package for Raspberry Pi Hardware, you can set up a connection to your Raspberry Pi from inside the MATLAB environment. You can set the `coder.HardwareImplementation` settings by choosing `Raspberry Pi` from the **Hardware Board** menu in the MATLAB Coder app, or from the command line, by entering:

```
cfg = coder.config('lib');
hwObj = coder.hardware('Raspberry Pi');
cfg.Hardware = hwObj;
```

The `coder.hardware` function creates a `coder.Hardware` object. When you assign the `coder.Hardware` object to the configuration object, the hardware implementation parameters are set accordingly.

Hello World Example

Generate C Source Code for External Hardware

Consider an elementary MATLAB Hello World function.

```
function helloworld %#codegen
fprintf('Hello world!\n');
```

Create a configuration object and specify source code generation. Set the hardware implementation parameters for the Raspberry Pi.

```
cfg = coder.config('lib','ecoder',false);
cfg.GenCodeOnly = true;
cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible->ARM Cortex';
```

Generate code:

```
codegen -config cfg helloworld -report
```

Transfer Files to Device

You can package the generated code for file transfer by using the `packNGo` function. This function creates a zip file containing the required generated code files. The `packNGo` function does not include the generated example main files that you can use to compile the code into an executable. Move the generated example main files or your own handwritten main files separately.

From the directory from which you entered the `codegen` command, gather the files for deployment:

```
myBuildInfoFile = 'codegen/lib/helloworld/buildInfo.mat';
load(myBuildInfoFile);
packNGo(buildInfo);

movefile ./codegen/lib/helloworld/examples/main.c
movefile ./codegen/lib/helloworld/examples/main.h
```

Transfer the files from your host machine running MATLAB to your external hardware target. You can use a file transfer program for your platform or direct commands, such as `scp` with the destination IP address of the Raspberry Pi.

Build Code on Device

Once you have transferred the files to a directory, from the terminal, run `unzip` on the zip file. Then use the Linux `gcc` build tool to create an executable. Name it `helloworld` with the `-o` option:

```
gcc helloworld.c helloworld_initialize.c helloworld_terminate.c main.c -o helloworld
```

To run the executable and verify that the build was successful, enter:

```
./helloworld
```

The terminal displays the output:

```
Hello world!
```

Spring Mass Damper System Example

Generate Source Code for a Spring Mass Damper System

This example shows how to generate C source code for a spring mass damper system that you can then build and run on a Raspberry Pi.

The Spring Mass Damper Model

The spring mass system with damping is a fundamental system in mechanics and dynamics. By using the equations of motion you can solve for the displacement of the mass in response to different initial conditions and external forces.

The function `springMassEqns` encodes the equations of motion in the form of two first order linear differential equations. The variables `dxdt(1)` and `dxdt(2)` are the velocity and the acceleration of the mass, respectively. The variable `x(1)` represents the position of the mass.

```
function dxdt = springMassEqns(t,x,x0,k,m,c,F)
dxdt = zeros(2,1);
dxdt(1) = x(2);
dxdt(2) = F/m - k/m*(x(1)-x0) - c/m*x(2);
```

MATLAB Algorithm

To simulate the displacement as a function of time, the function `springMassStep` applies the ODE solver `ode45` to the equations of motion.

```
function [x] = springMassStep(xi,vi,ti,dt,g) %#codegen
% Set spring equilibrium position
x0 = 1;
% Set spring, mass, damper constants
k = 1000;
m = 10;
c = 25;
% Scale acceleration g like a gravity force
F = m*g;
% Solve ODE for displacement at ti + dt
initCond = [xi vi];
tspan = [ti ti+dt];
[~,x] = ode45(@(t,x) springMassEqns(t,x,x0,k,m,c,F),tspan,initCond);
```

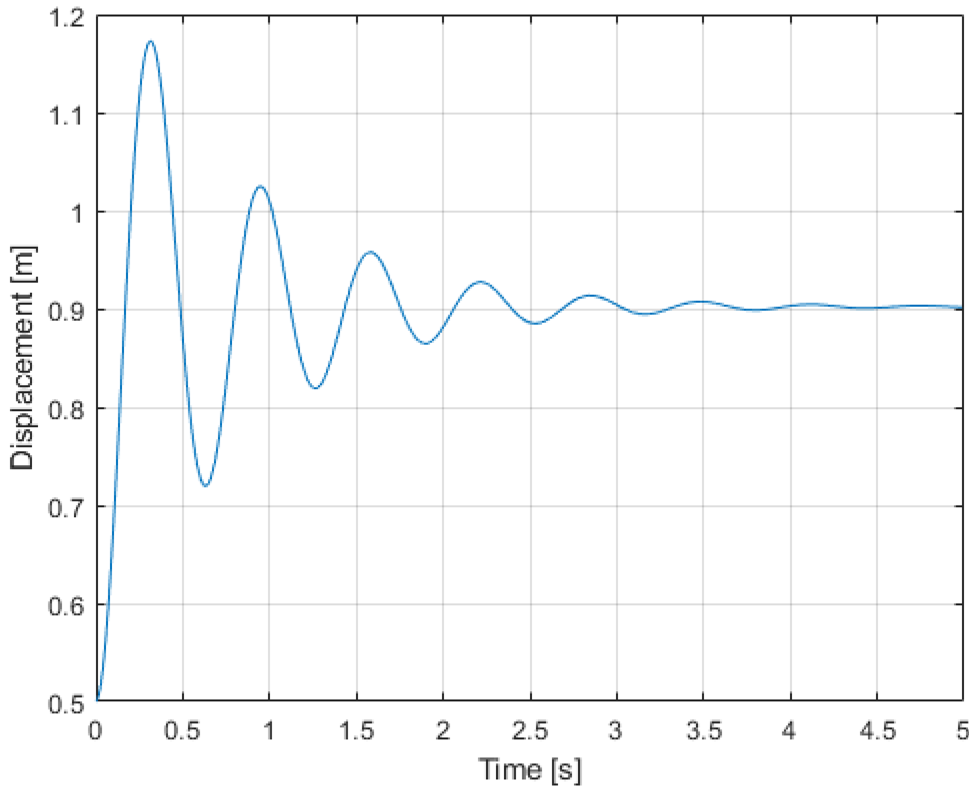
The `springMassTakeStep` function calls the `springMassStep` function and returns the final displacement at the end of a fixed time interval.

```
function [xf, vf] = springMassTakeStep(xi,vi,ti,dt,g) %#codegen
% Function springMassTakeStep acts as a wrapper for the ODE solving function, step.
% It takes the same input parameters as springMassStep, but only outputs the final
% position and velocity.
x = springMassStep(xi,vi,ti,dt,g);
xf = x(end,1);
vf = x(end,2);
```

Algorithm Results

To understand typical behavior of the model and the algorithm, the script `springMassSim` simulates the system over a typical parameter range. The output results show the displacement of the mass as a function of time. You can adjust the initial conditions, model parameters, and forcing function to see how the model responds.

```
springMassSim
```



Generate C Source Code

After verifying that the MATLAB model works as expected, generate C source code for deployment.

```
xi = 0.5;
vi = 0;
ti = 0;
dt = .01;
g = 0;

cfg = coder.config('lib','ecoder',false);
cfg.GenCodeOnly = true;
cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible->ARM Cortex';
codegen springMassTakeStep -args {xi,vi,ti,dt,g} -config cfg -report
```

```
Code generation successful: To view the report, open('codegen\lib\springMassTakeStep\html\report
```

Transfer Files to Device

After generating the C source code for the algorithm, you can modify the example `main.c` and `main.h` files for your application. For this example, the attached file `springMass_main.c` shows how to use the generated code. The corresponding header file `springMass_main.h` is also attached to the previous example with the supporting files.

From the directory from which you generated code, gather the files for deployment:

```
myBuildInfoFile = 'codegen/lib/springMassTakeStep/buildInfo.mat';
load(myBuildInfoFile);
packNGo(buildInfo);
```

Transfer the zip file and your main `.c` and `.h` file from your host machine that is running MATLAB to the target. You can use a file transfer program for your platform or direct commands, such as `scp` with the destination IP address of the Raspberry Pi.

Build Code on Device

Main File

The main function `springMass_main.c` executes the generated code to simulate the displacement of the spring mass damper system over time. The function uses the USB mouse input from the Raspberry Pi to impart a force on the mass. The strength of the force is proportional to the speed of the horizontal mouse movement. If you do not move the mouse, the example simulates the unforced dynamics. To provide a visualization of the dynamics, the main file includes a routine to print the position of the mass over time.

Device File

To use the mouse input, you must identify which device file on your system corresponds to the mouse. On the Linux platform, external USB device input is recorded in a device file stored in the `/dev/input/` folder. The `/dev/input/` folder typically contains files for multiple input devices. To identify which file corresponds to your USB mouse, use the `od` command on each file, and check to see which file updates in response to mouse movement.

```
od filename
```

Build

To build the code from the Linux terminal, navigate to the location where you transferred your files. Unzip the zip file. Use the `gcc` command and specify all the `.c` files from the spring mass example:

```
gcc *.c -o springMassSim -lm
```

The `-lm` flag instructs the compiler to link to the required C math libraries. To run the executable, specify the previously identified USB mouse device file, here assumed to be `event0`:

```
./springMassSim /dev/input/event0
```

Move the mouse to apply a force to the mass and view the resulting dynamics. If the mass does not respond to mouse movement, try specifying a different device file. Terminate the program by entering `ctrl + c` during execution.

See Also

`coder.HardwareImplementation` | `coder.hardware` | `packNGo`

More About

- "Use an Example C Main in an Application" on page 31-25

Deploy Generated Code

Deployment is the process of using the generated code in an application that runs outside of the MATLAB environment. Many topics and considerations are relevant to the deployment process.

Main Function

To create an application, create or use a C/C++ main function to call the C/C++ entry-point functions generated from your MATLAB functions. The main function specifies input, output, and other functionality that your MATLAB algorithms do not specify. The code generator produces an example main function by default. Use the generated example main as a starting point for creating a new main function. The example main provides a clear example for how to pass input to and output from the generated code. For more information and examples, see:

- "Incorporate Generated Code Using an Example Main Function" on page 31-23
- "Structure of Generated Example C/C++ Main Function" on page 31-47

Your C/C++ code must call an initialize function and a terminate function that are generated in addition to your C/C++ entry-point functions. By default, the generated C/C++ entry-point function calls the initialize function. The generated example main function calls the terminate function. As you create and edit your own main function, ensure that both initialize and terminate functions are called. For more information, see:

- "Use Generated Initialize and Terminate Functions" on page 27-25

Generated Function Interfaces

To write a main function, you must be familiar with the generated function interfaces.

Data Types

The generated C/C++ function prototypes use data types that correspond to the types that you use in your MATLAB code. See "Mapping MATLAB Types to Types in Generated Code" on page 33-15. With Embedded Coder, you can customize the appearance and style of generated data types. See "Code Appearance" (Embedded Coder).

Argument Passing Behavior

C/C++ entry-point functions generated from MATLAB Coder follow these conventions:

- Pass input arrays by reference.
- Return output arrays by reference.
- Pass input scalars by value.
- Return scalars by value for single-output functions.
- Return scalars by reference:
  - For functions with multiple outputs.
  - When you use the same variable as input and output.

If you use the same variable as input and output in your MATLAB code, the generated code passes the scalar by reference. See “Avoid Data Copies of Function Inputs in Generated Code” on page 34-6.

Array Definition

Fixed-size and variable-size arrays are represented by different data types in the generated C/C++ code. For more information, see “Use C Arrays in the Generated Function Interfaces” on page 31-3.

Executable Applications

After you generate code and write a main file that uses the generated code, then you must build your code into an executable by using either MATLAB Coder or other build tools. You might want to run the executable application on your MATLAB platform, the host platform, or on a different platform, the target platform. To package the required elements of the generated code into an exportable zip file that you can manually transfer to a target platform, use the `packNGo` function.

The code generation folder does not necessarily contain all files used by the generated code. The folder can also contain supporting files that are not used by the generated code. Use `packNGo` to package and move generated code files rather than moving and including the entire code generation folder contents.

Binary Deployment

You can generate binaries directly by using the `codegen` command or the MATLAB Coder app by selecting a build type of static library, dynamic library, or executable (lib, dll, or exe). By default, the generated binaries are functional for the host platform hardware and operating system. To build an executable, you must specify or provide a main file. For an example, see “Generating Standalone C/C++ Executables from MATLAB Code” on page 27-4. If you set the `GenerateExampleMain` property of a configuration object to `'GenerateCodeAndCompile'`, the code generator builds an executable by using the generated example main file.

If you want to deploy your code to another platform, then you can use hardware support packages that provide support for generating and building the binary code for that platform. This support includes specific toolchains and code generation configuration settings that the target hardware requires. For a list of support packages provided for MATLAB Coder, see “MATLAB Coder Supported Hardware”. Many additional hardware support packages are available for Embedded Coder. See “Embedded Coder Supported Hardware” (Embedded Coder). If you want to specify a custom toolchain for build that is not available from a hardware support package, you can register your own toolchain. See “Custom Toolchain Registration”.

In the MATLAB Coder app, select a hardware support package during the **Generate Code** step from the **Hardware Board** drop-down list. From the command line, specify a hardware support package by using the `coder.hardware` function.

Source Code Deployment

In certain cases, you might choose to generate source code, and then manually build the source code for your project. Manually build the source code when:

- Your generated source code is easy to build. For example, your generated code does not require linking against additional libraries.
- You want to create an executable for custom hardware for which you do not have a hardware support package.

- You are knowledgeable in building C/C++ source code or the build system for the target platform is already configured.

The code generator produces a `buildInfo` object that enables you to view and modify build information that MATLAB Coder uses to create binary outputs. You can use this information for understanding how to manually build your generated code. See “Build Process Customization” on page 27-116 and `RTW.BuildInfo`.

The code generator produces a makefile that shows build information such as compile and link flags. Find this makefile in the code generation folder. The generated makefile is specific to the target platform that you specify by selecting a hardware support package or the host platform, if no hardware support package is specified. If you manually build your source code, you can use this makefile to identify and troubleshoot build requirements, such as compiling and linking flags.

To see how to manually configure code generation and build for a target platform, see “Deploy Generated C Code to External Hardware: Raspberry Pi Examples” on page 31-64.

Static and Dynamic Libraries

When you want to use generated code functionality in an existing C/C++ project, you can generate a static library or dynamic library. Libraries can provide a more modular interface than generated source code. When MATLAB Coder generates a static library or dynamic library:

- The library is suitable for the platform that you are working on, unless you specify an alternative platform through a hardware support package.
- The generated header files for C code explicitly declare the exported functions as `extern "C"` to simplify integration of the library into C++ applications.
- The generated library file extensions correspond to the MATLAB host platform operating system.

Operating System	Static Library	Dynamic Library
Windows	`.lib`	`.dll` and `.lib` for corresponding import library
macOS	`.a`	`.dylib`
Linux	`.a`	`.so`

You must compile and link against libraries when you build an executable. When an executable that uses a dynamic library runs, the library must be on the system path or in the executable folder. For examples of using a generated library, see:

- “Use a Dynamic Library in a Microsoft Visual Studio Project” on page 31-20
- “Integrate Multiple Generated C++ Code Projects” on page 39-14

Loading generated dynamic libraries into MATLAB by using the `loadlibrary` function is not recommended and can result in incorrect behaviors or crashes.

Generated File Structure

By default, MATLAB Coder produces one C code file for each MATLAB code file. You can choose to partition the generated code into one single file and generate code with customized output folders and binary names. See “How MATLAB Coder Partitions Generated Code” on page 27-106.

With Embedded Coder, you can customize generated file names. See “Customize C/C++ File Names Generated from MATLAB Code” (Embedded Coder).

Code Verification

Before you deploy generated code for execution outside the MATLAB environment, you can verify it inside the MATLAB environment. The primary workflow for verification with MATLAB Coder is the generation and execution of C/C++ MEX functions. MEX functions run inside the MATLAB environment and provide run-time error checking and diagnostics. See “Code Verification”.

Embedded Coder offers deep additional functionality for code verification and testing. You can use software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution to test the behavior of the generated code on software and hardware outside of the MATLAB environment. See “Verification” (Embedded Coder).

Custom Hardware Considerations

If your target supports only `single` data types and not `double` data types, you can generate single-precision code by using the `codegen -singleC` option. This option requires Fixed-Point Designer. If your target supports only integer data types, use the `-float2fixed` option. See `codegen`.

Other Deployment Strategies

MATLAB Coder generates readable and portable C/C++ code for a subset of the MATLAB language. If you want to generate a standalone executable application for the host platform that uses the MATLAB Runtime libraries, but runs without a MATLAB license, then use MATLAB Compiler SDK. For a product comparison, see https://www.mathworks.com/matlabcentral/answers/223937-should-i-use-matlab-compiler-sdk-or-matlab-coder-to-integrate-my-matlab-applications-with-c-c

See Also

`coder.hardware` | `packNGo`

More About

- “Mapping MATLAB Types to Types in Generated Code” on page 33-15
- “Embedded Coder Capabilities for Code Generation from MATLAB Code” (Embedded Coder)

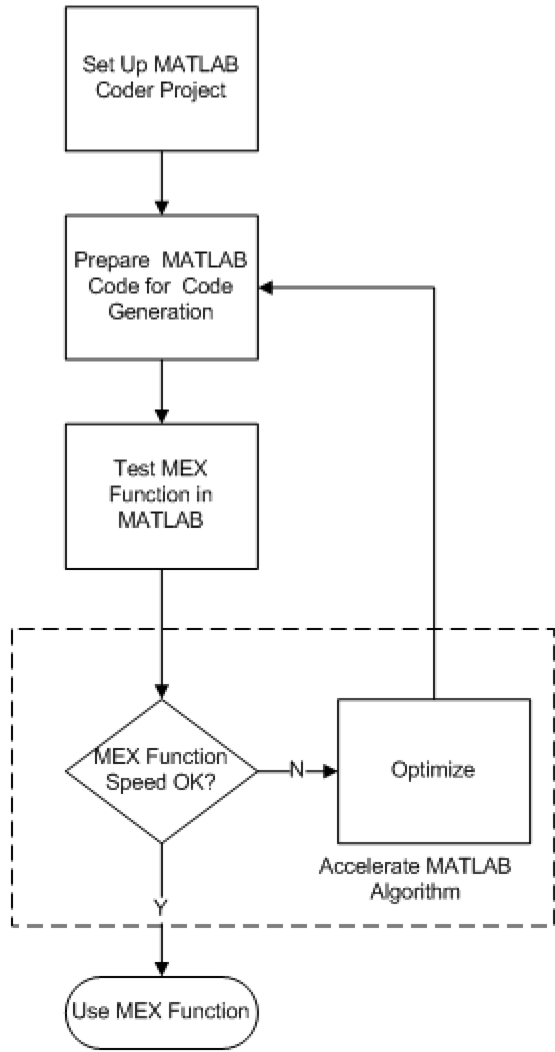
External Websites

- https://www.mathworks.com/matlabcentral/fileexchange/62243-run-on-hardware

Accelerating MATLAB Algorithms

- “Workflow for Accelerating MATLAB Algorithms” on page 32-2
- “Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms” on page 32-3
- “Accelerate MATLAB Algorithms” on page 32-6
- “Modifying MATLAB Code for Acceleration” on page 32-7
- “Profile MEX Functions by Using MATLAB Profiler” on page 32-8
- “Control Run-Time Checks” on page 32-12
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 32-14
- “Control Compilation of parfor-Loops” on page 32-18
- “Reduction Assignments in parfor-Loops” on page 32-19
- “Classification of Variables in parfor-Loops” on page 32-20
- “Accelerate MATLAB Algorithms That Use Parallel for-Loops (parfor)” on page 32-27
- “Specify Maximum Number of Threads in parfor-Loops” on page 32-28
- “Troubleshooting parfor-Loops” on page 32-29
- “Generate MEX Code to Accelerate Simulation of Bouncing Balls” on page 32-30
- “Generate MEX Code to Calculate Geodesics in Curved Space-Time” on page 32-34
- “Generate Accelerated MEX Code for Reverberation Using MATLAB® Classes” on page 32-38
- “Using PARFOR to Speed Up an Image Contrast Enhancement Algorithm” on page 32-40
- “Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler” on page 32-49

Workflow for Accelerating MATLAB Algorithms



See Also

- “Set Up a MATLAB Coder Project” on page 24-2
- “Workflow for Preparing MATLAB Code for Code Generation” on page 25-2
- “Workflow for Testing MEX Functions in MATLAB” on page 26-3
- “Modifying MATLAB Code for Acceleration” on page 32-7

Best Practices for Using MEX Functions to Accelerate MATLAB Algorithms

In this section...

"Accelerate Code That Dominates Execution Time" on page 32-3
"Include Loops Inside MEX Function" on page 32-3
"Avoid Generating MEX Functions from Unsupported Functions" on page 32-4
"Avoid Generating MEX Functions if Built-In MATLAB Functions Dominate Run Time" on page 32-4
"Minimize MEX Function Calls" on page 32-4

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 Your Code to Improve Performance".

Include Loops Inside MEX Function

Instead of calling a MEX function inside a loop in the MATLAB code, include the loop inside the MEX function. Including the loop eliminates the overheads in calling the MEX function for every run of the loop.

For example, the following code finds the greatest element in every row of a 1000–by–1000 matrix, `mat`. You can accelerate sections 1,2, and 3 using a MEX function.:

```
% Section 1 begins
for i = 1:10000

   % Section 2 begins
   max = mat(i,0); % Initialize max
   for j = 1:10000

     % Section 3 begins
     if (mat(i,j) > max)
       max = mat(i,j) % Store the current maximum
     end
```

```
    % Section 3 ends

  end
  % Section 2 ends

end
% Section 1 ends
```

Accelerate section 1 using a MEX function. Accelerate section 1 first so that the MEX function is called only once.. If you cannot accelerate section 1 first, then accelerate sections 2 or 3, in that order. If section 2 (or 3) is accelerated using a MEX function, the function is called 10000 (or 10000 × 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” 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 “Use MATLAB Engine to Execute a Function Call in Generated Code” on page 20-8.

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 “Use MATLAB Engine to Execute a Function Call in Generated Code” on page 20-8.

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.

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 32-7
- “Control Run-Time Checks” on page 32-12

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 34-33
- "Inline Code" on page 34-8
- "Avoid Data Copies of Function Inputs in Generated Code" on page 34-6

Profile MEX Functions by Using MATLAB Profiler

You can profile execution times for MEX functions generated by MATLAB Coder by using the MATLAB Profiler. The profile for the generated code shows the number of calls and the time spent for each line of the corresponding MATLAB function. Use the Profiler to identify the lines of MATLAB code that produce generated code that take the most time. This information can help you identify and correct performance issues early in the development cycle. For more information on the MATLAB Profiler, see `profile` and "Profile Your Code to Improve Performance".

The graphical interface to the Profiler is not supported in MATLAB Online.

MEX Profile Generation

You can use the MATLAB Profiler with a generated MEX function. Alternatively, if you have a test file that calls your MATLAB function, you can generate the MEX function and profile it in one step. You can perform these operations at the command line or in the MATLAB Coder app.

To use the Profiler with a generated MEX function:

1. Enable MEX profiling by setting the configuration object property `EnableMexProfiling` to `true`.

   Alternatively, you can use `codegen` with the `-profile` option.

   The equivalent setting in the MATLAB Coder app is **Enable execution profiling** in the **Generate** step.
2. Generate the MEX file `MyFunction_mex`.
3. Run the MATLAB Profiler and view the Profile Summary Report, which opens in a separate window.

   ```
   profile on;
   MyFunction_mex;
   profile viewer;
   ```

   Make sure that you have not changed or moved the original MATLAB file `MyFunction.m`. Otherwise, the Profiler does not consider `MyFunction_mex` for profiling.

If you have a test file `MyFunctionTest.m` that calls your MATLAB function, you can:

- Generate the MEX function and profile it in one step by using `codegen` with the `-test` and the `-profile` options. If you turned on the MATLAB Profiler before, turn it off before you use these two options together.

  ```
  codegen MyFunction -test MyFunctionTest -profile
  ```
- Profile the MEX function by selecting **Enable execution profiling** in the **Verify** step of the app. If you turned on the MATLAB Profiler before, turn it off before you perform this action.

Example

You use the Profiler to identify the functions or the lines of the MATLAB code that produce generated code that take the most time. Following is an example of a MATLAB function that converts the representation of its input matrices `A` and `B` from row-major to column-major layout in one of its lines.

Such a conversion has a long execution time for large matrices. Avoiding the conversion by modifying that particular line makes the function more efficient.

Consider the MATLAB function:

```
function [y] = MyFunction(A,B) %#codegen

% Generated code uses row-major representation of matrices A and B
coder.rowMajor;
length = size(A,1);

% Summing absolute values of all elements of A and B by traversing over the
% matrices row by row
sum_abs = 0;
for row = 1:length
   for col = 1:length
       sum_abs = sum_abs + abs(A(row,col)) + abs(B(row,col));
   end
end

% Calling external C function 'foo.c' that returns the sum of all elements
% of A and B
sum = 0;
sum = coder.ceval('foo',coder.ref(A),coder.ref(B),length);

% Returning the difference of sum_abs and sum
y = sum_abs - sum;
end
```

The generated code for this function uses a row-major representation of the square matrices `A` and `B`. The code first computes `sum_abs` (the sum of absolute values of all elements of `A` and `B`) by traversing over the matrices row by row. This algorithm is optimized for matrices that are represented in a row-major layout. The code then uses `coder.ceval` to call the external C function `foo.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double *A, double *B, double length)
{
 int i,j,s;
 double sum = 0;
 s = (int)length;

 /*Summing all the elements of A and B*/
 for(i=0;i<s*s;i++)
 {
         sum += A[i] + B[i];
 }
 return(sum);
}
```

The corresponding C header file `foo.h` is:

```
#include "rtwtypes.h"

double foo(double *A, double *B, double length);
```

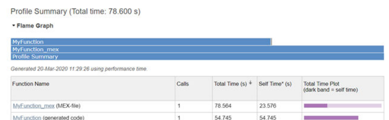
`foo.c` returns the variable `sum`, which is the sum of all elements of `A` and `B`. The performance of the function `foo.c` is independent of whether the matrices `A` and `B` are represented in row-major or column-major layouts. `MyFunction` returns the difference of `sum_abs` and `sum`.

You can measure the performance of `MyFunction` for large input matrices `A` and `B`, and then optimize it further:

1. Enable MEX profiling and generate MEX code for `MyFunction`. Run `MyFunction_mex` for two large random matrices `A` and `B`. View the Profile Summary Report.

```
A = rand(20000);
B = rand(20000);

codegen MyFunction -args {A,B} foo.c foo.h -profile

profile on;
MyFunction_mex(A,B);
profile viewer;
```

A separate window opens showing the Profile Summary Report.



Profile Summary (Total time: 78.600 s)

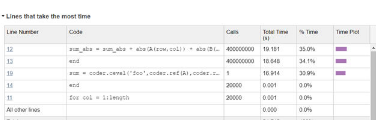
▾ Flame Graph

Generated 20-Mar-2020 11:29:26 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
MyFunction_mex (MEX-file)	1	78.564	23.576	
MyFunction (generated code)	1	54.745	54.745	

The Profile Summary Report shows the total time and the self time for the MEX file and its child, which is the generated code for the original MATLAB function.

2. Under Function Name, click the first link to view the Profile Detail Report for the generated code for `MyFunction`. You can see the lines where the most time was spent:



▾ Lines that take the most time

Line Number	Code	Calls	Total Time (s)	% Time	Time Plot
12	sum_abs = sum_abs + abs(A(row,col)) + abs(B(…	400000000	19.181	35.0%	
13	end	400000000	18.648	34.1%	
19	sum = coder.ceval('foo',coder.ref(A),coder.r…	1	16.914	30.9%	
14	end	20000	0.001	0.0%	
11	for col = 1:length	20000	0.001	0.0%	
All other lines			0.000	0.0%	
Totals			54.745	100%	

3. The line calling `coder.ceval` takes a lot of time (16.914 s). This line has considerable execution time because `coder.ceval` converts the representation of the matrices `A` and `B` from row-major layout to column-major layout before passing them to the external C function. You can avoid this conversion by using an additional argument `-layout:rowMajor` in `coder.ceval`:

```
sum = coder.ceval('-layout:rowMajor','foo',coder.ref(A),coder.ref(B),length);
```

4. Generate the MEX function and profile again using the modified `MyFunction`.

```
A = rand(20000);
B = rand(20000);

codegen MyFunction -args {A,B} foo.c foo.h -profile

profile on;
MyFunction_mex(A,B);
profile viewer;
```

The Profile Detail Report for `MyFunction` shows that the line calling `coder.ceval` now takes only 0.653 s:

▾ Lines that take the most time

Line Number	Code	Calls	Total Time (s)	% Time	Time Plot
12	sum_abs = sum_abs + abs(A(row,col)) + abs(B(…	400000000	26.590	48.2%	
13	end	400000000	25.331	46.0%	
19	sum = coder.ceval('-layout:rowMajor','foo',c…	1	0.653	1.2%	
14	end	20000	0.001	0.0%	
11	for col = 1:length	20000	0.001	0.0%	
All other lines			2.535	4.6%	
Totals			55.112	100%	

Effect of Folding Expressions on MEX Code Coverage

When you use `coder.const` to fold expressions into constants, it causes a difference in the code coverage between the MATLAB function and the MEX function. For example, consider the function:

```
function y = MyFoldFunction %#codegen
a = 1;
b = 2;
c = a + b;
y = 5 + coder.const(c);
end
```

Profiling the MATLAB function `MyFoldFunction` shows this code coverage in the Profile Detail Report:

▾ Function listing

```
Time      Calls  Line
                   1  function y = MyFoldFunction %#codegen
                   2  a = 1;
                   3  b = 2;
                   4  c = a + b;
< 0.001       1    5  y = 5 + coder.const(c);
< 0.001       1    6  end
```

However, profiling the MEX function `MyFoldFunction_mex` shows a different code coverage:

▾ Function listing

```
Time      Calls  Line
                   1  function y = MyFoldFunction %#codegen
< 0.001       1    2  a = 1;
< 0.001       1    3  b = 2;
< 0.001       1    4  c = a + b;
  0.008       1    5  y = 5 + coder.const(c);
< 0.001       1    6  end
```

Lines 2, 3, and 4 are not executed in the generated code because you have folded the expression `c = a + b` into a constant for code generation.

This example uses user-defined expression folding. The code generator sometimes automatically folds certain expressions to optimize the performance of the generated code. Such optimizations also cause the coverage of the MEX function to be different from the MATLAB function.

See Also

`codegen` | `coder.MexCodeConfig` | `coder.ceval` | `coder.const` | `coder.rowMajor` | `profile`

More About

- “Profile Your Code to Improve Performance”
- “Generate Code That Uses Row-Major Array Layout” on page 37-4

Control Run-Time Checks

In this section...
“Types of Run-Time Checks” on page 32-12
“When to Disable Run-Time Checks” on page 32-12
“How to Disable Run-Time Checks” on page 32-13

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 “Using the coder.extrinsic Construct” on page 20-9.

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.

Consider disabling...	Only if...
Extrinsic calls	You are using extrinsic calls only for functions that do not affect application results.

How to Disable Run-Time Checks

You can disable run-time checks explicitly from the project settings dialog box, the command line, or a MEX configuration dialog box.

Disabling Run-Time Checks Using the MATLAB Coder App

1. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow ▼.
2. Set **Build type** to `MEX`.
3. Click **More Settings**.
4. On the **Speed** tab, clear **Ensure memory integrity**, **Enable responsiveness to CTRL+C and graphics refreshing**, or **Keep Extrinsic calls**, as applicable.

Disabling Run-Time Checks From the Command Line

1. In the MATLAB workspace, define the MEX configuration object:

   ```
   mexcfg = coder.config('mex');
   ```
2. At the command line, set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties to false, as applicable:

   ```
   mexcfg.IntegrityChecks = false;
   mexcfg.ExtrinsicCalls = false;
   mexcfg.ResponsivenessChecks = false;
   ```

Algorithm Acceleration Using Parallel for-Loops (parfor)

In this section...

"Parallel for-Loops (parfor) in Generated Code" on page 32-14
"How parfor-Loops Improve Execution Speed" on page 32-14
"When to Use parfor-Loops" on page 32-15
"When Not to Use parfor-Loops" on page 32-15
"parfor-Loop Syntax" on page 32-15
"parfor Restrictions" on page 32-16

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 32-14.

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 32-2.

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 32-16.

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 32-20. 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 32-22.
- 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 `https://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 36-3.
- 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 30-19.
- 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 32-20.
- **`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` with one of these methods:

- By using the `codegen` function with `-O disable:openmp` option at the command line.
- By using a code generation configuration object with the property `EnableOpenMP` set to false. For example:

  ```
  cfg = coder.config('lib');
  cfg.EnableOpenMP = false;
  codegen myFunction -config cfg
  ```
- 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.

See Also

`parfor`

More About

- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 32-14
- “Configure Build Settings” on page 27-13

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 32-22.

Multiple Reductions in a parfor-Loop

You can perform the same reduction assignment multiple times within a `parfor`-loop provided that you use the same data type each time.

For example, in the following `parfor`-loop, `u(i)` and `v(i)` must be the same type.

```
parfor i = 1:10;
  X = X + u(i);
  X = X + v(i);
end
```

Similarly, the following example is valid provided that `u(i)` and `v(i)` are the same type.

```
parfor i=1:10
  r = foo(r,u(i));
  r = foo(r,v(i));
end
```

Classification of Variables in parfor-Loops

In this section...
"Overview" on page 32-20
"Sliced Variables" on page 32-20
"Broadcast Variables" on page 32-22
"Reduction Variables" on page 32-22
"Temporary Variables" on page 32-25

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 32-24
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 32-24

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.

<table>
<tr><th>Invalid Use of Reduction Variable</th><th>Valid Use of Reduction Variable</th></tr>
<tr><td><pre>function [y0, y1, y2] = pfuserfcn(u)
    y0 = 0;
    y1 = 1;
    [F, N] = fiprops();
    y2 = fi(1,N,F);
    parfor (i=1:numel(u),12)
        y0 = y0 + u(i);
        y1 = y1 * u(i);
        y2 = fcn(y2, u(i));
    end
end

function y = fcn(u, v)
  y = u * v;
end</pre></td><td><pre>function [y0, y1, y2] = pfuserfcn(u)
    y0 = 0;
    y1 = 1;
    [F, N] = fiprops();
    y2 = fi(1,N,F);
    parfor (i=1:numel(u),12)
        y0 = y0 + u(i);
        y1 = y1 * u(i);
        y2 = fcn(y2, u(i));
    end
end
% fcn handles inputs of type double
% and fi
function y = fcn(u, v)
    if isa(u,'double')
        y = u * v;
    else
        [F, N] = fiprops();
        y = u * fi(v,N,F);
    end
end

function [F, N] = fiprops()
    N = numerictype(1,96,30);
    F = fimath('ProductMode',...
        'SpecifyPrecision',...
        'ProductWordLength',96);
end</pre></td></tr>
</table>

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 `https://www.mathworks.com/support/compilers/current_release/`.

Generate MEX Code to Accelerate 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.

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
```

```
Code generation successful.
```

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.

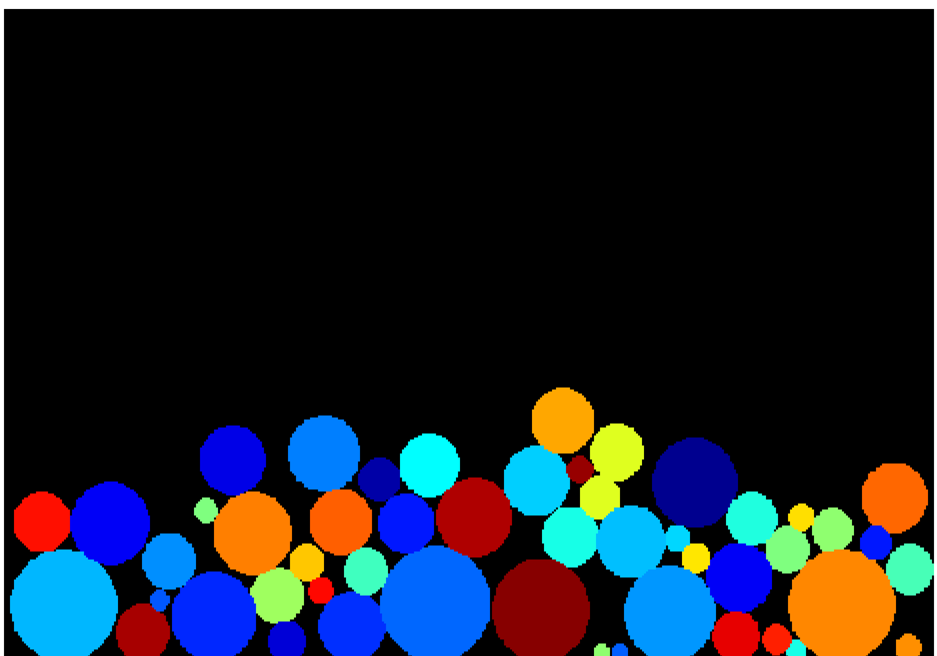
Compare Results

Run and time the original `run_balls` function followed by the generated MEX function.

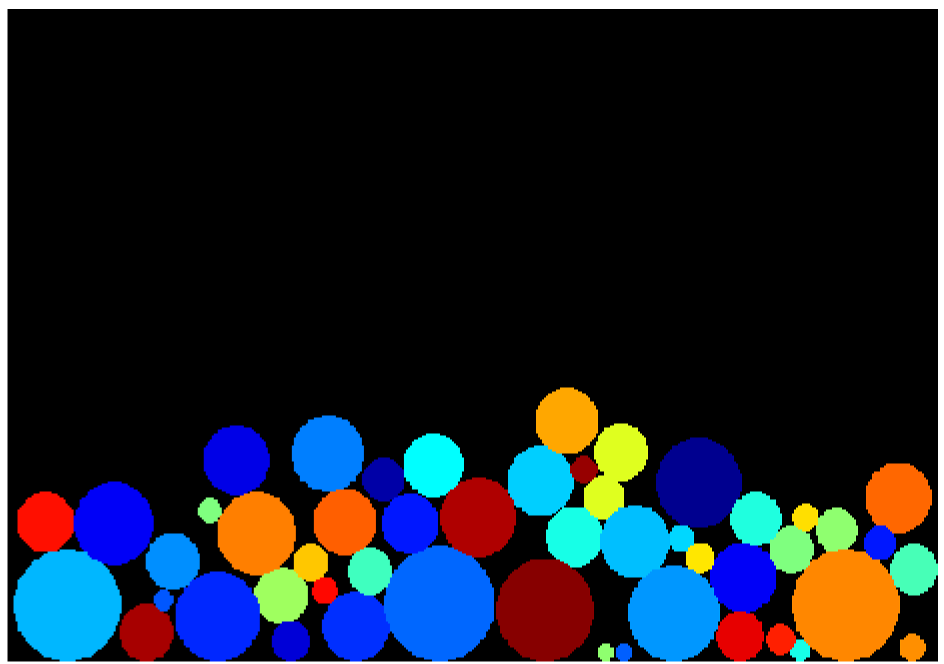
```
tic, run_balls(50); t1 = toc;
```

```
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
```

```
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
```

Estimated speed up is:

```
fprintf(1, 'Speed up: x ~%2.1f\n', t1/t2);
```

```
Speed up: x ~1.9
```

Generate MEX Code to Calculate Geodesics in Curved Space-Time

These examples are using Einstein's General Relativity to calculate geodesics in curved space-time.

Prerequisites

There are no prerequisites for this example.

Example: Computing the Precession of the Planet Mercury

This example computes the precession of the planet Mercury numerically. The precession is a slight rotation of the elliptical orbit around the sun. Analytically, using the equations of general relativity the value is extremely small, an extra 43" (arc seconds) per century. An arc second is 1/3600th of one degree (counting 360 degrees for a complete revolution.) Even though the extra precession is extremely small it matches exactly with observation. Pure Newtonian mechanics (if we choose to ignore all the other planets of our solar system) predicts no precession.

This application is using Euler's method with variable time step where the major time step is .5 seconds. We reduce the time step as we approach one complete revolution. The precession is computed as the planet is reaching its maximum distance from the sun for which we compute its relative angle to the coordinate axis.

Generate the MEX Function: Precession of the Planet Mercury

Generate a MEX function using the command `codegen` followed by the name of the MATLAB file to compile.

```
codegen gr_mercury_precession
```

```
Code generation successful.
```

The `gr_mercury_precession` 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 `gr_mercury_precession_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

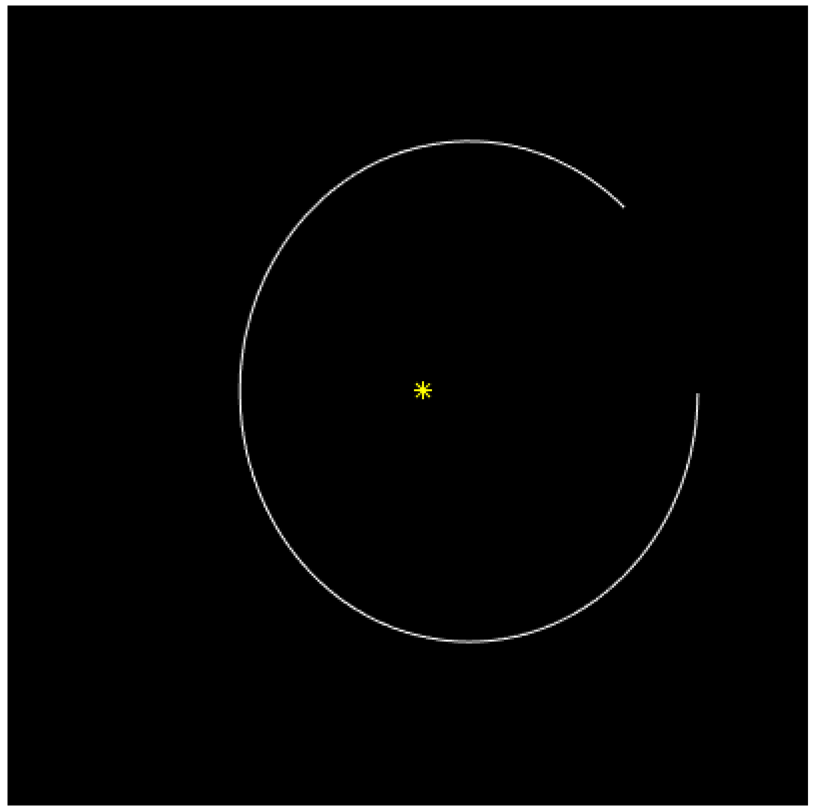
Run the MEX Function: Precession of the Planet Mercury

Run the generated MEX function.

```
gr_mercury_precession_mex
```

```
Progress: 5%
Progress: 10%
Progress: 15%
Progress: 20%
Progress: 25%
Progress: 30%
Progress: 35%
Progress: 40%
Progress: 45%
Progress: 50%
Progress: 55%
Progress: 60%
```

```
Progress: 65%
Progress: 70%
Progress: 75%
Progress: 80%
Progress: 85%
Progress: 90%
Progress: 95%
Progress: 100%
```



```
precession: 0.10468" (0 years 87.87009 days) => 43.481"/century
```

Example: Ray-tracing a Black Hole

Einstein's equations of motion in general relativity can handle any object at any speed, so let's apply it to photons that travel with the speed of light. In this configuration we have a black hole in front of a background image. To make the effect more visible, we increase the mass of the black hole to astronomical proportions as well as the background image. In this way we can study the effects of gravitational lensing; the background image becomes distorted by the curved space-time produced by the black hole.

Generate a MEX Function: Ray-tracing a Black Hole

```
codegen gr_raytrace
```

```
Code generation successful.
```

Run the MEX Function: Ray-tracing a Black Hole

Ray-tracing the picture takes a minute or two on a 2 GHz x86 machine. On your screen, you see the original picture (the Vittorio Emanuele Mall in Milano, Italy) and, to the right, the rendered image of the same picture with a black hole in front of it.

```
gr_raytrace_mex('mall.jpg');
```

```
Progress: 5%
Progress: 10%
Progress: 15%
Progress: 20%
Progress: 25%
Progress: 30%
Progress: 35%
Progress: 40%
Progress: 45%
Progress: 50%
Progress: 55%
Progress: 60%
Progress: 65%
```

```
Progress: 70%
Progress: 75%
Progress: 80%
Progress: 85%
Progress: 90%
Progress: 95%
Progress: 100%
```

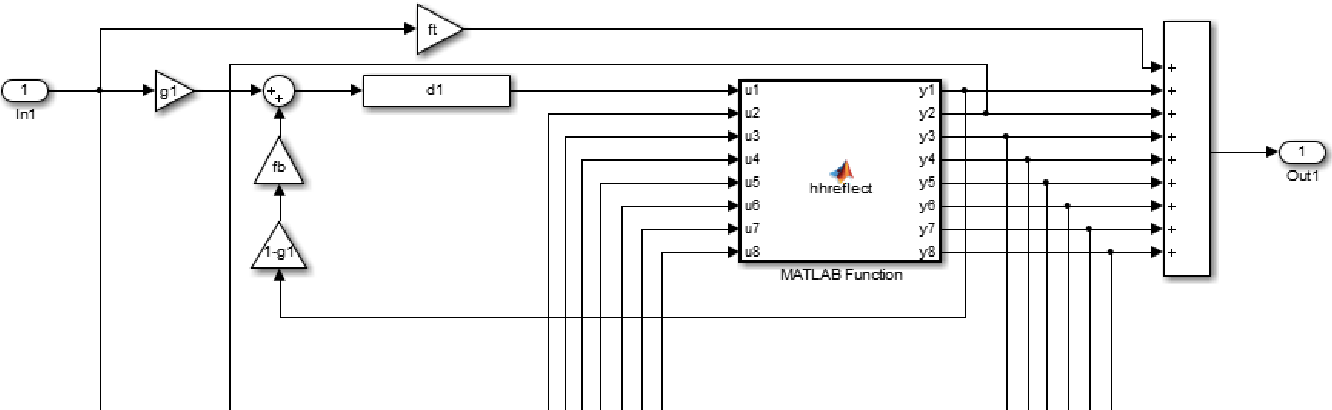
Generate Accelerated MEX Code for Reverberation Using MATLAB® Classes

This example shows how to accelerate the execution of a MATLAB algorithm that uses MATLAB classes. The classes create a reverberation effect, that is, the "echo" you hear in a large empty room.

Implementing a Simple Reverberation Effect

There are many ways to implement a reverberation effect with different characteristics. In terms of audio quality, this is not an advanced effect, but shows the capabilities of using MATLAB classes with MATLAB Coder™.

This reverberation effect is implemented based on the following block diagram:



The diagram shows only the first delay line. Imagine another seven delay lines being repeated in the diagram but each delay line has an individual delay and associated feedback gain block. The Householder reflection (i.e. `hhreflect` function) is essentially mixing/permuting the signals without changing the energy of the total signal. Therefore, we are essentially duplicating the incoming signal and feeding it back with small time displacements. The result is a reverberation effect.

Files Used

- `reverb_test.m`: Main file testing the reverberation effect
- `do_reverb.m`: Function abstraction of the Reverb class
- `Reverb.m`: Effect implementation implemented as a MATLAB class
- `Delay.m`: Delay effect for `Reverb.m` implemented as a MATLAB class
- `hhreflect.m`: Householder reflection for `Reverb.m`
- `get_prime.m`: Function to compute prime numbers (for `Reverb.m`)
- `speech_dft.mat`: Test sample file

Generate a MEX Function

```
codegen do_reverb
```

```
Code generation successful.
```

Run the MEX Function

This processes the sample file (`speech_dft.mat`), applies the reverberation effect, and outputs the result to the computer's audio output.

```
reverb_test;
```

```
Running time = 20 milliseconds
```

Generate a Faster MEX Function

Disable the integrity checks (e.g. out of bound checks for matrices) to obtain a faster but potentially unsafe MEX function.

```
cfg = coder.config;
cfg.IntegrityChecks = false;
codegen -config cfg do_reverb
```

```
Code generation successful.
```

Retry the MEX Function

```
reverb_test;
```

```
Running time = 7 milliseconds
```

Using PARFOR to Speed Up an Image Contrast Enhancement Algorithm

This example shows how to generate a standalone C library from MATLAB® code that applies a simple histogram equalization function to images to improve image contrast. The example uses `parfor` to process each of the standard three RGB image planes on separate threads. The example also shows how to generate and run a MEX function in MATLAB prior to generating C code to verify that the MATLAB code is suitable for code generation.

MATLAB Coder™ uses the OpenMP portable shared memory parallel programming standard to implement its support for `parfor`. See The OpenMP API Specification for Parallel Programming. Whereas MATLAB supports `parfor` by creating multiple worker sessions, MATLAB Coder uses OpenMP to create multiple threads running on the same machine.

Prerequisites

In order to support parallelization, the compiler must support the OpenMP shared memory parallel programming standard. If your compiler does not have this support, then you can still run this example, but the generated code will run serially.

About the `histequalize` Function

The `histequalize.m` function takes an image (represented as an NxMx3 matrix) and returns an image with enhanced contrast.

```
type histequalize
```

```
function equalizedImage = histequalize(originalImage) %#codegen
% equalizedImage = histequalize(originalImage)
% Histogram equalization (or linearization) for improving image contrast.
% Given an NxMx3 image, equalizes the histogram of each of the three image
% planes in order to improve image contrast.

    assert(size(originalImage,1) <= 8192);
    assert(size(originalImage,2) <= 8192);
    assert(size(originalImage,3) == 3);
    assert(isa(originalImage, 'uint8'));

    [L, originalHist] = computeHistogram(originalImage);
    equalizedImage = equalize(L, originalHist, originalImage);
end

function [L, originalHist] = computeHistogram(originalImage)
    L = double(max(max(max(originalImage)))) + 1;
    originalHist = coder.nullcopy(zeros(3,L));
    sz = size(originalImage);
    N = sz(1);
    M = sz(2);
    parfor plane = 1:sz(3)
        planeHist = zeros(1,L);
        for y = 1:N
            for x = 1:M
                r = originalImage(y,x,plane);
                planeHist(r+1) = planeHist(r+1) + 1;
            end
        end
```

```
        originalHist(plane,:) = planeHist;
    end
end

function equalizedImage = equalize(L, originalHist, originalImage)
    equalizedImage = coder.nullcopy(originalImage);
    sz = size(originalImage);
    N = sz(1);
    M = sz(2);
    normalizer = (L - 1)/(N*M);
    parfor plane = 1:sz(3)
        planeHist = originalHist(plane,:);
        for y = 1:N
            for x = 1:M
                r = originalImage(y,x,plane);
                s = 0;
                for j = 0:int32(r)
                    s = s + planeHist(j+1);
                end
                s = normalizer * s;
                equalizedImage(y,x,plane) = s;
            end
        end
    end
end
```

Generate the MEX Function

Generate a MEX function using the `codegen` command.

```
codegen histequalize
```

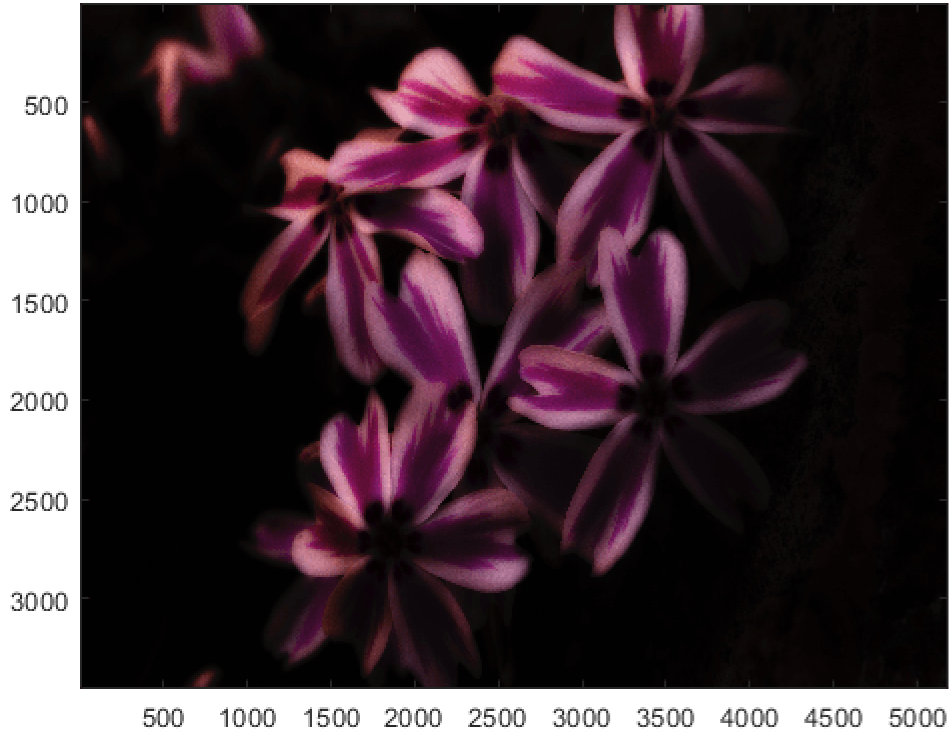
```
Code generation successful.
```

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 `histequalize_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 to read the low-contrast image.

```
lcIm = imread('LowContrast.jpg');
image(lcIm);
```



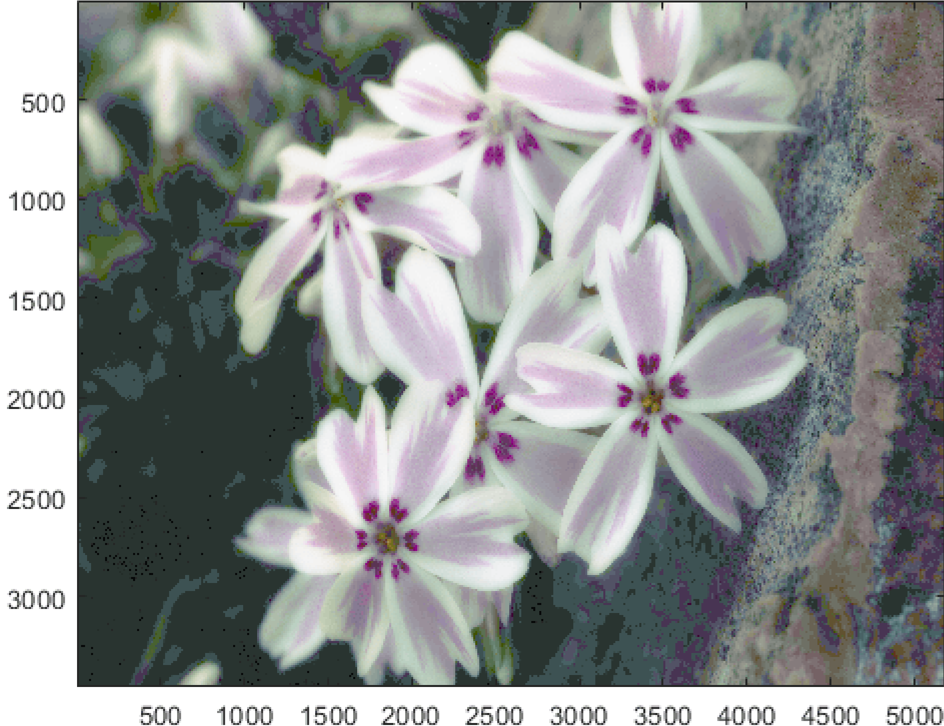
Run the MEX Function (The Histogram Equalization Algorithm)

Pass the low-contrast image.

```
hcIm = histequalize_mex(lcIm);
```

Display the Result

```
image(hcIm);
```



Generate Standalone C Code

```
codegen -config:lib histequalize
```

```
Code generation successful.
```

Using `codegen` with the `-config:lib` option produces a standalone C library. By default, the code generated for the library is in the folder `codegen/lib/histequalize/`.

Inspect the Generated Function

Notice that the generated code contains OpenMP pragmas that control parallelization of the code using multiple threads.

```
type codegen/lib/histequalize/histequalize.c
```

```
/*
 * File: histequalize.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:53:41
 */

/* Include Files */
#include "histequalize.h"
#include "histequalize_data.h"
#include "histequalize_emxutil.h"
#include "histequalize_initialize.h"
```

```
#include "histequalize_types.h"
#include <math.h>
#include <string.h>

/* Function Declarations */
static void computeHistogram(const emxArray_uint8_T *originalImage, double *L,
                             double originalHist_data[],
                             int originalHist_size[2]);

static void equalize(double L, const double originalHist_data[],
                     const emxArray_uint8_T *originalImage,
                     emxArray_uint8_T *equalizedImage);

static double rt_roundd_snf(double u);

/* Function Definitions */
/*
 * Arguments    : const emxArray_uint8_T *originalImage
 *                double *L
 *                double originalHist_data[]
 *                int originalHist_size[2]
 * Return Type  : void
 */
static void computeHistogram(const emxArray_uint8_T *originalImage, double *L,
                             double originalHist_data[],
                             int originalHist_size[2])
{
  double planeHist_data[256];
  int b_i;
  int i;
  int loop_ub;
  int maxval_size_idx_1;
  int npages;
  int p;
  int plane;
  int vlen;
  int x;
  int xOffset;
  int xPageOffset;
  int y;
  short planeHist_size[2];
  unsigned char maxval_data[24576];
  unsigned char b_maxval[3];
  unsigned char maxval;
  unsigned char r;
  maxval_size_idx_1 = originalImage->size[1];
  if (originalImage->size[1] == 0) {
    maxval_size_idx_1 = originalImage->size[1];
    vlen = originalImage->size[1] * 3;
    if (0 <= vlen - 1) {
      memset(&maxval_data[0], 0, vlen * sizeof(unsigned char));
    }
  } else {
    vlen = originalImage->size[0];
    npages = originalImage->size[1] * 3;
    for (p = 0; p < npages; p++) {
      xPageOffset = p * vlen;
      maxval_data[p] = originalImage->data[xPageOffset];
```

```
      for (i = 2; i <= vlen; i++) {
        xOffset = (xPageOffset + i) - 1;
        if (maxval_data[p] < originalImage->data[xOffset]) {
          maxval_data[p] = originalImage->data[xOffset];
        }
      }
    }
  }
  for (p = 0; p < 3; p++) {
    xPageOffset = p * maxval_size_idx_1;
    b_maxval[p] = maxval_data[xPageOffset];
    for (i = 2; i <= maxval_size_idx_1; i++) {
      xOffset = (xPageOffset + i) - 1;
      if (b_maxval[p] < maxval_data[xOffset]) {
        b_maxval[p] = maxval_data[xOffset];
      }
    }
  }
  maxval = b_maxval[0];
  if (b_maxval[0] < b_maxval[1]) {
    maxval = b_maxval[1];
  }
  if (maxval < b_maxval[2]) {
    maxval = b_maxval[2];
  }
  *L = (double)maxval + 1.0;
  originalHist_size[0] = 3;
  originalHist_size[1] = maxval + 1;
  vlen = originalImage->size[0];
  npages = originalImage->size[1];
#pragma omp parallel for num_threads(omp_get_max_threads()) private(           \
    r, planeHist_data, planeHist_size, loop_ub, y, x, b_i)

  for (plane = 0; plane < 3; plane++) {
    loop_ub = (int)*L;
    planeHist_size[1] = (short)*L;
    if (0 <= loop_ub - 1) {
      memset(&planeHist_data[0], 0, loop_ub * sizeof(double));
    }
    for (y = 0; y < vlen; y++) {
      for (x = 0; x < npages; x++) {
        r = originalImage
                ->data[(y + originalImage->size[0] * x) +
                       originalImage->size[0] * originalImage->size[1] * plane];
        b_i = (int)(r + 1U);
        if (r + 1U > 255U) {
          b_i = 255;
        }
        loop_ub = (int)(r + 1U);
        if (r + 1U > 255U) {
          loop_ub = 255;
        }
        planeHist_data[(unsigned char)b_i - 1] =
            planeHist_data[(unsigned char)loop_ub - 1] + 1.0;
      }
    }
    loop_ub = planeHist_size[1];
    for (b_i = 0; b_i < loop_ub; b_i++) {
```

```
      originalHist_data[plane + 3 * b_i] = planeHist_data[b_i];
    }
  }
}

/*
 * Arguments    : double L
 *                const double originalHist_data[]
 *                const emxArray_uint8_T *originalImage
 *                emxArray_uint8_T *equalizedImage
 * Return Type  : void
 */
static void equalize(double L, const double originalHist_data[],
                     const emxArray_uint8_T *originalImage,
                     emxArray_uint8_T *equalizedImage)
{
  double normalizer;
  double s;
  int M;
  int N;
  int i;
  int j;
  int plane;
  int x;
  int y;
  unsigned char r;
  N = equalizedImage->size[0] * equalizedImage->size[1] *
      equalizedImage->size[2];
  equalizedImage->size[0] = originalImage->size[0];
  equalizedImage->size[1] = originalImage->size[1];
  equalizedImage->size[2] = 3;
  emxEnsureCapacity_uint8_T(equalizedImage, N);
  N = originalImage->size[0];
  M = originalImage->size[1];
  normalizer = (L - 1.0) / ((double)(unsigned int)originalImage->size[0] *
                            (double)(unsigned int)originalImage->size[1]);
#pragma omp parallel for num_threads(omp_get_max_threads()) private(s, r, y,   \
                                                                    x, i, j)

  for (plane = 0; plane < 3; plane++) {
    for (y = 0; y < N; y++) {
      for (x = 0; x < M; x++) {
        r = originalImage
                ->data[(y + originalImage->size[0] * x) +
                       originalImage->size[0] * originalImage->size[1] * plane];
        s = 0.0;
        i = r;
        for (j = 0; j <= i; j++) {
          s += originalHist_data[plane + 3 * j];
        }
        s *= normalizer;
        s = rt_roundd_snf(s);
        if (s < 256.0) {
          if (s >= 0.0) {
            r = (unsigned char)s;
          } else {
            r = 0U;
          }
```

```
        } else if (s >= 256.0) {
          r = MAX_uint8_T;
        } else {
          r = 0U;
        }
        equalizedImage
            ->data[(y + equalizedImage->size[0] * x) +
                   equalizedImage->size[0] * equalizedImage->size[1] * plane] =
            r;
      }
    }
  }
}

/*
 * Arguments    : double u
 * Return Type  : double
 */
static double rt_roundd_snf(double u)
{
  double y;
  if (fabs(u) < 4.503599627370496E+15) {
    if (u >= 0.5) {
      y = floor(u + 0.5);
    } else if (u > -0.5) {
      y = u * 0.0;
    } else {
      y = ceil(u - 0.5);
    }
  } else {
    y = u;
  }
  return y;
}

/*
 * equalizedImage = histequalize(originalImage)
 *  Histogram equalization (or linearization) for improving image contrast.
 *  Given an NxMx3 image, equalizes the histogram of each of the three image
 *  planes in order to improve image contrast.
 *
 * Arguments    : const emxArray_uint8_T *originalImage
 *                emxArray_uint8_T *equalizedImage
 * Return Type  : void
 */
void histequalize(const emxArray_uint8_T *originalImage,
                  emxArray_uint8_T *equalizedImage)
{
  double originalHist_data[768];
  double L;
  int originalHist_size[2];
  if (!isInitialized_histequalize) {
    histequalize_initialize();
  }
  computeHistogram(originalImage, &L, originalHist_data, originalHist_size);
  equalize(L, originalHist_data, originalImage, equalizedImage);
}
```

```
/*
 * File trailer for histequalize.c
 *
 * [EOF]
 */
```

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. The algorithm 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 of the simulation on scopes.

For details about this application, see “System Identification Using RLS Adaptive Filtering” (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,cutoffFreq,ff] = ...
    myRLSFilterSystemIDSim(tuningUIStruct)
% myRLSFilterSystemIDSim implements the algorithm used in
% myRLSFilterSystemIDApp.
% This function 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 2013-2017 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

if tuningUIStruct.Reset
    % reset System objects
    reset(rlsFilt);
    reset(unknownSys);
    reset(transferFunctionEstimator);
    reset(sine);
end

% Tune FIR cutoff frequency and RLS forgetting factor
if  tuningUIStruct.ValuesChanged
    param = tuningUIStruct.TuningValues;
    unknownSys.CutoffFrequency  = param(1);
    rlsFilt.ForgettingFactor = param(2);
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;

end
```

myRLSFilterSystemIDApp

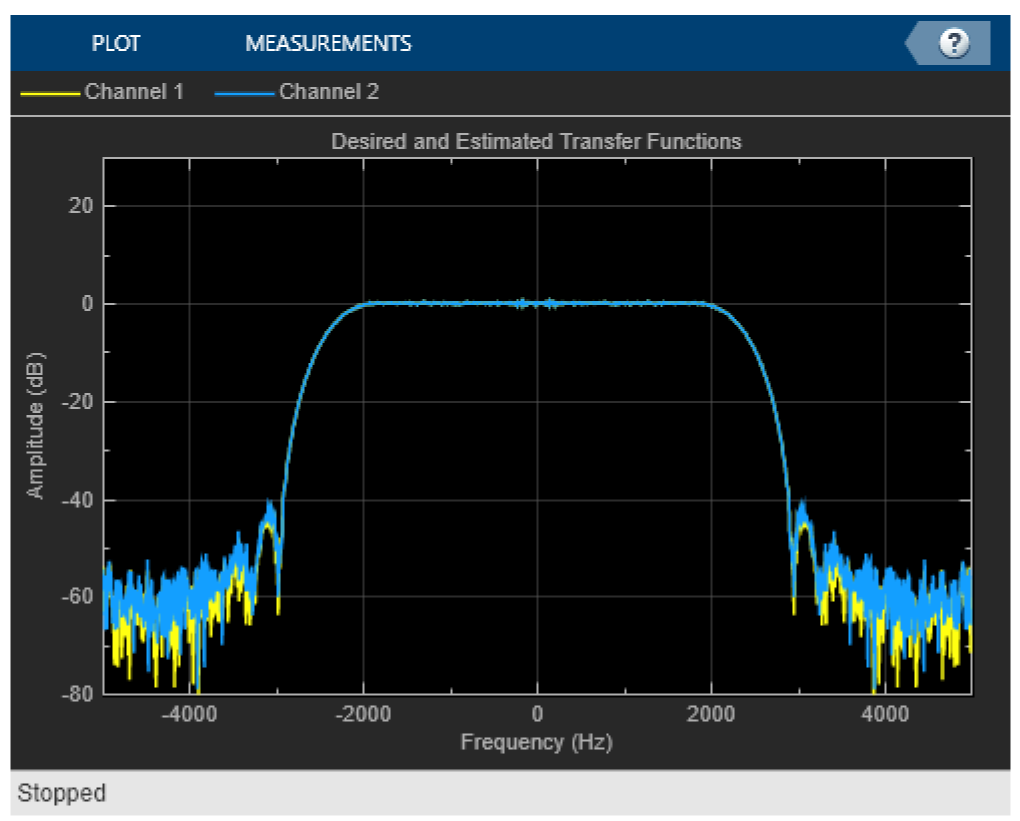
```
function scopeHandles = myRLSFilterSystemIDApp(numTSteps)
% myRLSFilterSystemIDApp initialize and execute RLS Filter
% system identification example. Then, display results using
% scopes. The function returns the handles to the scope and UI objects.
%
% Input:
%   numTSteps - number of time steps
% Outputs:
%   scopeHandles    - Handle to the visualization scopes

% Copyright 2013-2017 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 = timescope('SampleRate',1e4,...
    'Position',[8 184 520 420],...
    'TimeSpanSource','property','TimeSpan',0.01,...
    '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
```

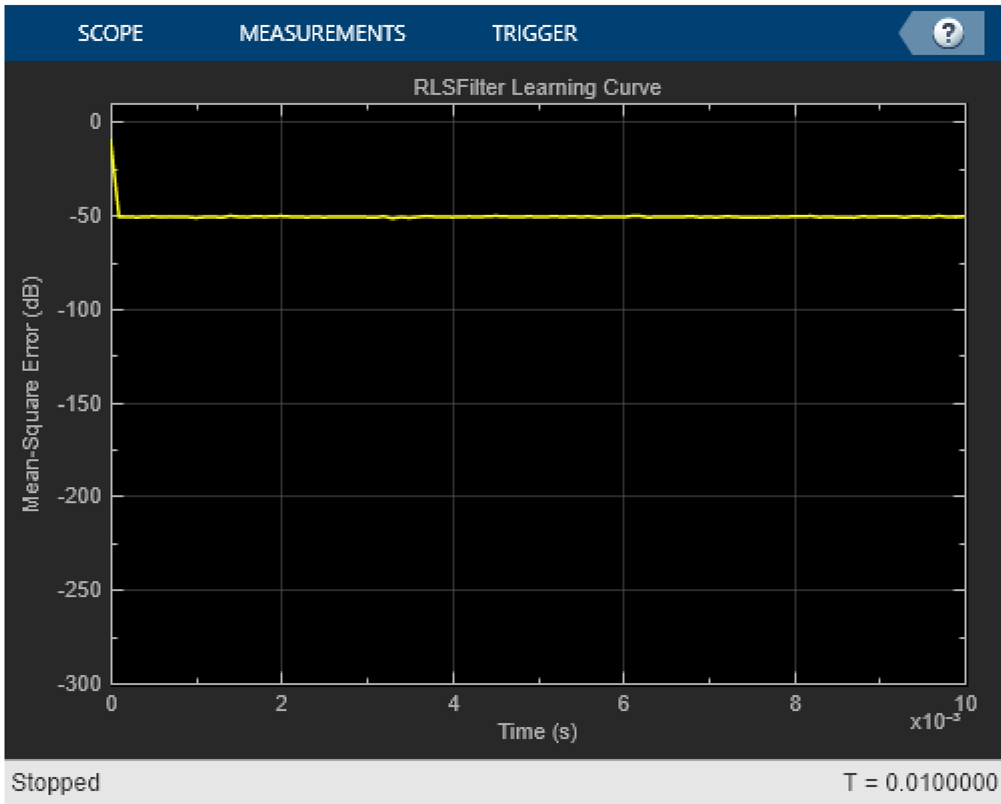
```
%  forgetting factor
Fs = 1e4;
param = struct([]);
param(1).Name = 'Cutoff Frequency (Hz)';
param(1).InitialValue = 0.48 * Fs/2;
param(1).Limits = Fs/2 * [1e-5, .9999];
param(2).Name = 'RLS Forgetting Factor';
param(2).InitialValue = 0.99;
param(2).Limits = [.3, 1];
hUI = HelperCreateParamTuningUI(param, 'RLS FIR Demo');
set(hUI,'Position',[outerSize+32, screen(4)-2*outerSize+8, ...
    outerSize+8, outerSize-92]);

% Execute algorithm
while(numTSteps>=0)

    S = HelperUnpackUIData(hUI);

    drawnow limitrate;   % needed to process UI callbacks

    [tfe,err] = myRLSFilterSystemIDSim(S);

    if S.Stop     % If "Stop Simulation" button is pressed
        break;
    end
    if S.Pause
        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);
```


PLOT
MEASUREMENTS
Channel 1
Channel 2
Desired and Estimated Transfer Functions
Amplitude (dB)
20
0
-20
-40
-60
-80
-4000
-2000
0
2000
4000
Frequency (Hz)
Stopped


SCOPE
MEASUREMENTS
TRIGGER
RLSFilter Learning Curve
Mean-Square Error (dB)
0
-50
-100
-150
-200
-250
-300
0
2
4
6
8
10
x10⁻³
Time (s)
Stopped
T = 0.0100000

Prepare Algorithm for Acceleration

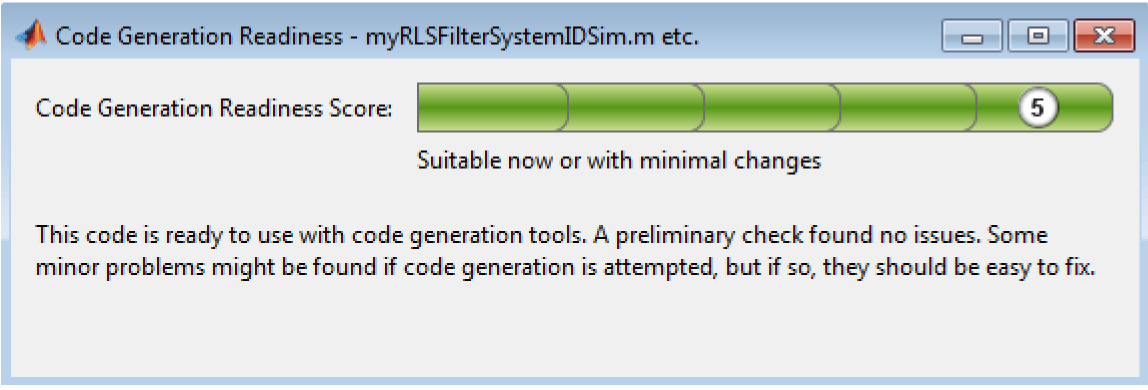
When you use MATLAB Coder to accelerate a MATLAB algorithm, the code must be suitable for code generation.

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');
```



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. For code generation, you must specify the type, size, and complexity of the input arguments. The function `myRLSFilterSystemIDSim` takes a structure that stores tuning information. Define an example tuning structure and pass it to `codegen` by using the `-args` option.

```
ParamStruct.TuningValues = [2400 0.99];
ParamStruct.ValuesChanged = false;
ParamStruct.Reset = false;
ParamStruct.Pause = false;
ParamStruct.Stop  = false;
codegen myRLSFilterSystemIDSim -args {ParamStruct};
```

```
Code generation successful.
```

`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(ParamStruct);
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(ParamStruct);
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: 2.0641 seconds
Time for MEX function: 0.29968 seconds
The MEX function is 6.8879 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” on page 32-6.

For this example, the MEX is sufficiently fast without further optimization.

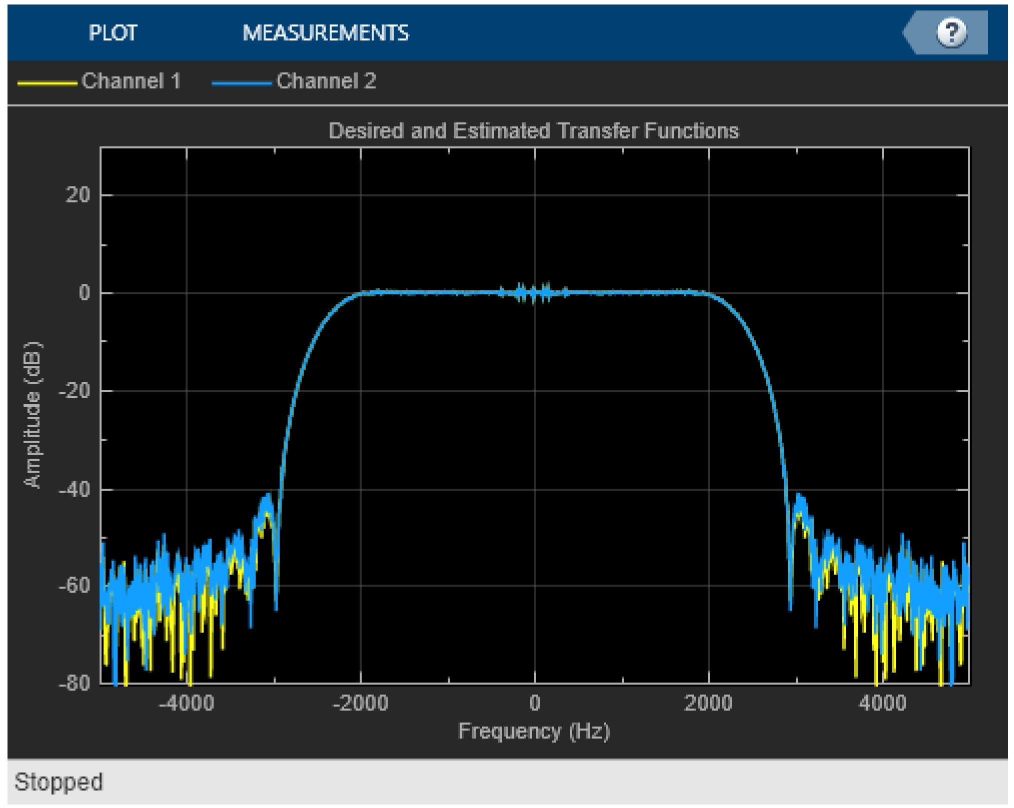
Modify the Application to Call the MEX Function

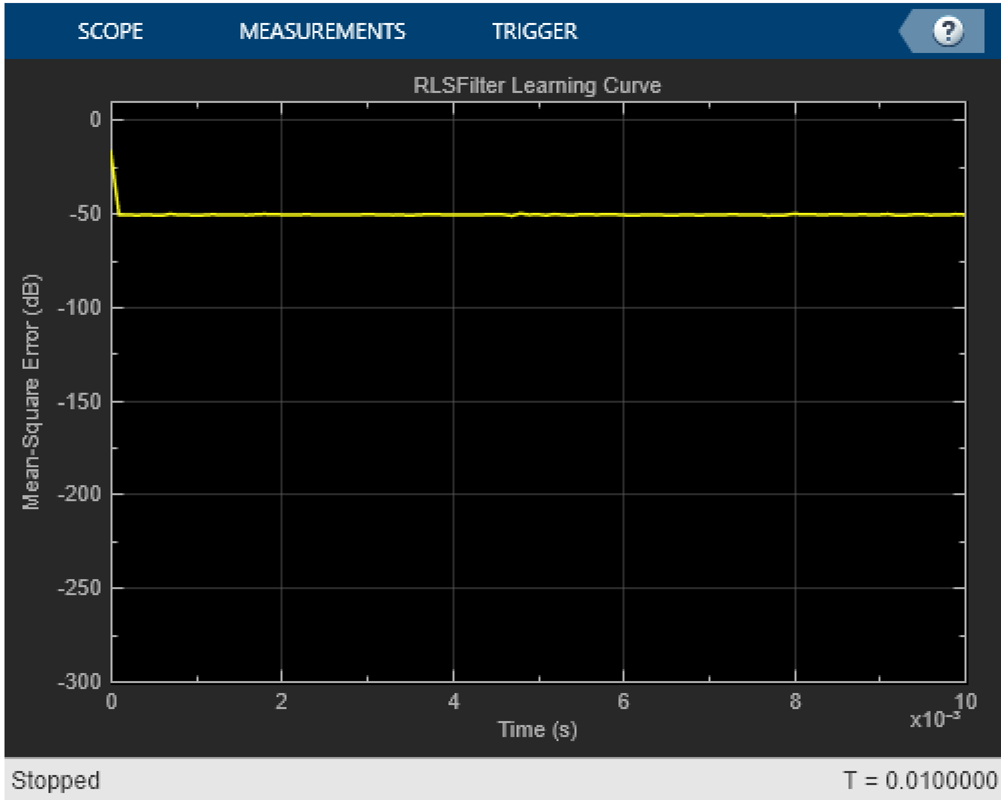
Modify `myRLSFilterSystemIDApp` so that it calls `myRLSFilterSystemIDSim_mex` instead of `myRLSFilterSystemIDSim`.

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);
```


PLOT
MEASUREMENTS
Channel 1
Channel 2
Desired and Estimated Transfer Functions
Amplitude (dB)
20
0
-20
-40
-60
-80
-4000
-2000
0
2000
4000
Frequency (Hz)
Stopped


SCOPE
MEASUREMENTS
TRIGGER
RLSFilter Learning Curve
Mean-Square Error (dB)
0
-50
-100
-150
-200
-250
-300
0
2
4
6
8
10
x10^{-3}
Time (s)
Stopped
T = 0.0100000

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

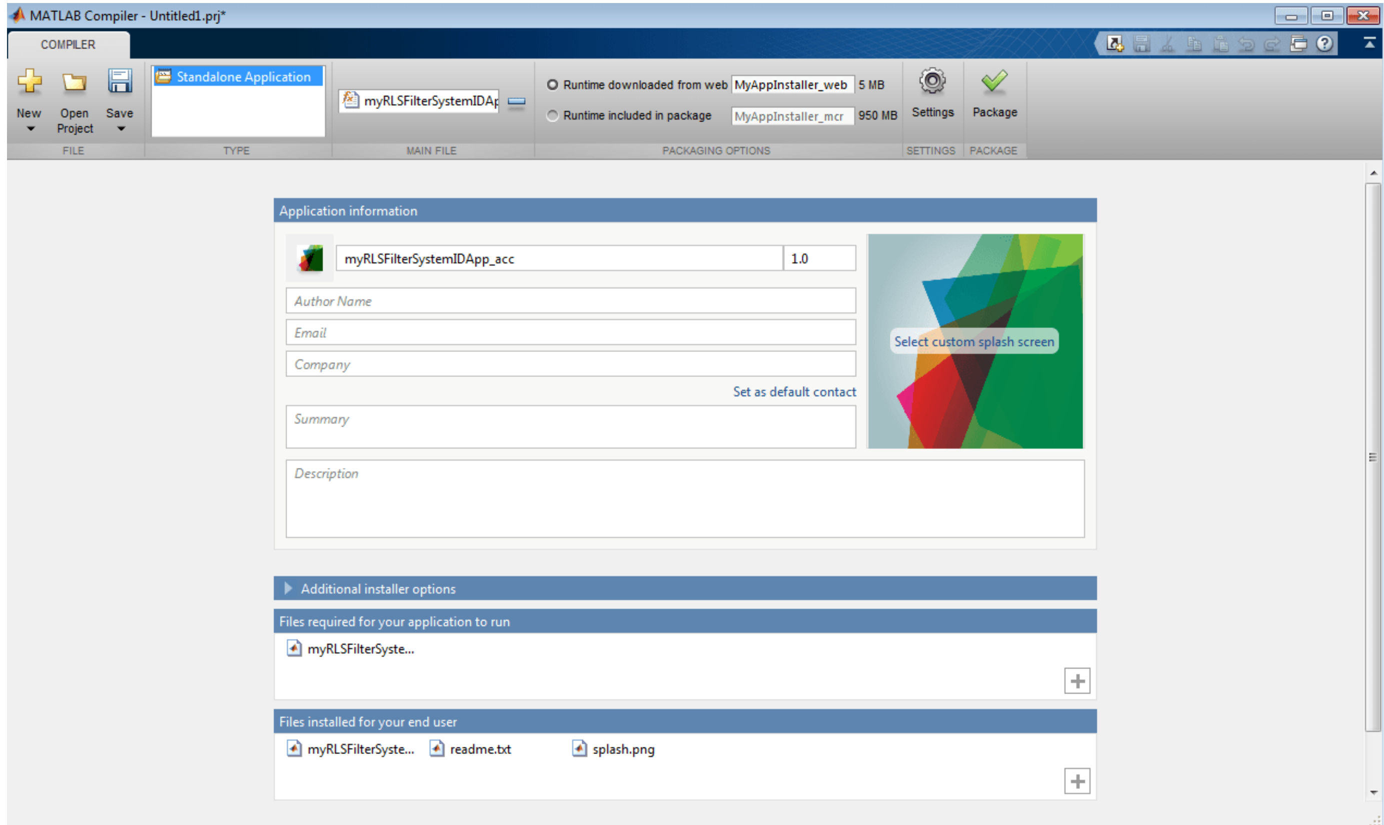
1. To open the Application Compiler App, on the **Apps** tab, under **Application Deployment**, click the app icon.

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.

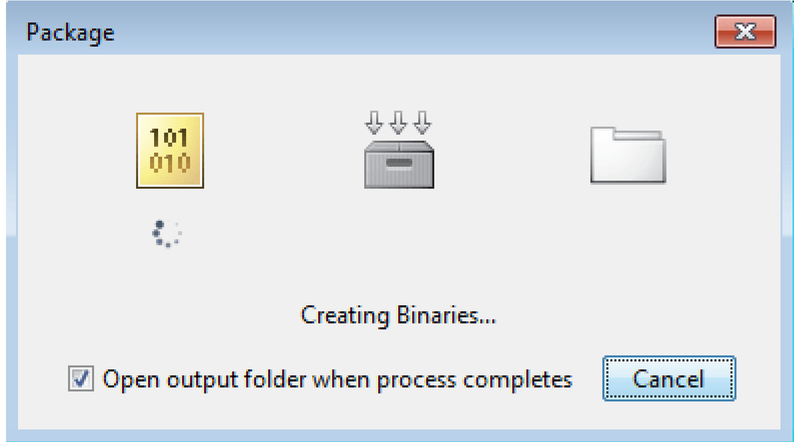
3. In the **Packaging Options** section of the toolstrip, make sure that the **Runtime downloaded from web** check box is selected.

This option creates an application installer that downloads and installs the MATLAB Runtime with the deployed MATLAB application.



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.



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 macOS, navigate to `/Applications/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 macOS, 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.

See Also

More About

- “System Identification Using RLS Adaptive Filtering” (DSP System Toolbox)
- “Workflow for Accelerating MATLAB Algorithms” on page 32-2
- “Accelerate MATLAB Algorithms” on page 32-6
- “Create Standalone Application from MATLAB” (MATLAB Compiler)
- “About the MATLAB Runtime” (MATLAB Compiler)
- MATLAB Compiler Support for MATLAB and toolboxes

External Code Integration

- “Call C/C++ Code from MATLAB Code” on page 33-2
- “Configure Build for External C/C++ Code” on page 33-9
- “Develop Interface for External C/C++ Code” on page 33-12
- “Mapping MATLAB Types to Types in Generated Code” on page 33-15
- “Generate Code to Read a Text File” on page 33-19
- “Generate C/C++ Strings from MATLAB Strings and Character Row Vectors” on page 33-27

Call C/C++ Code from MATLAB Code

In this section...

"Call C Code" on page 33-2
"Return Multiple Values from a C Function" on page 33-3
"Pass Data by Reference" on page 33-4
"Integrate External Code that Uses Custom Data Types" on page 33-5
"Integrate External Code that Uses Pointers, Structures, and Arrays" on page 33-6

From within your MATLAB code, you can directly call external C/C++ code, also called custom code or legacy code. To call C/C++ functions, use `coder.ceval`. The code generator integrates your C/C++ code into the C/C++ code generated from MATLAB. Integrate code when there are external libraries, optimized code, or object files developed in C/C++ that you want to use with your generated code. When the external code uses variable types that are not defined or recognized by MATLAB, use the `coder.opaque` function in conjunction with `coder.ceval`. To reserve certain identifier names for use in your custom C/C++ code that you want to integrate with the generated code, use the `coder.reservedName` function.

Following are some of the primary workflows for external code integration. For more examples, see the `coder.ceval` reference page.

Note By using `coder.ceval`, you gain unrestricted access to external code. Misuse of these functions or errors in your code can destabilize MATLAB and cause it to stop working. To debug your code and analyze error messages from compilation, view the **Build Logs** tab in the code generation report.

Call C Code

This example shows how to integrate a simple C function with MATLAB® code by using `coder.ceval`. Consider the MATLAB function, `mathOps`:

```
function [added, multed] = mathOps(in1, in2)
added = in1+in2;
multed = in1*in2;
end
```

For this example, suppose that you want to implement the addition operation by using external C code. Consider the C function, `adder`, implemented in the file `adder.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "adder.h"

double adder(double in1, double in2)
{
  return in1 + in2;
}
```

To integrate adder with your MATLAB code, you need a header file that contains the function prototype. See the file adder.h:

```
double adder(double in1, double in2);
```

Use the coder.ceval command to call the C function in mathOpsIntegrated.m. Include the header file by using coder.cinclude.

```
function [added, multed] = mathOpsIntegrated(in1, in2)
%#codegen
% for code generation, preinitialize the output variable
% data type, size, and complexity
added = 0;
% generate an include in the C code
coder.cinclude('adder.h');
% evaluate the C function
added = coder.ceval('adder', in1, in2);
multed = in1*in2;
end
```

To generate code, use the codegen command. Specify the source file adder.c as an input. To test the C code, execute the MEX function and inspect the output results.

```
codegen mathOpsIntegrated -args {1, 2} adder.c

[test1, test2] = mathOpsIntegrated_mex(10, 20)
```

```
Code generation successful.


test1 =

    30


test2 =

   200
```

Return Multiple Values from a C Function

The C language restricts functions from returning multiple outputs. Instead, they return only a single, scalar value. The MATLAB functions coder.ref, coder.rref and coder.wref allow you to return multiple outputs from an external C/C++ function.

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 three separate values a, b, and c through a return statement. Instead, create a C function with multiple pointer type arguments and pass the output parameters by reference. For example:

```
void foo(double x,double y,double *a,double *b,double *c)
```

Then you can call the C function from a MATLAB function by using the `coder.ceval` function.

```
coder.ceval('foo',x,y,coder.ref(a),coder.ref(b),coder.ref(c));
```

If your external C function only writes to or only reads from the memory that is passed by reference, you can use the `coder.wref` or `coder.rref` functions instead of `coder.ref`. Under certain circumstances, these functions can enable further optimization of the generated code. When you use `coder.wref(arg)` to pass `arg` by reference, your external C/C++ function must fully initialize the memory referenced by `arg`.

Pass Data by Reference

This example shows how to pass data by reference to and from an external C function.

Pass by reference is an important technique for C/C++ code integration. When you pass data by reference, the program does not need to copy data from one function to another. With pass by value, C code can return only a single scalar variable. With pass by reference, C code can return multiple variables, including arrays.

Consider the MATLAB function `adderRef`. This function uses external C code to add two arrays. The `coder.rref` and `coder.wref` commands instruct the code generator to pass pointers to the arrays, rather than copy them.

```
function out = adderRef(in1, in2)
%#codegen
out = zeros(size(in1));
% the input numel(in1) is converted to integer type
% to match the cAdd function signature
coder.ceval('cAdd', coder.rref(in1), coder.rref(in2), coder.wref(out), int32(numel(in1)) );
end
```

The C code, `cAdd.c`, uses linear indexing to access the elements of the arrays:

```
#include <stdio.h>
#include <stdlib.h>
#include "cAdd.h"

void cAdd(const double* in1, const double* in2, double* out, int numel)
{
    int i;
    for (i=0; i<numel; i++) {
        out[i] = in1[i] + in2[i];
    }
}
```

To build the C code you must provide a header file, `cAdd.h`, with the function signature:

```
void cAdd(const double* in1, const double* in2, double* out, int numel);
```

Test the C code by generating a MEX function and comparing its output with the output from the addition operation in MATLAB.

```
A = rand(2,2)+1;
B = rand(2,2)+10;

codegen adderRef -args {A, B} cAdd.c cAdd.h -report

if (adderRef_mex(A,B) - (A+B) == 0)
    fprintf(['\n' 'adderRef was successful.']);
end
```

```
Code generation successful: To view the report, open('codegen\mex\adderRef\html\report.mldatx').

adderRef was successful.
```

Integrate External Code that Uses Custom Data Types

This example shows how to call a C function that uses data types that are not natively defined within MATLAB®.

For example, if your C code performs file input or output on a C 'FILE *' type, there is no corresponding type within MATLAB. To interact with this data type in your MATLAB code, you must initialize it by using the function `coder.opaque`. In the case of structure types, you can use `coder.cstructname`.

For example, consider the MATLAB function `addCTypes.m`. This function uses `coder.ceval` with input types defined in external code. The function `coder.opaque` initializes the type in MATLAB.

```
function [out] = addCTypes(a,b)
%#codegen
% generate include statements for header files
coder.cinclude('MyStruct.h');
coder.cinclude('createStruct.h');
coder.cinclude('useStruct.h');
% initialize variables before use
in = coder.opaque('MyStruct');
out = 0;
% call C functions
in = coder.ceval('createStruct',a,b);
out = coder.ceval('useStruct',in);
end
```

The `createStruct` function outputs a C structure type:

```
#include <stdio.h>
#include <stdlib.h>
#include "MyStruct.h"
#include "createStruct.h"

struct MyStruct createStruct(double a, double b) {
    struct MyStruct out;
    out.p1 = a;
    out.p2 = b;
```

```
    return out;
}
```

The `useStruct` function performs an operation on the C type:

```
#include "MyStruct.h"
#include "useStruct.h"

double useStruct(struct MyStruct in) {
    return in.p1 + in.p2;
}
```

To generate code, specify the source (.c) files as inputs:

```
codegen addCTypes -args {1,2} -report createStruct.c useStruct.c
```

```
Code generation successful: To view the report, open('codegen\mex\addCTypes\html\report.mldatx')
```

Integrate External Code that Uses Pointers, Structures, and Arrays

This example shows how to integrate external code that operates on a C style array with MATLAB® code. The external code computes a summation over array data. You can customize the code to change the input data or computation.

This example shows how to combine multiple different elements of external code integration functionality. For example, you:

- Interface with an external structure type by using `coder.cstructname`
- Interface with an external pointer type by using `coder.opaque`
- Execute external code by using `coder.ceval`
- Pass data by reference to external code by using `coder.ref`

Explore the Integrated Code

The extSum function uses external C code to compute a summation operation on an array of 32-bit integers. The array size is controlled by a user input.

```
function x = extSum(u)
%#codegen
% set bounds on input type to use static memory allocation
u = int32(u);
assert(0 < u && u < 101);
% initialize an array
temparray = int32(1):u;
% declare an external structure and use it
s = makeStruct(u);
x = callExtCode(s, temparray);
```

To simplify the generated code, you set bounds on the size of the array. The bounds prevents the use of dynamic memory allocation in the generated code.

The function makeStruct declares a MATLAB structure type and initializes one of the fields to a pointer type by using coder.opaque. The C structure corresponding to this definition is contained in a header file that you provide by using the HeaderFile parameter in the coder.cstructname function. The C structure type provides a simple representation for an array of integers.

```
function s = makeStruct(u)
% create structure type based on external header definition
s.numel = u;
s.vals = coder.opaque('int32_T *','NULL');
coder.cstructname(s,'myArrayType','extern','HeaderFile','arrayCode.h');
```

With the external structure type fully initialized, you pass it as an input to the external code in the callExtCode function. This function initializes the array, calls an operation on the array to return a single output, and then frees the initialized memory.

```
function x = callExtCode(s, temparray)
% declare output type
x = int32(0);
% declare external source file
coder.updateBuildInfo('addSourceFiles','arrayCode.c');
% call c code
coder.ceval('arrayInit',coder.ref(s),coder.ref(temparray));
x = coder.ceval('arraySum',coder.ref(s));
coder.ceval('arrayDest',coder.ref(s));
```

The function uses coder.updateBuildInfo to provide the .c file to the code generator.

Generate a MEX Function

To generate a MEX function that you can run and test in MATLAB, enter:

```
codegen extSum -args {10}
```

```
Code generation successful.
```

Test the MEX function. Enter:

```
extSum_mex(10)
```

```
ans =

  int32

   55
```

The external C code, contained in the files arrayCode.c and arrayCode.h, uses the custom type definition int32_T. The generated MEX code produces and uses this custom type definition. If you want to generate standalone (lib, dll, or exe) code that uses this custom data type, then you can

modify the `DataTypeReplacement` property of your configuration object. See “Mapping MATLAB Types to Types in Generated Code” on page 33-15.

See Also

`codegen` | `coder.ceval` | `coder.cinclude` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.reservedName` | `coder.rref` | `coder.wref`

More About

- “Configure Build for External C/C++ Code” on page 33-9
- “Unit Test External C Code with MATLAB Coder” on page 28-33

Configure Build for External C/C++ Code

In this section...

"Provide External Files for Code Generation" on page 33-9
"Configure Build from Within a Function" on page 33-9
"Configure Build by Using the Configuration Object" on page 33-10
"Configure Build by Using the MATLAB Coder App" on page 33-11

To integrate your external C/C++ code with MATLAB, you must provide the external files to the code generator. These files consist of source files, header files, object files, and library files that are used to build the generated code.

You can configure the build at the command line, within a function, or by setting code generation configuration object properties. Specify files at the command line for a quick and simple way to generate code. When you want to preconfigure a function for other projects and code deployments, configure the build within the function. The configuration object provides a standardized set of build properties. You can also specify external files by using the MATLAB Coder App, or by using a class derived from `coder.ExternalDependency`. For more information, see "Develop Interface for External C/C++ Code" on page 33-12.

Provide External Files for Code Generation

Suppose that you want to generate code for a function that uses `coder.ceval` to call the C function `myCFn`. The external source and header files for `myCFn` reside in the folder `C:\custom`. Use this command:

```
codegen myMatlabFn C:\custom\myCFn.c C:\custom\myCFn.h
```

Configure Build from Within a Function

This example shows how to configure the build for external C/C++ code from within a MATLAB® function. Configure the build within a function so that you can more easily integrate it with other projects.

Suppose that you have a top-level MATLAB function, `myFn`:

```
function [out] = myFn(in)
%#codegen
y = mySubFn(in);
out = y + 10;
end
```

This function calls another function, `mySubFn`, that uses the external C code `foo.c`. By using `coder.updateBuildInfo` and `coder.cinclude`, you set all the necessary external code dependencies from within `mySubFn`.

```
function [y] = mySubFn(x)
%#codegen
```

```
coder.cinclude('foo.h');
coder.updateBuildInfo('addSourceFiles', 'foo.c');
% Pre-initialize y to double type.
y = 0;
y = coder.ceval('foo',x);
end
```

You can generate code containing `mySubFn` without needing to configure additional build settings or specify external file inputs at the command line. To generate code for the top-level function `myFn`, enter:

```
codegen myFn -args {5} -report
```

```
Code generation successful: To view the report, open('codegen\mex\myFn\html\report.mldatx').
```

Configure Build by Using the Configuration Object

Customize a build by setting properties of the code generation configuration object. With these properties you can specify external file locations, custom source code, and other build parameters.

Custom Code Property	Description
`CustomHeaderCode`	Specify code to appear near the top of each C/C++ header file generated from your MATLAB code.
`CustomInclude`	Specify a list of include directories to add to the include path when compiling the generated code. Provide an absolute path or a path relative to the project folder. If your folder path name contains spaces, you must enclose it in double quotes: `cfg.CustomInclude = '"C:\Program Files\MATLAB\work"'`
`CustomLibrary`	Specify a list of static library or object files to link with the generated code.
`CustomSource`	Specify a list of source files to compile and link with the generated code. The build process looks for the source files first in the current folder and then in the include folders that you specify in `CustomInclude`.
`CustomSourceCode`	Specify code to appear near the top of the generated C/C++ source file, outside of a function. Do not specify a C static function definition.

For example, declare a standalone code configuration object and specify these properties:

```
cfg = coder.config('lib');
cfg.CustomInclude = 'C:\custom\src C:\custom\lib';
cfg.CustomSource = 'cfunction.c';
cfg.CustomLibrary = 'chelper.obj clibrary.lib';
cfg.CustomSourceCode = '#include "cgfunction.h"';
```

Apply the properties at the command line by using the `codegen` command with the `-config` argument:

```
codegen -config cfg myMatlabFn
```

Configure Build by Using the MATLAB Coder App

1. Open the MATLAB Coder App and proceed to the **Generate Code** step.
2. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow ▼.
3. Click **More Settings**.
4. On the **Custom Code** tab, choose your build configuration settings. Click **Help** to display information about the entry fields.

See Also

`codegen` | `coder.CodeConfig` | `coder.ExternalDependency` | `coder.MexCodeConfig` | `coder.cinclude` | `coder.config` | `coder.updateBuildInfo`

More About

- “Call C/C++ Code from MATLAB Code” on page 33-2
- “Build Process Customization” on page 27-116

Develop Interface for External C/C++ Code

You can develop an interface to external code by using the base class `coder.ExternalDependency`. Using a class for external code can provide certain advantages. You can:

- Place related functions into a single package, without exposing them to the user (encapsulation).
- Create an extensible interface that can be shared across projects.
- Define custom build configuration settings so that build information is preconfigured.

Create a class from coder.ExternalDependency

To instantiate a class derived from the abstract class `coder.ExternalDependency`, you must define the methods `getDescriptiveName`, `isSupportedContext`, and `updateBuildInfo`. These methods address error reporting, build support, and build configuration.

Consider an example of a subclass called `myExternalMathAPI` derived from `coder.ExternalDependency`. This subclass assumes that you have all your needed source and header files contained in your current working folder, with no other dependencies. If you have additional dependencies, such as source, library, or header files, you can redefine `updateBuildInfo`, or derive a subclass from `myExternalMathAPI` which overloads the `updateBuildInfo` method as necessary and adds new methods to the interface. To assist in build configuration, you can use the build information and build context objects accessible by the `updateBuildInfo` method.

```
classdef myExternalMathAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

        % Provide a name for use in error messages
        function bName = getDescriptiveName(~)
            bName = 'myExternalMathAPI';
        end

        % Error out if build context is not supported
        function supported = isSupportedContext(buildContext)
            myTarget = {'mex','rtw'};
            if  buildContext.isCodeGenTarget(myTarget)
                supported = true;
            else
                error('API only supported for mex, lib, exe, dll');
            end
        end

        % Configure simple build in this example
        % Redefine the method as necessary for your dependencies
        function updateBuildInfo(buildInfo, buildContext)
            src = {'extAdd.c','extSub.c','extDiv.c'};
            buildInfo.addSourceFiles(src);
        end

        % Define class methods
        function c = add(a, b)
            coder.cinclude('extAdd.h');
```

```
            c = 0;
            c = coder.ceval('extAdd', a, b);
        end

        function c = subtract(a, b)
            coder.cinclude('extSubtract.h');
            c = 0;
            c = coder.ceval('extSub', a, b);
        end

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

Call the external C/C++ code through the interface:

```
myExternalMathAPI.add(a,b);
myExternalMathAPI.substract(a,b);
myExternalMathAPI.divide(a,b);
```

Best Practices for Using coder.ExternalDependency

Provide an Error Message for Unsupported Build

The `isSupportedContext` method returns true if the external code interface is supported in the build context. If the external code interface is not supported, use `error` to terminate code generation with an error message. For example:

```
function supported = isSupportedContext(buildContext)
    if  buildContext.isMatlabHostTarget()
        supported = true;
    else
        error('MyLibrary is not available for this target');
    end
end
```

Parametrize Methods for MATLAB and Generated Code

Parametrize methods that call external functions so that the methods run in MATLAB. For example:

```
function c = add(a, b)
     if coder.target('MATLAB')
          % running in MATLAB, use built-in addition
          c = a + b;
     else
          % running in generated code, call library function
          c = 0;
          c = coder.ceval('extAdd', a, b);
     end
end
```

Parametrize updateBuildInfo for Multiple Platforms

Parametrize the `updateBuildInfo` method to support multiple platforms. For example, use `coder.BuildConfig.getStdLibInfo` to get the platform-specific library file extensions.

```
function updateBuildInfo(buildInfo, buildContext)
     % Get file extensions for the current platform
     [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

     % Parametrize library extension
     libName =  strcat('myLib', linkLibExt);
     % Other linking parameters
     libPath = 'c:\Link_Objects';
     libPriority = '';
     libPreCompiled = true;
     libLinkOnly = true;

     % Linking command
     buildInfo.addLinkObjects(libName,libPath,libPriority,libPreCompiled,libLinkOnly);
end
```

See Also

`coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.updateBuildInfo` | `error`

More About

- “Build Process Customization” on page 27-116
- “Integrate External/Custom Code” on page 34-39
- “Configure Build for External C/C++ Code” on page 33-9
- “Static Methods”

Mapping MATLAB Types to Types in Generated Code

The code generator produces data types in C/C++ that correspond to the data types that you use in your MATLAB code. The data types that are generated depend on the target platform and compiler. The code generator can produce either built-in C data types, such as `short`, `long`, `int`, and so on, or custom data types defined by using C `typedef` statements. By default, the code generator produces built-in types for standalone code (lib, dll, or exe) and custom types for MEX code. To use built-in C types, modify the `DataTypeReplacement` property of the code generation configuration object or use the MATLAB Coder App. For more information, see “Specify Data Types Used in Generated Code” on page 27-24.

To produce custom C/C++ types, the code generator uses predefined data types in the header file `tmwtypes.h`, located in `fullfile(matlabroot,'extern','include')`. The code generator can also produce custom data types based on analysis of your MATLAB code. Custom data types are defined in the files `rtwtypes.h` and `myFunction_types.h` located in the code generation directory. *`myFunction`* is the name of your top-level function. The code generator cannot produce code for every data type that exists within MATLAB. See “MATLAB Language Features Supported for C/C++ Code Generation” on page 2-24.

When you do not use built-in C data types, the code generator produces these data types:

MATLAB Data Type	Corresponding Custom C/C++ Data Type
`logical`	`boolean_T`
`char`	`char_T`
`string`	`rtString`
`int8`	`int8_T`
`int16`	`int16_T`
`int32`	`int32_T`
`int64`	`int64_T`
`uint8`	`uint8_T`
`uint16`	`uint16_T`
`uint32`	`uint32_T`
`uint64`	`uint64_T`
`single`	`real32_T`
`double`	`real_T`
`complex`	See “Complex Types” on page 33-16.
`struct`	See “Structure Types” on page 33-16.
`fi`	See “Fixed-Point Types” on page 33-16.

When a variable is passed by reference, the corresponding custom data type uses the dereference operator. For example, the corresponding custom C/C++ data type for `int8` when passed by reference is `int8_T*`.

Dynamically allocated arrays map to a custom `emxArray_` type. For example, a dynamically allocated `char` array maps to a type of `emxArray_char_T`. A dynamically allocated double array maps to the type `emxArray_real_T`. Dynamic allocation occurs, for example, when array size is not known at

compile time or when you create a variable-size array by using coder.varsize without specifying explicit upper bounds. For more information on variable-size arrays, see “Use C Arrays in the Generated Function Interfaces” on page 31-3.

Complex Types

In MATLAB, complexity is defined as a property of a data type. This table lists the predefined data types that the code generator uses for MATLAB complex data types.

MATLAB Complex Data Type	Corresponding Custom C/C++ Data Type
int8	cint8_T
int16	cint16_T
int32	cint32_T
int64	cint64_T
uint8	cuint8_T
uint16	cuint16_T
uint32	cuint32_T
uint64	cuint64_T
single	creal32_T
double	creal_T

The code generator defines each complex value as a structure with a real component re and an imaginary component im. For example, see the typedef for creal32_T from tmwtypes.h:

```
typedef struct {
  real32_T re;/* Real component*/
  real32_T im;/* Imaginary component*/
} creal32_T;
```

Suppose you define a variable x of type creal32_T. The generated code accesses 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, for example, by using coder.cstructname. However, you *must* use the names re for the real components and im for the imaginary components in your definitions.

For more information, see “Code Generation for Complex Data” on page 5-3.

Structure Types

MATLAB Coder maps structures to C/C++ types field-by-field. The order of the structure fields in the MATLAB definition is preserved. To control the name of the generated C/C++ structure type, or provide a definition, use the coder.cstructname function. If you are not using dynamic memory allocation, arrays in structures translate into single-dimension arrays, not pointers. For more information, see “Structures”.

Fixed-Point Types

The numerictype properties of a fi object determine its C/C++ data type. By default, the code generator tries to use built-in C/C++ types. However, you can choose to use custom C/C++ data

types instead. The following table shows how the `Signedness`, `WordLength`, and `FractionLength` properties determine the custom C/C++ data type. The custom 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	Corresponding Custom C/C++ Data Type
1	8	7	`int8_T`
1	13	10	`int16_T`
1	16	15	`int16_T`
0	19	15	`uint32_T`

Character Vectors

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, generated code that expects a string can stop working without compiler errors or warnings.

Multiword Types

Multiword types are custom types that are generated when the target hardware cannot store your MATLAB data type in a built-in C/C++ type. Multiword types are generated as C/C++ 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 a `long` with a size of 64-bits, the code generator produces a structure with an array of two 64-bit long types.

```
typedef struct
{
  unsigned long chunks[2];
} uint128m_T;
```

The C/C++ data type generated from a 64-bit integer MATLAB type depends on the sizes of the integer types on the target hardware. If a built-in type wide enough to store 64-bits does not exist, then the 64-bit MATLAB Coder type maps to a custom multiword type.

See Also

`coder.cstructname` | `coder.opaque`

More About

- “Fundamental MATLAB Classes”
- “Integrate External Code that Uses Custom Data Types” on page 33-5

Generate Code to Read a Text File

This example shows how to generate a standalone C library from MATLAB® code that reads a file from disk using the functions fopen/fread/fclose.

About the `readfile` Function

The `readfile.m` function takes a file name (or path) as input and returns a string containing the contents of the file.

```
type readfile
```

```
% y = readfile(filename)
% Read file 'filename' and return a MATLAB string with the contents
% of the file.
function y = readfile(filename) %#codegen

% Put class and size constraints on function input.
assert(isa(filename, 'char'));
assert(size(filename, 1) == 1);
assert(size(filename, 2) <= 1024);

% Call fopen(filename 'r'), but we need to convert the MATLAB
% string into a C type string (which is the same string with the
% NUL (\0) string terminator).
f = fopen(filename, 'r');

% Call fseek(f, 0, SEEK_END) to set file position to the end of
% the file.
fseek(f, 0, 'eof');

% Call ftell(f) which will return the length of the file in bytes
% (as current file position is at the end of the file).
filelen = int32(ftell(f));

% Reset current file position
fseek(f,0,'bof');

% Initialize a buffer
maxBufferSize = int32(2^16);
buffer = zeros(1, maxBufferSize,'uint8');

% Remaining is the number of bytes to read (from the file)
remaining = filelen;

% Index is the current position to read into the buffer
index = int32(1);

while remaining > 0
    % Buffer overflow?
    if remaining + index > size(buffer,2)
        fprintf('Attempt to read file which is bigger than internal buffer.\n');
        fprintf('Current buffer size is %d bytes and file size is %d bytes.\n', maxBufferSize, fi
        break
    end
    % Read as much as possible from the file into internal buffer
```

```
    [dataRead, nread] = fread(f,remaining, 'char');
    buffer(index:index+nread-1) = dataRead;
    n = int32(nread);
    if n == 0
        % Nothing more to read
        break;
    end
    % Did something went wrong when reading?
    if n < 0
        fprintf('Could not read from file: %d.\n', n);
        break;
    end
    % Update state variables
    remaining = remaining - n;
    index = index + n;
end

% Close file
fclose(f);

y = char(buffer(1:index));
```

Generate the MEX Function for Testing

Generate a MEX function using the `codegen` command.

```
codegen readfile
```

```
Code generation successful.
```

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 `readfile_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Run the MEX Function

Call the generated MEX function and display the size of the returned string and its first 100 characters.

```
y = readfile_mex('readfile.m');
size(y)
```

```
ans = 1×2

           1        1857
```

```
y(1:100)
```

```
ans =
    '% y = readfile(filename)
     % Read file 'filename' and return a MATLAB string with the contents
     % of th'
```

Generate C Code

```
codegen -config:lib readfile
```

```
Code generation successful.
```

Using `codegen` with the specified `-config cfg` option produces a standalone C library.

Inspect the Generated Code

By default, the code generated for the library is in the folder `codegen/lib/readfile/`.

The files are:

```
dir codegen/lib/readfile/
```

```
.                         readfile.lib              readfile_rtw.rsp
..                        readfile.obj              readfile_rtw_comp.rsp
.gitignore                readfile_data.c           readfile_rtw_ref.rsp
_clang-format             readfile_data.h           readfile_rtwutil.c
buildInfo.mat             readfile_data.obj         readfile_rtwutil.h
codeInfo.mat              readfile_emxAPI.c         readfile_rtwutil.obj
codedescriptor.dmr        readfile_emxAPI.h         readfile_terminate.c
compileInfo.mat           readfile_emxAPI.obj       readfile_terminate.h
defines.txt               readfile_emxutil.c        readfile_terminate.obj
examples                  readfile_emxutil.h        readfile_types.h
fileManager.c             readfile_emxutil.obj      rtw_proj.tmw
fileManager.h             readfile_initialize.c     rtwtypes.h
fileManager.obj           readfile_initialize.h     setup_msvc.bat
interface                 readfile_initialize.obj
readfile.c                readfile_rtw.bat
readfile.h                readfile_rtw.mk
```

Inspect the C Code for the `readfile.c` Function

```
type codegen/lib/readfile/readfile.c
```

```
/*
 * File: readfile.c
 *
 * MATLAB Coder version            : 5.2
 * C/C++ source code generated on  : 23-Feb-2021 16:51:27
 */

/* Include Files */
#include "readfile.h"
#include "fileManager.h"
#include "readfile_data.h"
#include "readfile_emxutil.h"
#include "readfile_initialize.h"
#include "readfile_rtwutil.h"
#include "readfile_types.h"
#include <stddef.h>
#include <stdio.h>
#include <string.h>

/* Function Definitions */
/*
 * Put class and size constraints on function input.
 *
 * Arguments    : const char filename_data[]
 *                const int filename_size[2]
 *                emxArray_char_T *y
```

```
 * Return Type  : void
 */
void readfile(const char filename_data[], const int filename_size[2],
              emxArray_char_T *y)
{
  FILE *filestar;
  int wherefrom;
  long position_t;
  size_t nBytes;
  size_t numReadSizeT;
  emxArray_uint8_T *At;
  emxArray_uint8_T *b_At;
  double position;
  int b_index;
  int bytesOut;
  int c;
  int i;
  int num2Read;
  int numRead;
  int other2Read;
  int remaining;
  short bdims_idx_0;
  unsigned char buffer[65536];
  char tbuf[1024];
  signed char fileid;
  boolean_T doEOF;
  boolean_T exitg1;
  if (!isInitialized_readfile) {
    readfile_initialize();
  }
  /*  y = readfile(filename) */
  /*  Read file 'filename' and return a MATLAB string with the contents */
  /*  of the file. */
  /*  Call fopen(filename 'r'), but we need to convert the MATLAB */
  /*  string into a C type string (which is the same string with the */
  /*  NUL (\0) string terminator). */
  fileid = cfopen(filename_data, filename_size, "rb");
  /*  Call fseek(f, 0, SEEK_END) to set file position to the end of */
  /*  the file. */
  wherefrom = SEEK_END;
  filestar = fileManager(fileid);
  if ((fileid == 0) || (fileid == 1) || (fileid == 2)) {
    filestar = NULL;
  }
  if (!(filestar == NULL)) {
    fseek(filestar, (long int)0.0, wherefrom);
  }
  /*  Call ftell(f) which will return the length of the file in bytes */
  /*  (as current file position is at the end of the file). */
  filestar = fileManager(fileid);
  if ((fileid == 0) || (fileid == 1) || (fileid == 2)) {
    filestar = NULL;
  }
  if (filestar == NULL) {
    position = -1.0;
  } else {
    position_t = ftell(filestar);
    position = (double)position_t;
```

```
}
position = rt_roundd_snf(position);
if (position < 2.147483648E+9) {
  if (position >= -2.147483648E+9) {
    i = (int)position;
  } else {
    i = MIN_int32_T;
  }
} else if (position >= 2.147483648E+9) {
  i = MAX_int32_T;
} else {
  i = 0;
}
/*  Reset current file position */
wherefrom = SEEK_SET;
filestar = fileManager(fileid);
if ((fileid == 0) || (fileid == 1) || (fileid == 2)) {
  filestar = NULL;
}
if (!(filestar == NULL)) {
  fseek(filestar, (long int)0.0, wherefrom);
}
/*  Initialize a buffer */
memset(&buffer[0], 0, 65536U * sizeof(unsigned char));
/*  Remaining is the number of bytes to read (from the file) */
remaining = i;
/*  Index is the current position to read into the buffer */
b_index = 1;
emxInit_uint8_T(&At, 2);
emxInit_uint8_T(&b_At, 1);
exitg1 = false;
while ((!exitg1) && (remaining > 0)) {
  /*  Buffer overflow? */
  if (b_index > MAX_int32_T - remaining) {
    other2Read = MAX_int32_T;
  } else {
    other2Read = remaining + b_index;
  }
  if (other2Read > 65536) {
    printf("Attempt to read file which is bigger than internal buffer.\n");
    fflush(stdout);
    printf("Current buffer size is %d bytes and file size is %d bytes.\n",
           65536, i);
    fflush(stdout);
    exitg1 = true;
  } else {
    /*  Read as much as possible from the file into internal buffer */
    if (remaining >= MAX_int32_T) {
      c = 1024;
      doEOF = true;
    } else {
      c = remaining;
      doEOF = false;
    }
    nBytes = sizeof(char);
    filestar = fileManager(fileid);
    if ((fileid == 0) || (fileid == 1) || (fileid == 2)) {
      filestar = NULL;
```

```
    }
    if (!doEOF) {
      if (filestar == NULL) {
        b_At->size[0] = 0;
        bytesOut = 0;
      } else {
        numRead = b_At->size[0];
        b_At->size[0] = remaining;
        emxEnsureCapacity_uint8_T(b_At, numRead);
        if (c > 1024) {
          bdims_idx_0 = 1024;
        } else {
          bdims_idx_0 = (short)c;
        }
        bytesOut = 0;
        numRead = 1;
        while ((bytesOut < c) && (numRead > 0)) {
          num2Read = bdims_idx_0;
          other2Read = c - bytesOut;
          if (bdims_idx_0 > other2Read) {
            num2Read = other2Read;
          }
          numRead = 0;
          other2Read = 1;
          while ((numRead < num2Read) && (other2Read > 0)) {
            numReadSizeT =
                fread(&tbuf[numRead], nBytes, num2Read - numRead, filestar);
            other2Read = (int)numReadSizeT;
            numRead += (int)numReadSizeT;
          }
          for (other2Read = 0; other2Read < numRead; other2Read++) {
            b_At->data[other2Read + bytesOut] =
                (unsigned char)tbuf[other2Read];
          }
          bytesOut += numRead;
        }
        numRead = bytesOut + 1;
        num2Read = b_At->size[0];
        for (other2Read = numRead; other2Read <= num2Read; other2Read++) {
          b_At->data[other2Read - 1] = 0U;
        }
        if (bytesOut < remaining) {
          numRead = b_At->size[0];
          if (1 > bytesOut) {
            b_At->size[0] = 0;
          } else {
            b_At->size[0] = bytesOut;
          }
          emxEnsureCapacity_uint8_T(b_At, numRead);
        }
      }
    } else {
      At->size[0] = 0;
      At->size[1] = 1;
      if (filestar == NULL) {
        bytesOut = 0;
      } else {
        c = 1;
```

```
      bytesOut = 0;
      while (c > 0) {
        c = 0;
        numRead = 1;
        while ((c < 1024) && (numRead > 0)) {
          numReadSizeT = fread(&tbuf[c], nBytes, 1024 - c, filestar);
          numRead = (int)numReadSizeT;
          c += (int)numReadSizeT;
        }
        if (1 > c) {
          other2Read = 0;
        } else {
          other2Read = c;
        }
        numRead = b_At->size[0];
        b_At->size[0] = At->size[0] + other2Read;
        emxEnsureCapacity_uint8_T(b_At, numRead);
        num2Read = At->size[0];
        for (numRead = 0; numRead < num2Read; numRead++) {
          b_At->data[numRead] = At->data[numRead];
        }
        for (numRead = 0; numRead < other2Read; numRead++) {
          b_At->data[numRead + At->size[0]] = (unsigned char)tbuf[numRead];
        }
        numRead = At->size[0] * At->size[1];
        At->size[0] = b_At->size[0];
        At->size[1] = 1;
        emxEnsureCapacity_uint8_T(At, numRead);
        other2Read = b_At->size[0];
        for (numRead = 0; numRead < other2Read; numRead++) {
          At->data[numRead] = b_At->data[numRead];
        }
        bytesOut += c;
      }
    }
    numRead = b_At->size[0];
    b_At->size[0] = At->size[0];
    emxEnsureCapacity_uint8_T(b_At, numRead);
    other2Read = At->size[0];
    for (numRead = 0; numRead < other2Read; numRead++) {
      b_At->data[numRead] = At->data[numRead];
    }
  }
  position = (double)b_index + (double)bytesOut;
  if (position < 2.147483648E+9) {
    if (position >= -2.147483648E+9) {
      numRead = (int)position;
    } else {
      numRead = MIN_int32_T;
    }
  } else {
    numRead = MAX_int32_T;
  }
  if (b_index > numRead - 1) {
    num2Read = -1;
    numRead = -1;
  } else {
    num2Read = b_index - 2;
```

```
        numRead -= 2;
      }
      other2Read = numRead - num2Read;
      for (numRead = 0; numRead < other2Read; numRead++) {
        buffer[(num2Read + numRead) + 1] = b_At->data[numRead];
      }
      if (bytesOut == 0) {
        /*  Nothing more to read */
        exitg1 = true;

        /*  Did something went wrong when reading? */
      } else if (bytesOut < 0) {
        printf("Could not read from file: %d.\n", bytesOut);
        fflush(stdout);
        exitg1 = true;
      } else {
        /*  Update state variables */
        remaining -= bytesOut;
        if ((b_index < 0) && (bytesOut < MIN_int32_T - b_index)) {
          b_index = MIN_int32_T;
        } else if ((b_index > 0) && (bytesOut > MAX_int32_T - b_index)) {
          b_index = MAX_int32_T;
        } else {
          b_index += bytesOut;
        }
      }
    }
  }
  emxFree_uint8_T(&b_At);
  emxFree_uint8_T(&At);
  /*  Close file */
  cfclose(fileid);
  i = y->size[0] * y->size[1];
  y->size[0] = 1;
  y->size[1] = b_index;
  emxEnsureCapacity_char_T(y, i);
  for (i = 0; i < b_index; i++) {
    y->data[i] = (signed char)buffer[i];
  }
}

/*
 * File trailer for readfile.c
 *
 * [EOF]
 */
```

Generate C/C++ Strings from MATLAB Strings and Character Row Vectors

By default, MATLAB strings and character row vectors are mapped to C/C++ character arrays in the generated code. To generate C/C++ strings from MATLAB strings or character row vectors, the MATLAB string or character row vector must be null-terminated (end with zero, `0`). For example, the string `"Hello World"+char(0)` and character row vector `['Hello World', 0]` are null-terminated.

If a MATLAB string or character row vector is not null-terminated, for example `'Hello World'`, the MATLAB string is mapped to character arrays `{ 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd' }` in the generated C/C++ code.

In MATLAB, consider this function:

```
function t = CharArrayNullAtEnd()
    t = ['Hello World',0];
end
```

The corresponding C/C++ code generated for this function is:

```
void CharArrayNullAtEnd(char t[12])
{
  int i;
  static const char cv[12] = "Hello World";
  for (i = 0; i < 12; i++) {
    t[i] = cv[i];
  }
}
```

Generating C/C++ strings instead of character arrays improves the readability of the generated code.

Note If the length of the characters is less than the `LoopUnrollThreshold`, a double quoted C/C++ string is not generated in the code even if it is null-terminated. Instead, the code generator produces a C character array that has individual character assignments. By default, the assigned value to `LoopUnrollThreshold` is 5. For more information on loop unrolling, see “Unroll for-Loops” on page 34-33.

Add New Line to Strings in Generated Code

When you generate C/C++ strings from null-terminated MATLAB strings or a character row vector, use the `newline` function in the MATLAB string or character row vector. The code generator maps `newline` function to newline character `'\n'` in the generated code. If you use the character `'\n'` in the MATLAB code instead, it is escaped and is mapped to `'\\n'` in the generated code.

In MATLAB, consider this function:

```
function StringNewline()

    string1 = ['Hello World' 0];
    string2 = ['My MATLAB' 0];
    formatSpecifier = ['%s' newline 0];
    coder.cinclude('<stdio.h>');
    coder.ceval('printf',coder.rref(formatSpecifier),coder.rref(string1));
    coder.ceval('printf',coder.rref(formatSpecifier),coder.rref(string2));
```

```
end
```

The corresponding C/C++ code generated for this function is:

```
void StringNewline(const emlrtStack *sp)
{
  static const char_T formatSpecifier[4] = "%s\n";
  static const char_T string1[12] = "Hello World";
  static const char_T string2[14] = "My MATLAB";
  (void)sp;
  printf(formatSpecifier, string1);
  printf(formatSpecifier, string2);
}
```

In the MATLAB function `StringNewline`, if `formatSpecifier` is `'%s\n'` instead of `['%s' newline 0]`, then the character `'\n'` is escaped and you have `{'\\','n'}` in the generated C/C++ code.

Limitations

A MATLAB character row vector that has multiple nulls, for example `['Hello', 0, 0]`, is not supported for C/C++ string generation.

See Also

`codegen` | `coder` | `coder.ceval` | `coder.config` | `coder.rref` | `newline`

More About

- "Code Generation for Strings" on page 5-11
- "Loop unrolling threshold" (Simulink Coder)

Generate Efficient and Reusable Code

- "Optimization Strategies" on page 34-3
- "Modularize MATLAB Code" on page 34-5
- "Avoid Data Copies of Function Inputs in Generated Code" on page 34-6
- "Inline Code" on page 34-8
- "Control Inlining to Fine-Tune Performance and Readability of Generated Code" on page 34-9
- "Fold Function Calls into Constants" on page 34-14
- "Control Stack Space Usage" on page 34-15
- "Stack Allocation and Performance" on page 34-18
- "Dynamic Memory Allocation and Performance" on page 34-19
- "Minimize Dynamic Memory Allocation" on page 34-20
- "Provide Maximum Size for Variable-Size Arrays" on page 34-21
- "Disable Dynamic Memory Allocation During Code Generation" on page 34-25
- "Set Dynamic Memory Allocation Threshold" on page 34-26
- "Excluding Unused Paths from Generated Code" on page 34-28
- "Prevent Code Generation for Unused Execution Paths" on page 34-29
- "Generate Code with Parallel for-Loops (parfor)" on page 34-31
- "Minimize Redundant Operations in Loops" on page 34-32
- "Unroll for-Loops" on page 34-33
- "Disable Support for Integer Overflow or Nonfinites" on page 34-37
- "Integrate External/Custom Code" on page 34-39
- "MATLAB Coder Optimizations in Generated Code" on page 34-43
- "Use coder.const with Extrinsic Function Calls" on page 34-46
- "memcpy Optimization" on page 34-48
- "memset Optimization" on page 34-49
- "Reuse Large Arrays and Structures" on page 34-50
- "LAPACK Calls in Generated Code" on page 34-51
- "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" on page 34-52
- "BLAS Calls in Generated Code" on page 34-55
- "Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls" on page 34-56
- "Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls" on page 34-60
- "Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code" on page 34-63
- "Speed Up MEX Generation by Using JIT Compilation" on page 34-67
- "Automatically Parallelize for Loops in Generated Code" on page 34-69

- “Resolve Issue: Array or Variable Access Pattern Not Suitable for Parallel Execution” on page 34-73

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 34-43.

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 34-31 "Automatically Parallelize for Loops in Generated Code" on page 34-69
You have variable-size arrays in your MATLAB code.	"Minimize Dynamic Memory Allocation" on page 34-20
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 34-26
You want your generated function to be called by reference.	"Avoid Data Copies of Function Inputs in Generated Code" on page 34-6
You are calling small functions in your MATLAB code.	"Inline Code" on page 34-8
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	"Control Inlining to Fine-Tune Performance and Readability of Generated Code" on page 34-9
You do not want to generate code for expressions that contain constants only.	"Fold Function Calls into Constants" on page 34-14
You have loop operations in your MATLAB code that do not depend on the loop index.	"Minimize Redundant Operations in Loops" on page 34-32
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 34-37
You know beforehand that `Inf`s and `NaN`s do not occur during execution of your generated code.	"Disable Support for Nonfinite Numbers" on page 34-37
You have `for`-loops with few iterations.	"Unroll for-Loops" on page 34-33
You already have legacy C/C++ code optimized for your target environment.	"Integrate External/Custom Code" on page 34-39
You want to speed up the code generated for basic vector and matrix functions.	"Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls" on page 34-56

Condition	Action
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 34-52
You want to speed up the code generated for fast fourier transform (FFT) functions.	"Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls" on page 34-60

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 34-29
You want your generated function to be called by reference.	"Avoid Data Copies of Function Inputs in Generated Code" on page 34-6
You have limited stack space for your generated code.	"Control Stack Space Usage" on page 34-15
You are calling small functions in your MATLAB code.	"Inline Code" on page 34-8
You have limited target memory for your generated code. You want to inline small functions and generate separate code for larger ones.	"Control Inlining to Fine-Tune Performance and Readability of Generated Code" on page 34-9
You do not want to generate code for expressions that contain constants only.	"Fold Function Calls into Constants" on page 34-14
You have loop operations in your MATLAB code that do not depend on the loop index.	"Minimize Redundant Operations in Loops" on page 34-32
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 34-37
You know beforehand that `Inf`-s and `NaN`-s does not occur during execution of your generated code.	"Disable Support for Nonfinite Numbers" on page 34-37
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 34-50

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`.

Avoid Data Copies of Function Inputs in Generated Code

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 the 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;
}
...
```

See Also

Related Examples

- “Pass Structure Arguments by Reference or by Value in Generated Code” on page 27-122

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`.

See Also

More About

- “Control Inlining to Fine-Tune Performance and Readability of Generated Code” on page 34-9

Control Inlining to Fine-Tune Performance and Readability of Generated Code

Inlining is an optimization technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, thereby improving speed.

Depending on your application, too much code inlining can also have certain disadvantages:

- Inlining can produce larger C/C++ code and reduce code readability. For example, suppose that you call a certain function `foo` many times in your source MATLAB code. If the code generator always inlines `foo`, the generated code size increases because `foo` is inlined every time it is called. However, for this to happen, the call sites must be different. For example, inlining does not lead to large code size if `foo` is called several times inside a loop.
- For out-of-line functions, stack space for variables local to the function is released when the function returns. For inlined functions, stack space remains occupied by the local variables even when the function returns. So, if you have limited RAM or stack space, you might want to restrict function inlining.

The code generator uses internal heuristics to determine whether to inline functions in the generated code. This help topic explains how to fine-tune these heuristics and generate code that meets the speed, readability, and stack space requirements of your application.

Control Inlining of a Specific MATLAB Function

To instruct the code generator to either always or never inline a certain MATLAB function, use the `coder.inline('always')` and `coder.inline('never')` directives inside the body of that function. To learn more about these directives, see `coder.inline`.

Control Inlining by Using Code Generation Settings

You might have different speed and readability requirements for the code generated for functions that you write and the code generated for MathWorks functions. Certain code generation settings enable you to separately control the inlining behavior for these two parts of the generated code base and at the boundary between them. These settings apply to both MEX and standalone code generation.

Code Configuration Parameter	Description	Options
In a code configuration object: `InlineBetweenUserFunctions` In the MATLAB Coder app: On the **All Settings** tab, **Inline between user functions**	Controls inlining behavior at all call sites where a function that you wrote calls another function that you wrote	`'Always'` | `'Speed'` (default) | `'Readability'` | `'Never'`

Code Configuration Parameter	Description	Options
In a code configuration object: `InlineBetweenMathWorksFunctions` In the MATLAB Coder app: On the **All Settings** tab, **Inline between MathWorks functions**	Controls inlining behavior at all call sites where a MathWorks function calls another MathWorks function	`'Always'` | `'Speed'` (default) | `'Readability'` | `'Never'`
In a code configuration object: `InlineBetweenUserAndMathWorksFunctions` In the MATLAB Coder app: On the **All Settings** tab, **Inline between user and MathWorks functions**	Controls inlining behavior at all call sites where a function that you wrote calls a MathWorks function, or a MathWorks function calls a function that you wrote	`'Always'` | `'Speed'` (default) | `'Readability'` | `'Never'`

Option descriptions:

- `'Always'`: Always performs inlining at a call site.
- `'Speed'`: Uses internal heuristics to determine whether to perform inlining at a call site. This setting usually leads to highly optimized code. This setting is the default setting.
- `'Readability'`: Almost never inlines function calls, except for calls to very small functions. Preserves modularity of code without sacrificing too much speed, whenever possible. Results in highly readable code.
- `'Never'`: Never inlines function calls. Results in maximum readability. This setting might significantly reduce the performance of the generated code.

Note In certain cases, the code generator might not strictly follow the option you choose for an inlining parameter. For example, if the body of a MathWorks function contains the `coder.inline('never')` directive and you set `InlineBetweenMathWorksFunctions` to `'Always'`, the code generator gives preference to the `coder.inline` directive and does not inline that function. For more information, see "Interaction Between Different Inlining Controls" on page 34-11.

An Example Inlining Strategy

This is an example inlining strategy that balances the speed and readability of the generated code. You instruct the code generator to perform these actions simultaneously:

- Preserve the modularity in the code that you write for better readability, even if that reduces the speed of the generated code. For this behavior, set `InlineBetweenUserFunctions` to `'Readability'`.
- Generate highly optimized code for MathWorks functions, even if that results in less readable code because you are less likely to inspect this part of your code base. For this behavior, set `InlineBetweenMathWorksFunctions` to `'Speed'`.

- In the generated code, separate functions that you write and MathWorks functions so that the generated code does not look very different from your MATLAB code. For this behavior, set `InlineBetweenUserAndMathWorksFunctions` to `'Readability'`.

Interaction Between Different Inlining Controls

- The `coder.inline('always')` or `coder.inline('never')` directive placed inside the body of a MATLAB function overrides the effect of the global inlining controls, including the `codegen` options and the code configuration settings. See `coder.inline`.

  Certain MathWorks functions include a call to the `coder.inline` directive that affects how those functions interact with the global inlining settings. For example, if the body of a MathWorks function contains the `coder.inline('never')` directive and you set `InlineBetweenMathWorksFunctions` to `'Always'`, the code generator gives preference to the `coder.inline` directive and does not inline that function.
- The `-O disable:inline` and `-O enable:inline` options of the `codegen` command override the individual values of the three code configuration parameters `InlineBetweenUserFunctions`, `InlineBetweenMathWorksFunctions`, and `InlineBetweenUserAndMathWorksFunctions`.

Example: Control Inlining at the Boundary Between Your Functions and MathWorks® Functions

This example shows how to control inlining behavior at all call sites where a function that you wrote calls a MathWorks function, or a MathWorks function calls a function that you wrote.

Define A Function That Calls MathWorks Functions

Define a MATLAB function `useBessely` that accepts a double array `x` as input, processes the input array by using the `bessely` function, and returns an array that has the same type and size as `x`.

```
type useBessely.m
```

```
function out = useBessely(x)
out = x + bessely(3,x);
end
```

Generate Code With Default Inlining Settings

Generate a static C++ library for the `useBessely` function. Specify the input to be a `1`-by-`100` `double` type. Use the default values for the inlining settings. These default values optimize the speed of the generated code. Use the `-c` flag that instructs the code generator to produce source code only and not build the source code.

```
codegen -c -lang:c++ -config:lib useBessely -args {zeros(1,100)} -report
```

```
Code generation successful: To view the report, open('codegen\lib\useBessely\html\report.mldatx')
```

Open the code generation report and inspect the generated code. Observe that no separate C++ function has been generated for the MathWorks function `bessely`. The code generator has inlined the code for the `bessely` function into the C++ `useBessely` function that is contained in the file `useBessely.cpp`.

Generate Code With Modified Inlining Settings

Define a code configuration object `cfg` for generating a static C++ library. Set the property `InlineBetweenUserAndMathWorksFunctions` to `'Never'`. This setting instructs the code generator to separate the function that you wrote and the MathWorks functions in the generated code. As a result, the generated C++ code is less efficient but more readable than the inlined code.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.InlineBetweenUserAndMathWorksFunctions = 'Never';
```

Generate code by using `cfg` as the code configuration object. Specify the input to be a `1`-by-`100` `double` type. Use the `-c` flag that instructs the code generator to produce source code only and not build the source code.

```
codegen -c -config cfg useBessely -args {zeros(1,100)} -report
```

```
Code generation successful: To view the report, open('codegen\lib\useBessely\html\report.mldatx')
```

Open the code generation report and inspect the generated code. The C++ function `useBessely` now calls another C++ function `coder::bessely` that contains the code generated for the MathWorks function `bessely`. As a reault, the generated C++ `useBessely` function looks similar to the MATLAB `useBessely` function that you wrote.

```
type codegen/lib/useBessely/useBessely.cpp
```

```
//
// File: useBessely.cpp
//
// MATLAB Coder version            : 5.2
// C/C++ source code generated on  : 23-Feb-2021 16:48:55
//

// Include Files
#include "useBessely.h"
#include "bessely.h"
#include "rt_nonfinite.h"

// Function Definitions
//
// Arguments    : const double x[100]
//                creal_T out[100]
// Return Type  : void
//
void useBessely(const double x[100], creal_T out[100])
{
  coder::bessely(x, out);
  for (int i{0}; i < 100; i++) {
    out[i].re += x[i];
  }
}

//
// File trailer for useBessely.cpp
//
```

```
// [EOF]
//
```

See Also

codegen | coder.CodeConfig | coder.EmbeddedCodeConfig | coder.MexCodeConfig | coder.inline

More About

- “Optimization Strategies” on page 34-3

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 expression `(1:10).^2` with `coder.const((1:10).^2)`, and then generate code for `AddShift` again 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 i;
  static const signed char iv[10] = { 1, 4, 9, 16, 25, 36,
                                  49, 64, 81, 100 };

  for (i = 0; i < 10; i++) {
    y[i] = (double)iv[i] + Shift;
  }
}
```

See Also

`coder.const`

More About

- "Use coder.const with Extrinsic Function Calls" on page 34-46

Control Stack Space Usage

You can control the maximum stack size used by your compiler or hardware. A stack is a block of memory that stores local variables for program execution. Stack memory is allocated during code generation. Stack allocation is typically more efficient for memory usage than static allocation.

The value of the configuration setting `StackUsageMax` is measured in bytes. Based on information from the target hardware settings and the possible execution paths in the code, the software estimates the stack variables that a certain value of `StackUsageMax` can accommodate. This estimate does not account for stack size changes introduced by the C compiler. Variables that do not fit in stack memory are spilled off the stack. The variables that are spilled off the stack are stored in static memory or a spill structure if you are trying to generate reentrant code.

- You can increase `StackUsageMax` to raise the number of variables allocated to stack memory. If your target hardware has sufficient stack space, this reduces the amount of variables that are spilled off the stack.
- You can decrease `StackUsageMax` to reduce the number of variables allocated to stack memory. If your target hardware lacks sufficient stack space, this increases the number of variables that are spilled off of the stack.

Variables in recursive functions that couldn't fit on the stack are not stored in a static memory, or in a spill structure if you generate reentrant code. Variables in recursive functions are not spilled off the stack, even if they exceed the stack usage size.

Similarly, code generation does not account for the stack usage of custom code in calls to `coder.ceval`.

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.

Control Stack Space Usage by 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`, `MEX`, `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'`, `mex`,`'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;
   ```

Generated Code Without Spilled Variables

The various outcomes depend on the amount of stack space.

On generating code for the following MATLAB code with ample stack space, the generated code is:

```
function y = fooNorm(x)
b = cast(x,'uint32');
y = sum(b);
end
```

The input to the function `fooNorm(x)` is a `100`-by-`100` matrix of ones.

```
void fooNorm(const double x[10000], double y[100])
{
  double d;
  unsigned int b[10000];
  ...
}
static void main_fooNorm(void)
{
  double dv[10000];
  double y[100];
  argInit_100x100_real_T(dv);
  fooNorm(dv, y);
}
```

This code snippet highlights the entry-point function `fooNorm`. The function `main_fooNorm` declares the variable `dv[10000]` and `y[100]`on the stack, which is the input to the function `fooNorm`.

Generated Code That Has Spilled Variables

When you generate code for the same MATLAB code with insufficient stack space, the code is:

```
void fooNorm(const double x[10000], double y[100])
{
  static unsigned int b[10000];
  double d;
  ...
}
static void main_fooNorm(void)
{
  static double dv[10000];
  static double y[100];
  argInit_100x100_real_T(dv);
  fooNorm(dv, y);
}
```

The variables `b[10000]`, `dv[10000]`, and `y[100]` are declared as static variables because they do not fit on the stack.

Generated Reentrant Code That Has Spilled Variables

When you generate reentrant code for the same MATLAB code with insufficient stack space, the generated code is:

```
void fooNorm(fooNormStackData *SD, const double x[10000], double y[100])
{
```

```
  double d;
  ...
}
static void main_fooNorm(void)
{
  static double dv[10000];
  static double y[100];
  argInit_100x100_real_T(dv);
  fooNorm(&fooNormStackDataGlobal, dv, y);
}
```

The input to `fooNorm` is a structure `fooNormStackData`. On generating reentrant code, when variables spill off the stack, a spill structure is generated that holds the variables that do not fit on the stack.

The structure `fooNormStackData` is defined as:

```
typedef struct {
  struct {
    unsigned int b[10000];
  } f0;
} fooNormStackData;
```

See Also

More About

- “Stack Allocation and Performance” on page 34-18
- “Generate Reentrant C Code from MATLAB Code” on page 35-2
- “Code Generation for Recursive Functions” on page 20-14

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 34-15.

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.

See Also

“Control Stack Space Usage” on page 34-15 | “Generate Reentrant C Code from MATLAB Code” on page 35-2

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 34-20.

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 34-21.
2. Depending on your requirements, do one of the following:

   - “Disable Dynamic Memory Allocation During Code Generation” on page 34-25.
   - “Set Dynamic Memory Allocation Threshold” on page 34-26

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.

See Also

More About

- “Dynamic Memory Allocation and Performance” on page 34-19

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:

  Example 34.1. 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:

Example 34.2. 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.

See Also

Related Examples

- “Minimize Dynamic Memory Allocation” on page 34-20
- “Disable Dynamic Memory Allocation During Code Generation” on page 34-25
- “Set Dynamic Memory Allocation Threshold” on page 34-26

More About

- “Dynamic Memory Allocation and Performance” on page 34-19

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.

See Also

Related Examples

- “Minimize Dynamic Memory Allocation” on page 34-20
- “Provide Maximum Size for Variable-Size Arrays” on page 34-21
- “Set Dynamic Memory Allocation Threshold” on page 34-26

More About

- “Dynamic Memory Allocation and Performance” on page 34-19

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.

   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.

See Also

Related Examples

- “Minimize Dynamic Memory Allocation” on page 34-20
- “Provide Maximum Size for Variable-Size Arrays” on page 34-21
- “Disable Dynamic Memory Allocation During Code Generation” on page 34-25

More About

- “Dynamic Memory Allocation and Performance” on page 34-19

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.

See Also

Related Examples

- "Prevent Code Generation for Unused Execution Paths" on page 34-29

Prevent Code Generation for Unused Execution Paths

In this section...

"Prevent Code Generation When Local Variable Controls Flow" on page 34-29
"Prevent Code Generation When Input Variable Controls Flow" on page 34-29

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 34-29.
- 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 34-29.

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.

See Also

More About

- “Excluding Unused Paths from Generated Code” on page 34-28

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.

See Also

More About

- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 32-14
- “Classification of Variables in parfor-Loops” on page 32-20
- “Reduction Assignments in parfor-Loops” on page 32-19

Minimize Redundant Operations in Loops

This example shows how to minimize redundant operations in loops. When a loop operation does not depend on the loop index, performing it inside a loop is redundant. This redundancy often goes unnoticed when you are performing multiple operations in a single MATLAB statement inside a loop. For example, in the following code, the inverse of the matrix B is being calculated 100 times inside the loop although it does not depend on the loop index:

```
for i=1:100
     C=C + inv(B)*A^i*B;
  end
```

Performing such redundant loop operations can lead to unnecessary processing. To avoid unnecessary processing, move operations outside loops as long as they do not depend on the loop index.

1. Define a function, `SeriesFunc(A,B,n)`, that calculates the sum of `n` terms in the following power series expansion:

   $$C = 1 + B^{-1}AB + B^{-1}A^2B + \ldots$$

   ```
     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.

Force Loop Unrolling by Using coder.unroll

The code generator uses heuristics to determine when to unroll a `for`-loop. To force loop unrolling, use `coder.unroll`. This affects only the `for` loop that is immediately after `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
```

Set Loop Unrolling Threshold for All for-Loops in the MATLAB Code

If a `for`-loop is not preceded by `coder.unroll`, the code generator uses a loop unrolling threshold to determine whether to automatically unroll the loop. If the number of loop iterations is less than the threshold, the code generator unrolls the loop. If the number of iterations is greater than or equal to the threshold, the code generator produces a `for`-loop. The default value of the threshold is `5`. By modifying this threshold, you can fine-tune loop unrolling. To modify the threshold:

- In a configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set the `LoopUnrollThreshold` property.
- In the MATLAB Coder app, on the **Speed** tab, set **Loop unrolling threshold**.

Unlike the `coder.unroll` directive, the threshold applies to all `for`-loops in your MATLAB code. The threshold can also apply to some `for`-loops produced during code generation.

For an individual loop, a `coder.unroll` directive takes precedence over the loop unrolling optimization.

Unroll Simple for-Loops

Consider this function:

```
function [x,y] = call_myloops()
%#codegen
x = myloop1(5);
y = myloop2(5);
end

function b = myloop1(n)
b = zeros(1,n);
for i = 1:n
    b(i)=i+n;
end
end

function b = myloop2(n)
b = zeros(1,n);
for i = 1:n
    b(i)=i*n;
end
end
```

To set the value of the loop unrolling threshold to `6`, and then generate a static library, run:

```
cfg = coder.CodeConfig;
cfg.LoopUnrollThreshold = 6;
codegen call_myloops -config cfg
```

This is the generated code for the `for`-loops. The code generator unrolled both `for`-loops.

```
  x[0] = 6.0;
  y[0] = 5.0;
  x[1] = 7.0;
  y[1] = 10.0;
  x[2] = 8.0;
  y[2] = 15.0;
  x[3] = 9.0;
  y[3] = 20.0;
  x[4] = 10.0;
  y[4] = 25.0;
```

Unroll Nested for-Loops

Suppose that your MATLAB code has two nested `for`-loops.

- If the number of iterations of the inner loop is less than the threshold, the code generator first unrolls the inner loop. Subsequently, if the product of the number of iterations of the two loops is also less than the threshold, the code generator unrolls the outer loop. Otherwise the code generator produces the outer `for`-loop.
- If the number of iterations of the inner loop is equal to or greater than the threshold, the code generator produces both `for`-loops.

This behavior is generalized to multiple nested `for`-loops.

Consider the function `nestedloops_1` with two nested `for`-loops:

```
function y = nestedloops_1
%#codegen
y = zeros(2,2);
for i = 1:2
    for j = 1:2
        y(i,j) = i+j;
    end
end
end
```

Generate code for `nestedloops_1` with the loop unrolling threshold set to the default value of `5`. Here is the generated code for the `for`-loops. The code generator unrolled both `for`-loops because the product of the number of iterations of the two loops is `4`, which is less than the threshold.

```
  y[0] = 2.0;
  y[2] = 3.0;
  y[1] = 3.0;
  y[3] = 4.0;
```

Now, generate code for the function `nestedloops_2` with the loop unrolling threshold set to the default value of `5`.

```
function y = nestedloops_2
%#codegen
y = zeros(3,2);
for i = 1:3
    for j = 1:2
        y(i,j) = i+j;
    end
end
end
```

The number of iterations of the inner loop is less than the threshold. The code generator unrolls the inner loop. But the product of the number of iterations of the two loops is `6`, which is greater than the threshold. Therefore, the code generator produces code for the outer `for`-loop. Here is the generated code for the `for`-loops.

```
for (i = 0; i < 3; i++) {
  y[i] = (double)i + 2.0;
  y[i + 3] = ((double)i + 1.0) + 2.0;
}
```

See Also

`coder.unroll`

More About

- "Nonconstant Index into varargin or varargout in a for-Loop" on page 36-14

Disable Support for Integer Overflow or Nonfinites

The code generator produces supporting code for these situations:

- The result of an integer operation falls outside the range that a data type can represent, known as integer overflow.
- An operation generates nonfinite values (`inf` and `NaN`).

If you know that these situations do not occur, to reduce the size of the generated code and increase its speed, you can suppress generation of the supporting code. However, if you suppress generation of the supporting code and one of these situations occurs, the behavior of the generated code might not match the behavior of the original MATLAB code.

Disable Support for Integer Overflow

By default, the code generator produces code to handle integer overflow. Overflows saturate to either the minimum or maximum value that the data type can represent. If you know that your code does not depend on integer overflow support, to improve performance, you can disable the generation of the code that handles integer overflow. To disable integer overflow support:

- In a code generation configuration object for MEX or standalone code (static library, dynamically linked library, or executable program), set the `SaturateOnIntegerOverflow` property to `false`.
- In the MATLAB Coder app, set **Saturate on integer overflow** to `No`.

Note Do not disable support for integer overflow unless you know that your code does not depend on it. If you disable the support and run-time checking is enabled, the generated code produces an error for integer overflow. If you disable integer overflow support and also disable run-time checking, the behavior for integer overflow is undefined. Most C compilers wrap on overflow.

To check whether your code depends on integer overflow support:

1. Disable integer overflow support.
2. Enable checks to detect integer overflow at run time.
   - For MEX, enable integrity checking. See “Control Run-Time Checks” on page 32-12.
   - For standalone code (static library, dynamically linked library, or executable program), enable run-time checks. See “Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors” on page 28-20.
3. Run the generated code over the full range of input values. If the generated code detects integer overflow, it produces an error.

Disable Support for Nonfinite Numbers

By default, for standalone code (static library, dynamically linked library, or executable program), the code generator produces code to handle nonfinite numbers (`inf` and `NaN`). If you know that your code does not depend on nonfinite number support, to improve the performance of the generated code, you can disable the support. To disable nonfinite support:

- In a code generation configuration object, set the `SupportNonFinite` property to `false`.
- In the MATLAB Coder app, set **Support nonfinite numbers** to `No`.

If you disable nonfinite support, test that your generated code behaves as expected.

See Also

More About

- "Configure Build Settings" on page 27-13
- "Control Run-Time Checks" on page 32-12
- "Generate Standalone C/C++ Code that Detects and Reports Run-Time Errors" on page 28-20

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

`assert` | `coder.BuildConfig` | `coder.ExternalDependency` | `coder.ceval` | `coder.opaque` | `coder.rref` | `coder.wref`

More About

- “Develop Interface for External C/C++ Code” on page 33-12

MATLAB Coder Optimizations in Generated Code

In this section...

"Constant Folding" on page 34-43
"Loop Fusion" on page 34-44
"Successive Matrix Operations Combined" on page 34-44
"Unreachable Code Elimination" on page 34-44
"memcpy Calls" on page 34-45
"memset Calls" on page 34-45

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`.

```
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 34-14.

Control Constant Folding

You can control the maximum number of instructions that can be constant-folded from the command line or the project settings dialog box.

- At the command line, create a configuration object for code generation. Set the property `ConstantFoldingTimeout` to the value that you want.

  ```
  cfg=coder.config('lib');
  cfg.ConstantFoldingTimeout = 200;
  ```

- Using the app, in the project settings dialog box, on the **All Settings** tab, set the field **Constant folding timeout** to the value that you want.

Loop Fusion

When possible, the code 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 34-48.

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 34-49.

Use coder.const with Extrinsic Function Calls

You can use `coder.const` to fold a function call into a constant in the generated code. The code generator evaluates the function call and replaces it with the result of the evaluation. If you make the function call extrinsic, the function call is evaluated by MATLAB instead of by the code generator. Use `coder.const` with an extrinsic function call to:

- Reduce code generation time, especially for constant-folding of computationally intensive expressions.
- Force constant-folding when `coder.const` is unable to constant-fold.

To make an individual function call extrinsic, use `feval`. To make all calls to a particular function extrinsic, use `coder.extrinsic`.

Reduce Code Generation Time by Using coder.const with feval

Consider this function that folds a computationally intensive expression `besselj(3, zTable)` into a constant:

```
function j = fcn(z)
zTable = coder.const(0:0.01:100);
jTable = coder.const(besselj(3,zTable));
j = interp1(zTable,jTable,z);
end
```

To make code generation of `fcn` faster, evaluate `besselj(3, zTable)` in MATLAB by using `feval`.

```
function j = fcn(z)
zTable = coder.const(0:0.01:100);
jTable = coder.const(feval('besselj',3,zTable));
j = interp1(zTable,jTable,z);
end
```

Force Constant-Folding by Using coder.const with feval

Consider this function that folds the function call `rand(1,100)` into a constant.

```
function yi = fcn(xi)
y = coder.const(rand(1,100));
yi = interp1(y,xi);
end
```

Code generation ends with an error.

```
codegen fcn -args {0} -config:lib -report
```

```
??? The input to coder.const cannot be reduced to a constant.
```

To successfully constant-fold `rand(1,100)`, evaluate it in MATLAB by using `feval`.

```
function yi = fcn(xi)
y = coder.const(feval('rand',1,100));
yi = interp1(y,xi);
end
```

See Also

`coder.const` | `coder.extrinsic`

More About

- “Fold Function Calls into Constants” on page 34-14
- “Use MATLAB Engine to Execute a Function Call in Generated Code” on page 20-8

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 a `for`-loop or multiple, consecutive element assignments. This table shows examples of generated C code with and without the `memcpy` optimization.

Code Generated with memcpy Optimization	Code Generated Without 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];`

To enable or disable the `memcpy` optimization:

- At the command line, set the code configuration object property `EnableMemcpy` to `true` or `false`. The default value is `true`.
- In the MATLAB Coder app, set **Use memcpy for vector assignment** to `Yes` or `No`. The default value is `Yes`.

When the `memcpy` optimization is enabled, the use of `memcpy` depends on the number of bytes to copy. 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.

- If the number of elements to copy is known at compile time, then the code generator produces a `memcpy` call only when the number of bytes is greater than or equal to the `memcpy` threshold.
- If the number of elements is not known at compile time, then the code generator produces a `memcpy` call without regard to the threshold.

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.

In certain cases, the code generator can produce a `memcpy` call without regard to the `EnableMemcpy` or `MemcpyThreshold` parameters, or their equivalent settings in the app.

See Also

More About

- “memset Optimization” on page 34-49
- “MATLAB Coder Optimizations in Generated Code” on page 34-43
- “Optimization Strategies” on page 34-3

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 a `for`-loop or multiple, consecutive element assignments. This table shows examples of generated C code with and without `memset`.

Code Generated with memset Optimization	Code Generated Without memset Optimization
`memset(&Y[0], 125, 100U * sizeof(signed char));`	`for (i = 0; i < 100; i++) {` `    Y[i] = 125;`

`memset(&Z[0], 0, 1000U * sizeof(double));`	`Z[0] = 0.0;`
	`Z[1] = 0.0;`
	`Z[2] = 0.0;`
	`...`
	`Z[999] = 0.0;`

The code generator can use the `memset` optimization for assignment of an integer constant or a floating-point zero. The use of `memset` depends on:

- The size of the value to assign. The size must meet the requirements for a C/C++ `memset` call.
- The number of bytes to assign. 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.
  - If the number of elements to assign is known at compile time, then the code generator produces a `memset` call only when the number of bytes is greater than or equal to the threshold.
  - If the number of elements is not known at compile time, then the code generator produces a `memset` call without regard to the threshold.

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)**.

For assignment of floating-point zero, to enable or disable the `memset` optimization:

- At the command line, set the code configuration object property `InitFltsAndDblsToZero` to `true` or `false`. 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`. The default value is `Yes`.

See Also

More About

- "memcpy Optimization" on page 34-48
- "MATLAB Coder Optimizations in Generated Code" on page 34-43
- "Optimization Strategies" on page 34-3

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`.

See Also

More About

- “Preserve Variable Names in Generated Code” on page 27-38
- “Optimization Strategies” on page 34-3
- “Configure Build Settings” on page 27-13

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 34-52.

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 34-52.

See Also

More About

- "Optimization Strategies" on page 34-3

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');
            buildInfo.addDefines('LAPACK_ILP64');
        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');
```

You must specify the integer type that your LAPACK library uses. Not specifying this integer type can result in incorrect behaviors or crashes. Do one of the following:

- Include these lines in the updateBuildInfo method.

  ```
  buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
  buildInfo.addDefines('LAPACK_ILP64');
  ```
- Alternatively, you can directly specify the integer type that your LAPACK library uses. For example, if the integer type is long long, include this line in the updateBuildInfo method.

  ```
  buildInfo.addDefines('lapack_int=long long');
  ```

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 mysvd -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 macOS platform, 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 34-51
- "Optimization Strategies" on page 34-3

External Websites

- www.netlib.org/lapack
- www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations

BLAS Calls in Generated Code

To improve the execution speed of code generated for certain low-level vector and matrix operations (such as matrix multiplication), MATLAB Coder can generate calls to BLAS functions instead of generating code for these operations. BLAS is a software library for low-level vector and matrix computations that has several highly optimized machine-specific implementations. MATLAB Coder uses the CBLAS C interface to BLAS.

For MEX generation, if the input arrays for the matrix functions meet certain criteria, the code generator produces BLAS calls. For standalone code (library or executable program), by default, the code generator does not produce BLAS calls. If you specify that you want to generate BLAS calls, and the input arrays for the matrix functions meet the criteria, the code generator produces BLAS calls.

For MEX functions, the code generator uses the BLAS library that is included with MATLAB. For standalone code, the code generator uses the BLAS library that you specify. See “Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls” on page 34-56.

See Also

More About

- “Optimization Strategies” on page 34-3

External Websites

- https://www.netlib.org/blas/

Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls

To improve the execution speed of code generated for certain low-level vector and matrix operations (such as matrix multiplication) in standalone code, specify that you want MATLAB Coder to generate BLAS calls. BLAS is a software library for low-level vector and matrix computations that has several highly optimized machine-specific implementations. The code generator uses the CBLAS C interface to BLAS. If you specify that you want to generate BLAS calls, and the input arrays for the matrix functions meet certain criteria, the code generator produces the BLAS calls. Otherwise, the code generator produces code for the matrix functions.

For BLAS calls in standalone code, MATLAB Coder uses the BLAS library that you specify. Specify a BLAS library that is optimized for your execution environment.

Specify BLAS Library

To generate BLAS calls in standalone code, you must have access to a BLAS callback class. A BLAS callback class specifies the BLAS library, the CBLAS header file, certain C data types that the particular CBLAS interface uses, and the compiler and linker options for the build process. Do one of the following:

- At the command line, set the code configuration object property `CustomBLASCallback` to the name of the callback class.
- In the MATLAB Coder app, set **Custom BLAS library callback** to the name of the callback class.

Write BLAS Callback Class

To generate calls to a specific BLAS library in the generated code, write a BLAS callback class. Share the callback class with others who want to use this BLAS library for BLAS calls in standalone code.

The callback class must derive from the abstract class `coder.BLASCallback`. This example is an implementation of the callback class `mklcallback` for integration with the Intel MKL BLAS library on a Windows platform.

```
classdef mklcallback < coder.BLASCallback
    methods (Static)
        function updateBuildInfo(buildInfo, ~)
            libPath = fullfile(pwd,'mkl','WIN','lib','intel64');
            libPriority = '';
            libPreCompiled = true;
            libLinkOnly = true;
            libs = {'mkl_intel_ilp64.lib' 'mkl_intel_thread.lib' 'mkl_core.lib'};
            buildInfo.addLinkObjects(libs, libPath, libPriority, libPreCompiled, ...
                                      libLinkOnly);
            buildInfo.addLinkObjects('libiomp5md.lib',fullfile(matlabroot,'bin', ...
                                'win64'), libPriority, libPreCompiled, libLinkOnly);
            buildInfo.addIncludePaths(fullfile(pwd,'mkl','WIN','include'));
            buildInfo.addDefines('-DMKL_ILP64');
        end
        function headerName = getHeaderFilename()
            headerName = 'mkl_cblas.h';
        end
        function intTypeName = getBLASIntTypeName()
```

```
            intTypeName = 'MKL_INT';
        end
    end
end
```

You must provide the getHeaderFilename, getBLASIntTypeName, and updateBuildInfo methods. The getHeaderFilename method returns the CBLAS header file name. If you are using a different BLAS library, replace mkl_cblas.h with the name of your CBLAS header file. The getBLASIntTypeName method returns the name of the integer data type that your CBLAS interface uses. If you are using a different BLAS library, replace MKL_INT with the name of the integer data type specific to your CBLAS interface. The updateBuildInfo method provides the information required for the build process to link to the BLAS library. Use code that is like the code in the example callback class to specify the location of header file, the full path name of the BLAS library, and the compiler and linker options. If you use the Intel MKL BLAS library, use the link line advisor to see which libraries and compiler options are recommended for your use case.

There are three other methods that are already implemented in coder.BLASCallback. These methods are getBLASDoubleComplexTypeName, getBLASSingleComplexTypeName, and useEnumNameRatherThanTypedef. By default, your callback class inherits these implementations from coder.BLASCallback. In certain situations, you must override these methods with your own definitions when you define your callback class.

The getBLASDoubleComplexTypeName method returns the type used for double-precision complex variables in the generated code. If your BLAS library takes a type other than double* and void* for double-precision complex array arguments, include this method in your callback class definition.

```
function doubleComplexTypeName = getBLASDoubleComplexTypeName()
doubleComplexTypeName = 'my_double_complex_type';
end
```

Replace my_double_complex_type with the type that your BLAS library takes for double-precision complex array arguments.

The getBLASSingleComplexTypeName method returns the type used for single-precision complex variables in the generated code. If your BLAS library takes a type other than float* and void* for single-precision complex array arguments, include this method in your callback class definition.

```
function singleComplexTypeName = getBLASSingleComplexTypeName()
doubleComplexTypeName = 'my_single_complex_type';
end
```

Replace my_single_complex_type with the type that your BLAS library takes for single-precision complex array arguments.

The useEnumNameRatherThanTypedef method returns false by default. If types for enumerations in your BLAS library include the enum keyword, redefine this method to return true in your callback class definition.

```
function p = useEnumNameRatherThanTypedef()
p = true;
end
```

An excerpt from generated C source code that includes the enum keyword is:

```
enum CBLAS_SIDE t;
enum CBLAS_UPLO b_t;
```

```
double temp;
enum CBLAS_TRANSPOSE c_t;
enum CBLAS_DIAG d_t;
```

Generate BLAS Calls by Specifying a BLAS Callback Class

This example shows how to generate code that calls BLAS functions in a specific BLAS library. The BLAS callback class `useMyBLAS` specifies the BLAS library that you want to use in this example.

1. Write a MATLAB function that calls a function for a basic matrix operation. For example, write a function `myMultiply` that multiplies two matrices `A` and `B`.

   ```
   function C = myMultiply(A,B) %#codegen
   C = A*B;
   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 BLAS callback class `useMyBLAS`.

   ```
   cfg.CustomBLASCallback = 'useMyBLAS';
   ```

   The callback class must be on the MATLAB path.
4. Generate code. Specify that the inputs `A` and `B` are 1000-by-1000 arrays of doubles.

   ```
   codegen myMultiply -args {zeros(1000),zeros(1000)} -config cfg -report
   ```

   If `A` and `B` are large enough, the code generator produces a BLAS call for the matrix multiplication function.

Locate BLAS Library in Execution Environment

The BLAS library must be available in your execution environment. If your BLAS library is shared, use environment variables or linker options to specify the location of the BLAS 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 macOS platform, modify the DYLD_LIBRARY_PATH environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, use the build information `addLinkFlags` method in the `updateBuildInfo` method of your BLAS callback class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

Usage Notes and Limitations for OpenBLAS Library

If you generate code that includes calls to the OpenBLAS library functions, follow these guidelines and restrictions:

- If you generate C++ code that includes calls to OpenBLAS library functions, compiling it with the `-pedantic` option produces warnings. To disable the `-pedantic` compiler option, include these lines in the `updateBuildInfo` method:

```
if ctx.getTargetLang() == 'C++'
    buildInfo.addCompileFlags('-Wno-pedantic');
end
```

- OpenBLAS does not support the C89/C90 standard.

See Also

`coder.BLASCallback`

More About

- “BLAS Calls in Generated Code” on page 34-55
- “Optimization Strategies” on page 34-3

External Websites

- https://www.netlib.org/blas/
- https://www.netlib.org/blas/faq.html#_5_a_id_are_optimized_blas_libraries_available_where_can_i_find_vendor_supplied_blas_a_are_optimized_blas_libraries_available_where_can_i_find_optimized_blas_libraries
- https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top/blas-and-sparse-blas-routines.html
- https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-link-line-advisor.html

Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls

This example shows how to produce calls to a specific installed FFTW library when you generate standalone code (static library, dynamically linked library, or executable program). For more information about FFTW, see www.fftw.org.

When you generate a MEX function from MATLAB code that includes fast Fourier transform (FFT) functions, the code generator uses the library that MATLAB uses for FFT algorithms. If you generate standalone C/C++ code, by default, the code generator produces code for the FFT algorithms instead of producing FFT library calls. To increase the speed of fast Fourier transforms in generated standalone code, specify that the code generator produce calls to a specific installed FFTW library.

The code generator produces FFTW library calls when all of these conditions are true:

- Your MATLAB code calls one of these functions:`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, or `ifftn`.
- You generate standalone C/C++ code.
- You have access to an FFTW library installation, version 3.2 or later.
- You specify the FFTW library installation in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.
- You set the `CustomFFTCallback` configuration parameter to the name of the callback class. In the MATLAB Coder app, use the **Custom FFT library callback** setting.

Install FFTW Library

If you do not have access to an installed FFTW library, version 3.2 or later, then you must install one. For a Linux platform or a Mac platform, consider using a package manager to install the FFTW library. For a Windows platform, in addition to `.dll` files, you must have `.lib` import libraries, as described in the Windows installation notes on the FFTW website.

See the installation instructions for your platform on the FFTW website.

Write an FFT Callback Class

To specify your installation of the FFTW library, write an FFT callback class. Share the callback class with others who want to use this FFTW library for FFTW calls in standalone code.

The callback class must derive from the abstract class `coder.fftw.StandaloneFFTW3Interface`. Use this example callback class as a template.

```
% copyright 2017 The MathWorks, Inc.

classdef useMyFFTW < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '/usr/lib/fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
```

```
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
             libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);
        end
    end
end
```

Implement the `updateBuildInfo` and `getNumThreads` methods. In the `updateBuildInfo` method, set `fftwLocation` to the full path for your installation of the library. Set `includePath` to the full path of the folder that contains `fftw3.h`. Set `libPath` to the full path of the folder that contains the library files. If your FFTW installation uses multiple threads, modify the `getNumThreads` method to return the number of threads that you want to use.

Optionally, you can implement these methods:

- `getPlanMethod` to specify the FFTW planning method. See `coder.fftw.StandaloneFFTW3Interface`.
- `lock` and `unlock` to synchronize multithreaded access to the FFTW planning process. See "Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code" on page 34-63.

Generate FFTW Library Calls by Specifying an FFT Library Callback Class

To generate FFTW library calls in standalone C code:

1. Write a MATLAB function that calls a MATLAB fast Fourier transform function. For example, write a function `myfft` that calls the MATLAB function `fft`.

   ```
   function y = myfft()
   %#codegen
   t = 0:1/50:10-1/50;
   x = sin(2*pi*15*t) + sin(2*pi*20*t);
   y = fft(x);
   end
   ```
2. Define a code generation configuration object for a static library, dynamically linked library, or executable program. For example, define a configuration object for a dynamically linked library.

   ```
   cfg = coder.config('dll');
   ```
3. Specify the FFTW callback class `useMyFFTW`.

   ```
   cfg.CustomFFTCallback = 'useMyFFTW';
   ```

   The callback class must be on the MATLAB path.
4. Generate code.

   ```
   codegen myfft -config cfg -report
   ```

Locate FFTW Library in Execution Environment

The FFTW library must be available in your execution environment. If the FFTW library is shared, use environment variables or linker options to specify the location of the 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 macOS platform, 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.fftw.StandaloneFFTW3Interface` class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

See Also

`coder.fftw.StandaloneFFTW3Interface`

More About

- “Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code” on page 34-63

External Websites

- www.fftw.org

Synchronize Multithreaded Access to FFTW Planning in Generated Standalone Code

This example shows how to generate standalone code (static library, dynamically linked library, or executable program) that synchronizes multithreaded access to the FFTW planning process.

The code generator produces FFTW library calls when all of these conditions are true:

- Your MATLAB code calls one of these functions:`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, or `ifftn`.
- You generate standalone C/C++ code.
- You have access to an FFTW library installation, version 3.2 or later.
- You specify the FFTW library installation in an FFT library callback class that derives from `coder.fftw.StandaloneFFTW3Interface`.
- You set the `CustomFFTCallback` configuration parameter to the name of the callback class. In the MATLAB Coder app, use the **Custom FFT library callback** setting.

If multiple threads call the FFTW library, then the generated code must prevent concurrent access to the FFTW planning process. To synchronize access to FFTW planning, in your FFT library callback class, implement the `lock` and `unlock` methods. You must also provide C code that manages a lock or mutex. Many libraries, such as OpenMP, pthreads, and the C++ standard library (C++ 11 and later) provide locks. This example shows how to implement the `lock` and `unlock` methods and provide supporting C code. To manage a lock, this example uses the OpenMP library.

Prerequisites

Before you start, for the basic workflow for generating FFTW library calls in standalone code, see “Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls” on page 34-60.

You must have:

- Access to an installed FFTW library.
- A compiler that supports the OpenMP library. To use a different library, such as pthreads, modify the supporting C code accordingly.

Create a MATLAB Function

Write a MATLAB function `mycustomfft` that calls a fast Fourier transform function inside a `parfor` loop:

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

t = 0:1/50:10-1/50;
x = sin(2*pi*15*t) + sin(2*pi*20*t);
y = fft(x);
parfor k = 1:100
    y = y + ifft(x+k);
end
```

Write Supporting C Code

Write C functions that initialize, set, and unset a lock. This example uses the OpenMP library to manage the lock. For a different library, modify the code accordingly.

- Create a file `mylock.c` that contains this C code:

```
#include "mylock.h"
#include "omp.h"

static omp_nest_lock_t lockVar;

void mylock_initialize(void)
{
    omp_init_nest_lock(&lockVar);
}

void mylock(void)
{
    omp_set_nest_lock(&lockVar);
}

void myunlock(void)
{
    omp_unset_nest_lock(&lockVar);
}
```

- Create a header file `mylock.h` that contains:

```
#ifndef MYLOCK_H
#define MYLOCK_H

 void mylock_initialize(void);
 void mylock(void);
 void myunlock(void);

#endif
```

Write an FFT Library Callback Class

Write an FFT callback class `myfftcb` that:

- Specifies the FFTW library.
- Implements `lock` and `unlock` methods that call the supporting C code to control access to the FFTW planning.

Use this class as a template. Replace `fftwLocation` with the location of your FFTW library installation.

```
classdef myfftcb < coder.fftw.StandaloneFFTW3Interface

    methods (Static)
        function th = getNumThreads
            coder.inline('always');
            th = int32(coder.const(1));
        end
```

```
        function lock()
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('mylock');
        end

        function unlock()
            coder.cinclude('mylock.h', 'InAllSourceFiles', true);
            coder.inline('always');
            coder.ceval('myunlock');
        end

        function updateBuildInfo(buildInfo, ctx)
            fftwLocation = '\usr\lib\fftw';
            includePath = fullfile(fftwLocation, 'include');
            buildInfo.addIncludePaths(includePath);
            libPath = fullfile(fftwLocation, 'lib');

            %Double
            libName1 = 'libfftw3-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName1 = [libName1 libExt];
            addLinkObjects(buildInfo, libName1, libPath, 1000, true, true);

            %Single
            libName2 = 'libfftw3f-3';
            [~, libExt] = ctx.getStdLibInfo();
            libName2 = [libName2 libExt];
            addLinkObjects(buildInfo, libName2, libPath, 1000, true, true);

        end
    end
end
```

Generate a Dynamically Linked Library

1 Create a code generation configuration object for generation of a dynamically linked library.

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

2 Configure code generation to use the FFT callback class `myfftcb`.

```
cfg.CustomFFTCallback = 'myfftcb';
```

3 Include the supporting C code in the build.

```
cfg.CustomSource = 'mylock.c';
```

4 Generate a call to the lock initialization function in the initialization code.

```
cfg.CustomInitializer = 'mylock_initialize();';
```

5 Generate the library.

```
codegen -config cfg mycustomfft -report
```

This example uses the OpenMP library. Therefore, the `EnableOpenMP` configuration parameter must be `true` or you must manually pass the OpenMP flags to your compiler. By default, the `EnableOpenMP` parameter is `true`.

Specify Configuration Parameters in the MATLAB Coder App

For the preceding example in the MATLAB Coder app, use these project settings:

- To specify the FFT library callback class, set **Custom FFT library callback** to `myfftcb`.
- To specify the C code to include, set **Additional source files** to `mylock.c`.
- To specify generation of a call to `mylock_initialize` in the initialization code, set **Initialize function** to `mylock_initialize();`.

See Also

`coder.ceval` | `coder.fftw.StandaloneFFTW3Interface`

More About

- "Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls" on page 34-60

External Websites

- www.fftw.org

Speed Up MEX Generation by Using JIT Compilation

In this section...

"Specify Use of JIT Compilation in the MATLAB Coder App" on page 34-67
"Specify Use of JIT Compilation at the Command Line" on page 34-67
"JIT Compilation Incompatibilities" on page 34-67

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 34-67. If JIT compilation is enabled, the absence of warning or error messages during code generation indicates successful JIT compilation. In a code generation report, the **Summary** tab indicates that the **Build Type** is `JIT MEX Function`.

Note JIT MEX functions are not compatible across different releases of MATLAB Coder software. Run the JIT MEX function by using MATLAB Coder software of the same release that you used to generate the function.

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

Functions
`codegen` | `coder.updateBuildInfo` | `parfor`

Objects
`coder.MexCodeConfig`

More About

- “JIT MEX Incompatibility Warning” on page 36-2
- “JIT Compilation Does Not Support OpenMP” on page 36-3
- “Speed Up Compilation by Generating Only Code” on page 27-74
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” on page 32-14

Automatically Parallelize for Loops in Generated Code

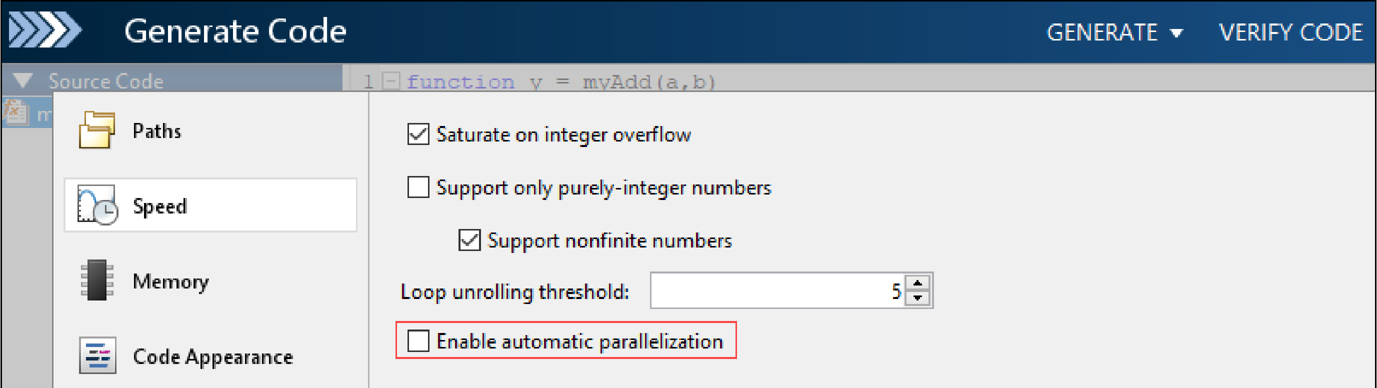
Iterations of parallel `for` loops can run simultaneously on multiple cores on the target hardware. Parallelization of a section of code might significantly improve the execution speed of the generated code. See “How parfor-Loops Improve Execution Speed” on page 32-14.

While generating C/C++ code from your MATLAB code, you can generate parallel `for` loops automatically. Automatic parallelization is a compiler transformation which converts sequential code to multi-threaded code without manual intervention.

Automatic parallelization of `for` loop supports these build types for C/C++ targets: MEX, static library, dynamically linked library, and executable.

Parallelize for Loops by Using MATLAB Coder App

To enable automatic parallelization of `for` loops, in the MATLAB Coder app, in the **Generate Code** step, select **More Settings > Speed > Enable automatic parallelization**.



Parallelize for Loops at Command Line

You can enable parallelization of the `for` loops by using the command line interface. Consider the function `autoparExample`:

```
function x = autoparExample(x)
%#codegen
for i = 10:numel(x)
    x(i) = sqrt(x(i));
end
end
```

To automatically generate parallel `for` loops, run these commands:

```
cfg = coder.config('lib');
cfg.EnableAutoParallelization = 1;
x = rand(1,2000);
codegen -config cfg autoparExample -args {x} -report
```

```
Code generation successful: View report
```

Inspect Generated Code and Code Insights

Open and inspect the code generation report.

Generated Code

Observe the Open Multiprocessing (OpenMP) pragmas generated above the `for` loops.

```
void autoparExample(double x[2000])
{
  int i;
  if (!isInitialized_autoparExample) {
    autoparExample_initialize();
  }
#pragma omp parallel for num_threads(omp_get_max_threads()) private(i)

  for (i = 0; i < 1991; i++) {
    x[i + 9] = sqrt(x[i + 9]);
  }
}
```

The gutter highlighted in green next to the loops shows the part of the code that is parallelized.

```
Function: autoparExample
1 function x = autoparExample(x)
2 %#codegen
3 for i = 10:numel(x)
4     x(i) = sqrt(x(i));
5 end
6 end
```

This loop has been automatically parallelized.

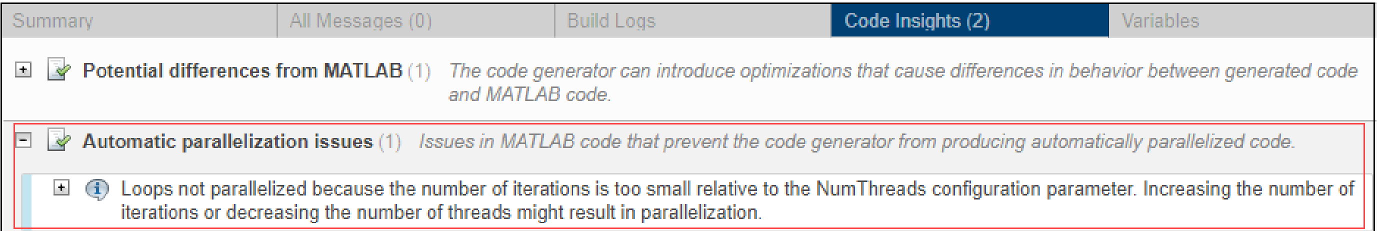
Code Insights

In the **Code Insights** tab, under **Automatic parallelization issues**, you can see detailed information about the `for` loops that were not parallelized in the generated code.

For example, to view a particular code insight, generate code again for the `autoparExample` function that you defined in the previous section. Specify a smaller size for the input arguments.

```
cfg = coder.config('lib');
cfg.EnableAutoParallelization = 1;
x = rand(1,1000);
codegen -config cfg autoparExample -args {x} -report
```

The generated code does not contain parallel `for` loops because the size of the input arguments are smaller than the threshold value for automatic parallelization. Open the report and click **Code Insights > Automatic parallelization issues** to view detailed information about the non-parallelized part of the code.



Disable Automatic Parallelization of a for Loop

You might want to disable automatic parallelization of a particular loop if that loop performs better in serial execution. To prevent parallelization of a specific `for` loop, place the `coder.loop.parallelize('never')` pragma immediately before the loop in your MATLAB code. This pragma overrides the `EnableAutoParallelization` setting. Also, this pragma supports only those `for` loops that are explicitly defined in your MATLAB code. For more information on explicit and implicit loops, see the next section.

For example, the code generator does not parallelize this loop:

```
% Pragma to disable automatic parallelization of for-loops
coder.loop.parallelize('never');
for i = 1:n
    y(i) = y(i)*sin(i);
end
```

See `coder.loop.parallelize`.

Parallelize Implicit for Loops

The example function `autoparExample` used in the previous sections contains an explicit `for` loop. But your MATLAB code can also contain implicit `for` loops that do not appear explicitly in the code that you author. For example, the MATLAB function `mtimes` multiplies two matrices and therefore must perform loop iterations implicitly over the matrix elements.

Automatic parallelization supports loops that are implicit in your MATLAB code. For example, consider this function `autoparExample_implicit`.

```
function y = autoparExample_implicit(y)
%#codegen
y = y * 17; % Generates implicit for loop
end
```

Generate code by running these commands:

```
cfg = coder.config('lib');
cfg.EnableAutoParallelization = 1;
y = rand(1,2000);
codegen -config cfg autoparExample_implicit -args {y} -report
```

Open the report and inspect the generated code. The generated code contains a parallel for loop for the multiplication operation.

```
void autoparExample_implicit(double y[2000])
{
```

```
  int i;
  if (!isInitialized_autoparExample_implicit) {
    autoparExample_implicit_initialize();
  }
#pragma omp parallel for num_threads(omp_get_max_threads()) private(i)

  for (i = 0; i < 2000; i++) {
    y[i] *= 17.0;
  }
```

Usage Notes and Limitations

- **Only outermost loops are parallelized**

  Automatic parallelization applies to the outermost loops in your MATLAB code only. This makes inner loops available for vectorization by the system compiler.
- **`parfor` loops remain as parallel loops**
  - Automatic parallelization honors `parfor` loops that you define and does not convert them into sequential `for` loops.
  - `for` loops that contain `parfor` loops are not parallelized.
- **Loops containing persistent variables are not parallelized automatically**

  Automatic parallelization does not support `for` loops whose bodies contain either persistent variables or functions that access persistent variables.
- **Loops containing external calls are not parallelized automatically**

  Automatic parallelization does not support `for` loops in your code that contain external calls.
- **Function not inlined after automatic parallelization**

  If a `for` loop inside a function is automatically parallelized, the code generator does not inline that function.
- Loops performing reduction operations are not parallelized automatically
- Empty loops and `while` loops are not parallelized automatically

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig` | `coder.config` | `coder.loop.parallelize` | `parfor`

More About

- “How parfor-Loops Improve Execution Speed” on page 32-14
- “Resolve Issue: Array or Variable Access Pattern Not Suitable for Parallel Execution” on page 34-73

Resolve Issue: Array or Variable Access Pattern Not Suitable for Parallel Execution

Issue

A `for` loop is suitable for parallelization if the different loop iterations can be executed independently of each other without affecting the final answer. If one iteration of a `for` loop accesses data written in a different iteration, the iterations cannot be executed independently and the loop cannot be converted to a parallel loop. Suppose that your MATLAB code contains such a loop and you set the configuration parameters `EnableAutoParallelization` and `EnableAutoParallelizationReporting` to `true` while generating code. In such situations, the code generator might produce this message in the **Code Insights** tab of the MATLAB Coder app or the code generation report:

```
Array or variable access pattern inside the loop is not suitable for parallel
execution.
```

Possible Solutions

This code insights message might be triggered either by a scalar access pattern or by an array access pattern that causes an iteration of your `for` loop to depend on another iteration.

Check for Scalar Access Patterns

Suppose that your MATLAB code contains a loop that has a persistent scalar. This is an example of such a loop:

```
for i = 1:numel(x)
    r = sqrt(y(i)) / x(i) * i;
    y(i) = x(i) - r;
end
```

The variable `r` is persistent across loop iterations and prevents parallelization.

Eliminate the persistent scalar

Removing the persistent scalar `r` allows the loop iterations to be executed independently. So, the serial loop can be converted to a parallel loop.

```
for i = 1:numel(x)
    y(i) = x(i) - (sqrt(y(i)) / x(i) * i);
end
```

Convert the persistent scalar to a vector

Converting the persistent scalar `r` to a vector `r(i)` allows the loop iterations to be executed independently. So, the serial loop can be converted to a parallel loop.

```
for i = 1:numel(x)
    r(i) = sqrt(y(i)) / x(i) * i;
    y(i) = x(i) - r(i);
end
```

Check for Array Access Patterns

Suppose that your MATLAB code contains a loop whose one iteration accesses data written in another iteration. This is an example of such a loop:

```
for i = 2:n
  a(i) = a(i+1)
  b(i) = b(i-1)
end
```

- The location `a(i)` that is written in the current iteration `i` was read in the previous iteration `i-1`.
- The location `b(i)` that is written in the current iteration `i` would be read in the next iteration `i+1`.

If the loop is parallelized, such memory access order is not be preserved because every iteration can potentially be executed in parallel. So, the code generator does not automatically parallelize this loop.

To address this issue, try to rewrite your code to avoid such array access patterns.

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig` | `coder.config`

More About

- "Automatically Parallelize for Loops in Generated Code" on page 34-69

Generating Reentrant C Code from MATLAB Code

- “Generate Reentrant C Code from MATLAB Code” on page 35-2
- “Reentrant Code” on page 35-9
- “Specify Generation of Reentrant Code” on page 35-11
- “API for Generated Reusable Code” on page 35-12
- “Call Reentrant Code in a Single-Threaded Environment” on page 35-13
- “Call Reentrant Code in a Multithreaded Environment” on page 35-14
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 35-15
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 35-19
- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 35-24

Generate Reentrant C Code from MATLAB Code

In this section...

"About This Tutorial" on page 35-2
"Copying Files Locally" on page 35-3
"About the Example" on page 35-3
"Providing a C main Function" on page 35-4
"Configuring Build Parameters" on page 35-6
"Generating the C Code" on page 35-6
"Viewing the Generated C Code" on page 35-6
"Running the Code" on page 35-7
"Key Points to Remember" on page 35-7
"Learn More" on page 35-8

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

  MATLAB Coder locates and uses a supported installed compiler. For the current list of supported compilers, see `https://www.mathworks.com/support/compilers/current_release/` on the MathWorks website.

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

Required Files

Type	Name	Description
Function code	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.

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 “Deploy Generated Code” on page 31-71.
- 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 list of generated files, click `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 35-12
Call reentrant code without persistent or global data on UNIX®	“Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 35-15
Call reentrant code with persistent data on Windows	“Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 35-19
Call reentrant code with persistent data on UNIX	“Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 35-24

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.

To deploy the generated code, you must 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 “Deploy Generated Code” on page 31-71.

See Also

Related Examples

- “Specify Generation of Reentrant Code” on page 35-11
- “Generate Reentrant C Code from MATLAB Code” on page 35-2
- “Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 35-15
- “Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 35-19
- “Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 35-24

Specify Generation of Reentrant Code

In this section...
"Specify Generation of Reentrant Code Using the MATLAB Coder App" on page 35-11
"Specify Generation of Reentrant Code Using the Command-Line Interface" on page 35-11

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;
   ```

See Also

Related Examples

- "Generate Reentrant C Code from MATLAB Code" on page 35-2
- "Call Reentrant Code with No Persistent or Global Data (UNIX Only)" on page 35-15
- "Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)" on page 35-19
- "Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)" on page 35-24

More About

- "Reentrant Code" on page 35-9

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 allocate memory for these variables per thread.

For more information on using these global structures, see “Multithreaded Examples” on page 35-14.

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 “Deploy Generated Code” on page 31-71.
- 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 root process. Otherwise, you can allocate memory per thread.
- 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 “Deploy Generated Code” on page 31-71.
- 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 35-2
	UNIX	“Call Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 35-15
Multithreaded with persistent or global data	Windows	“Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 35-19
	UNIX	“Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 35-24

Call Reentrant Code with No Persistent or Global Data (UNIX Only)

In this section...

"Provide a Main Function" on page 35-15
"Generate Reentrant C Code" on page 35-17
"Examine the Generated Code" on page 35-17
"Run the Code" on page 35-18

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 "Deploy Generated Code" on page 31-71.
- 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 “Build Process Customization” on page 27-116.
- 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.

See Also

“Control Stack Space Usage” on page 34-15 | “Stack Allocation and Performance” on page 34-18

Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)

In this section...

"MATLAB Code for This Example" on page 35-19

"Provide a Main Function" on page 35-19

"Generate Reentrant C Code" on page 35-22

"Examine the Generated Code" on page 35-22

"Run the Code" on page 35-23

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 root process if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see "Deploy Generated Code" on page 31-71.
- 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.

See Also

“Control Stack Space Usage” on page 34-15 | “Stack Allocation and Performance” on page 34-18

Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)

In this section...

"MATLAB Code for This Example" on page 35-24
"Provide a Main Function" on page 35-24
"Generate Reentrant C Code" on page 35-27
"Examine the Generated Code" on page 35-28
"Run the Code" on page 35-28

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 root process if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see "Deploy Generated Code" on page 31-71.
- 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 “Build Process Customization” on page 27-116.
- 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')
```

See Also

“Control Stack Space Usage” on page 34-15 | “Stack Allocation and Performance” on page 34-18

Troubleshooting Code Generation Problems

- “JIT MEX Incompatibility Warning” on page 36-2
- “JIT Compilation Does Not Support OpenMP” on page 36-3
- “Output Variable Must Be Assigned Before Run-Time Recursive Call” on page 36-4
- “Compile-Time Recursion Limit Reached” on page 36-7
- “Unable to Determine That Every Element of Cell Array Is Assigned” on page 36-10
- “Nonconstant Index into varargin or varargout in a for-Loop” on page 36-14
- “Unknown Output Type for coder.ceval” on page 36-16
- “MEX Generated on macOS Platform Stays Loaded in Memory” on page 36-18
- “Resolve Error: Code Generator Failed to Produce C++ Destructor for MATLAB Class” on page 36-19

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.

See Also

More About

- "Speed Up MEX Generation by Using JIT Compilation" on page 34-67

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 34-67
- "Algorithm Acceleration Using Parallel for-Loops (parfor)" on page 32-14

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
```

See Also

More About

- "Code Generation for Recursive Functions" on page 20-14

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. 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 36-7
- “Increase the Compile-Time Recursion Limit” on page 36-9

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 36-7.
- 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 36-8.

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.

See Also

More About

- “Code Generation for Recursive Functions” on page 20-14
- “Configure Build Settings” on page 27-13

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 = CellVarSize1D(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 = CellAssign3D(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 9-9.

Solution

Try one of these solutions:

- "Use recognized pattern for assigning elements" on page 36-11
- "Use repmat" on page 36-11
- "Use coder.nullcopy" on page 36-12

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 = repDefine(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 = repVarSize(n, j)
%#codegen
c = cell(1,2);
c{1} = 1;
c{2} = 2;
c1= repmat(c,1,n);
z = c1{j};
end
```

You can pass an initially empty, variable-size cell array into or out of a function by using `repmat`. Use the following pattern:

```
function x = emptyVarSizeCellArray
x = repmat({'abc'},0,0);
coder.varsize('x');
end
```

This code indicates that `x` is an empty, variable-size cell array of `1x3` characters that can be passed into or out of functions.

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 = nulcpyCell(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 9-8

Nonconstant Index into varargin or varargout in a for-Loop

Issue

Your MATLAB code contains a `for`-loop that indexes into `varargin` or `varargout`. 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 36-14
- “Rewrite the Code” on page 36-15

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 19-2
- “Unroll for-Loops” on page 34-33
- “Optimization Strategies” on page 34-3

Unknown Output Type for coder.ceval

Issue

You see this error message:

```
Output of 'coder.ceval' has unknown type. The enclosing
expression cannot be evaluated.
Specify the output type by assigning the output of
'coder.ceval' to a variable with a known type.
```

Cause

This error message occurs when the code generator cannot determine the output type of a `coder.ceval` call.

Solution

Initialize a temporary variable with the expected output type. Assign the output of `coder.ceval` to this variable.

Example

Assume that you have a C function called `cFunctionThatReturnsDouble`. You want to generate C library code for a function `foo`. The code generator returns the error message because it cannot determine the return type of `coder.ceval`.

```
function foo
%#codegen
callFunction(coder.ceval('cFunctionThatReturnsDouble'));
end

function callFunction(~)
end
```

To fix the error, define the type of the C function output by using a temporary variable.

```
function foo
%#codegen
temp = 0;
temp = coder.ceval('cFunctionThatReturnsDouble');
callFunction(temp);
end

function callFunction(~)
end
```

You can also use `coder.opaque` to initialize the temporary variable.

Example Using Classes

Assume that you have a class with a custom `set` method. This class uses the `set` method to ensure that the object property value falls within a certain range.

```
classdef classWithSetter
   properties
      expectedResult = []
   end
   properties(Constant)
      scalingFactor = 0.001
   end
   methods
      function obj = set.expectedResult(obj,erIn)
         if erIn >= 0 && erIn <= 100
            erIn = erIn.*obj.scalingFactor;
            obj.expectedResult = erIn;
         else
            obj.expectedResult = NaN;
         end
      end
   end
end
```

When generating C library code for the function `foo`, the code generator produces the error message. The input type into the `set` method cannot be determined.

```
function foo
%#codegen
obj = classWithSetter;
obj.expectedResult = coder.ceval('cFunctionThatReturnsDouble');
end
```

To fix the error, initialize a temporary variable with a known type. For this example, use a type of scalar double.

```
function foo
%#codegen
obj = classWithSetter;
temp = 0;
temp = coder.ceval('cFunctionThatReturnsDouble');
obj.expectedResult = temp;
end
```

See Also

`coder.ceval` | `coder.opaque`

MEX Generated on macOS Platform Stays Loaded in Memory

Issue

When generating MEX code on the macOS platform, you get one of these messages:

- Warning message:

```
The generated code contains usage of OpenMP thread private variable.
This can cause the MEX to remain loaded in the memory.
```

- Error message:

```
The MEX file 'foo_mex' is still loaded in memory.
To clear the MEX file from memory, close the MATLAB session.
```

Cause

Your MATLAB code contains global or persistent variables that are reachable from the body of a `parfor`-loop. Here is an example MATLAB function that contains this code pattern.

```
function y = foo(x)
y = coder.nullcopy(x);
parfor i = 1:numel(x)
    y(i) = x(i) + sub;
end

function y = sub
persistent t;
if isempty(t)
    t = 2;
end
y = t;
```

When you generate a MEX function for `foo` for the first time, you can receive the warning message.

If you try to overwrite the generated MEX by generating code for `foo` again, you can receive the error message.

Solution

If you receive the warning message, you can use the generated MEX function.

If you receive the error message, close the current MATLAB session to clear the MEX function `foo_mex` from memory. To overwrite the previously generated MEX function , open a new MATLAB session and generate MEX code for `foo`.

See Also

`parfor`

More About

- “Generate Code with Parallel for-Loops (parfor)” on page 34-31

Resolve Error: Code Generator Failed to Produce C++ Destructor for MATLAB Class

Issue

Generating reentrant code with C++ classes from MATLAB code increases the chances of stack overflow during code execution. Code generation might stop and produce this message:

```
Code generator failed to produce C++ destructor for MATLAB class 'y'.
Generated code is not exception-safe. To enable generation of C++ destructor,
disable 'Generate Re-entrant code (MultiInstanceCode)' configuration
parameter.
```

This message might appear if both these conditions are true:

- You choose to generate C++ code with classes from MATLAB code by setting `TargetLang` to 'C++' and `CppPreserveClasses` to `true` in the configuration object (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), or in the project build settings, by setting **Language** to C++and checking the **Generate C++ classes from MATLAB classes** check box.
- You choose to generate reentrant code by enabling the `MultiInstanceCode` parameter in a code configuration object, or by enabling the **Generate re-entrant code** parameter in the **Memory** tab of the MATLAB Coder app.
- The destructor of a class in your MATLAB code has a persistent variable or calls another function that declares and uses a persistent variable.

Possible Solutions

Depending on whether the type of code you want to generate, try one of these solutions.

Raise the Stack Limit

You can raise the stack limit to generate reentrant code that has C++ classes for MATLAB classes. Do one of the following:

- In the project settings dialog box under the **Memory** tab, set the **Stack usage max** parameter.
- In the configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set the `StackUsageMax` parameter.

Raise the stack limit by doubling the stack value until code generation succeeds. The code generator then might have sufficient memory to generate C++ classes in reentrant code.

Note The maximum configurable stack limit depends on the linker in your system. The default stack size varies based on your operating system and system configuration.

The destructor of a class in your MATLAB code must not have a persistent variable or call another function that declares a persistent variable.

Disable Reentrant Code

To generate code that contains C++ classes for MATLAB classes, you can disable generation of reentrant code. Do one of the following:

- In the code configuration object, disable the `MultiInstanceCode` parameter.
- In the MATLAB Coder app, disable the **Generate re-entrant code** parameter.

Generate Structures Instead of Classes

You can change the default behavior of the code generator to produce structures for MATLAB classes. Do one of the following:

- In the configuration object for standalone code generation (`coder.CodeConfig` or `coder.EmbeddedCodeConfig`), set `TargetLang` to `'C++'` and `CppPreserveClasses` to `false`.
- In the MATLAB Coder app, set **Language** to `C++`. In the project build settings, clear the **Generate C++ classes from MATLAB classes** check box.

See Also

More About

- “Generate Reentrant C Code from MATLAB Code” on page 35-2
- “Generate C++ Classes for MATLAB Classes” on page 16-2

Row-Major Array Layout

- “Row-Major and Column-Major Array Layouts” on page 37-2
- “Generate Code That Uses Row-Major Array Layout” on page 37-4

Row-Major and Column-Major Array Layouts

The elements of an array can be stored in column-major layout or row-major layout. For an array stored in column-major layout, the elements of the columns are contiguous in memory. In row-major layout, the elements of the rows are contiguous. Array layout is also called order, format, and representation. The order in which elements are stored can be important for integration, usability, and performance. Certain algorithms perform better on data stored in a particular order.

Programming languages and environments typically assume a single array layout for all data. MATLAB and Fortran use column-major layout by default, whereas C and C++ use row-major layout. With MATLAB Coder, you can generate C/C++ code that uses row-major layout or column-major layout. See “Generate Code That Uses Row-Major Array Layout” on page 37-4.

Array Storage in Computer Memory

Computer memory stores data in terms of one-dimensional arrays. For example, when you declare a 3-by-3 matrix, the software stores this matrix as a one-dimensional array with nine elements. By default, MATLAB stores these elements with a column-major array layout. The elements of each column are contiguous in memory.

Consider the matrix A:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

The matrix A is represented in memory by default with this arrangement:

```
     1     4     7     2     5     8     3     6     9
```

In row-major array layout, the programming language stores row elements contiguously in memory. In row-major layout, the elements of the array are stored as:

```
     1     2     3     4     5     6     7     8     9
```

N-dimensional arrays can also be stored in column-major or row-major layout. In column-major layout, the elements from the first (leftmost) dimension or index are contiguous in memory. In row-major, the elements from the last (rightmost) dimension or index are contiguous.

Conversions Between Different Array Layouts

When you mix row-major data and column-major data in the same code, array layout conversions are necessary. For example, you can generate code that includes row-major and column-major function specializations. Function specializations use one type of array layout for all input, output, and internal data. When passing data between functions, the code generator automatically inserts array layout conversions as needed. Input and output data to generated MEX functions is also converted as needed.

For two-dimensional data, transpose operations convert data between row-major layout and column-major layout. Consider the transposed version of A:

```
A' =
     1     4     7
```

```
     2     5     8
     3     6     9
```

The column-major layout of `A'` matches the row-major layout of `A`. (For complex numbers, array layout conversions use a nonconjugate transpose.)

See Also

`coder.columnMajor` | `coder.isColumnMajor` | `coder.isRowMajor` | `coder.rowMajor`

More About

- “Generate Code That Uses Row-Major Array Layout” on page 37-4
- “MATLAB Data”
- “Generate Code That Uses N-Dimensional Indexing” on page 27-133

Generate Code That Uses Row-Major Array Layout

Array layout can be important for integration, usability, and performance. The code generator produces code that uses column-major layout by default. However, many devices, sensors, and libraries use row-major array layout for their data. You can apply your code directly to this data by generating code that uses row-major layout. Array layout can also affect performance. Many algorithms perform memory access more efficiently for one specific array layout.

You can specify row-major array layout at the command line, with code generation configuration properties, or by using the MATLAB Coder app. You can also specify row-major layout or column-major layout for individual functions and classes. The inputs and outputs of your entry-point (top-level) functions must all use the same array layout.

Specify Row-Major Layout

Consider this function for adding two matrices. The algorithm performs the addition through explicit row and column traversal.

```
function [S] = addMatrix(A,B)
%#codegen
S = zeros(size(A));
for row = 1:size(A,1)
   for col = 1:size(A,2)
       S(row,col) = A(row,col) + B(row,col);
   end
end
```

Generate C code for `addMatrix` by using the `-rowmajor` option. Specify the form of the input parameters by using the `-args` option and launch the code generation report.

```
codegen addMatrix -args {ones(20,10),ones(20,10)} -config:lib -launchreport -rowmajor
```

Alternatively, configure your code for row-major layout by modifying the `RowMajor` parameter in the code generation configuration object. You can use this parameter with any type of configuration object: `lib`, `mex`, `dll`, or `exe`.

```
cfg = coder.config('lib');
cfg.RowMajor = true;
codegen addMatrix -args {ones(20,10),ones(20,10)} -config cfg -launchreport
```

Code generation results in this C code:

```
...
/* generated code for addMatrix using row-major */
for (row = 0; row < 20; row++) {
  for (col = 0; col < 10; col++) {
      S[col + 10 * row] = A[col + 10 * row] + B[col + 10 * row];
   }
}
...
```

To specify row-major layout with the MATLAB Coder app:

1 Open the **Generate** dialog box. On the **Generate Code** page, click the **Generate** arrow ▼.

2. Click **More Settings**.
3. On the **Memory** tab, set **Array layout:** `Row-major`.

To verify that your generated code uses row-major layout, compare the array indexing in your generated code with the array indexing in code that uses column-major layout. You can also generate code that uses N-dimensional indexing. N-dimensional indexing can make differences in array layout more apparent. For more information, see “Generate Code That Uses N-Dimensional Indexing” on page 27-133.

MATLAB stores data in column-major layout by default. When you call a generated MEX function that uses row-major layout, the software automatically converts input data from column-major layout to row-major layout. Output data returned from the MEX function is converted back to column-major layout. For standalone `lib`, `dll`, and `exe` code generation, the code generator assumes that entry-point function inputs and outputs are stored with the same array layout as the function.

Array Layout and Algorithmic Efficiency

For certain algorithms, row-major layout provides more efficient memory access. Consider the C code shown for `addMatrix` that uses row-major layout. The arrays are indexed by the generated code using the formula:

```
[col + 10 * row]
```

Because the arrays are stored in row-major layout, adjacent memory elements are separated by single column increments. The *stride length* for the algorithm is equal to one. The stride length is the distance in memory elements between consecutive memory accesses. A shorter stride length provides more efficient memory access.

Using column-major layout for the data results in a longer stride length and less efficient memory access. To see this comparison, generate code that uses column-major layout:

```
codegen addMatrix -args {ones(20,10),ones(20,10)} -config:lib -launchreport
```

Code generation produces this C code:

```
...
/* generated code for addMatrix using column-major */
for (row = 0; row < 20; row++) {
  for (col = 0; col < 10; col++) {
     S[row + 20 * col] = A[row + 20 * col] + B[row + 20 * col];
  }
}
...
```

In column-major layout, the column elements are contiguous in memory in the generated code. Adjacent memory elements are separated by single row increments and indexed by the formula:

```
[row + 20 * col]
```

However, the algorithm iterates through the columns in the inner for-loop. Therefore, the column-major C code must make a stride of 20 elements for each consecutive memory access.

The array layout that provides the most efficient memory access depends on the algorithm. For this algorithm, row-major layout of the data provides more efficient memory access. The algorithm traverses over the data row by row. Row-major storage is therefore more efficient.

Row-Major Layout for N-Dimensional Arrays

You can use row-major layout for N-dimensional arrays. When an array is stored in row-major layout, the elements from the last (rightmost) dimension or index are contiguous in memory. In column-major layout, the elements from the first (leftmost) dimension or index are contiguous.

Consider the example function `addMatrix3D`, which accepts three-dimensional inputs.

```
function [S] = addMatrix3D(A,B)
%#codegen
S = zeros(size(A));
for i = 1:size(A,1)
    for j = 1:size(A,2)
        for k = 1:size(A,3)
            S(i,j,k) = A(i,j,k) + B(i,j,k);
        end
    end
end
end
```

Generate code that uses row-major layout:

```
codegen addMatrix3D -args {ones(20,10,5),ones(20,10,5)} -config:lib -launchreport -rowmajor
```

The code generator produces this C code:

```
...
/* row-major layout */
for (i = 0; i < 20; i++) {
    for (j = 0; j < 10; j++) {
        for (k = 0; k < 5; k++) {
            S[(k + 5 * j) + 50 * i] = A[(k + 5 * j) + 50 * i]
                                      + B[(k + 5 * j) + 50 * i];
        }
    }
}
...
```

In row-major layout, adjacent memory elements are separated by single increments of the last index, `k`. The inner for-loop iterates over adjacent elements separated by only one position in memory. Compare the differences to generated code that uses column-major layout:

```
...
/* column-major layout */
for (i = 0; i < 20; i++) {
    for (j = 0; j < 10; j++) {
        for (k = 0; k < 5; k++) {
            S[(i + 20 * j) + 200 * k] = A[(i + 20 * j) + 200 * k]
                                        + B[(i + 20 * j) + 200 * k];
        }
    }
}
...
```

In column-major layout, adjacent elements are separated by single increments of the first index, `i`. The inner for-loop now iterates over adjacent elements separated by 200 positions in memory. The long stride length can cause performance degradation due to cache misses.

Because the algorithm iterates through the last index, k, in the inner for-loop, the stride length is much longer for the generated code that uses column-major layout. For this algorithm, row-major layout of the data provides more efficient memory access.

Specify Array Layout in External Function Calls

To call external C/C++ functions that expect data stored with a specific layout, use `coder.ceval` with the `layout` syntax. If you do not use this syntax, the external function inputs and outputs are assumed to use column-major layout by default.

Consider an external C function designed to use row-major layout called `myCFunctionRM`. To integrate this function into your code, call the function using the `'-layout:rowMajor'` or `'-row'` option. This option ensures that the input and output arrays are stored in row-major order. The code generator automatically inserts array layout conversions as needed.

```
coder.ceval('-layout:rowMajor','myCFunctionRM',coder.ref(in),coder.ref(out))
```

Within a MATLAB function that uses row-major layout, you may seek to call an external function designed to use column-major layout. In this case, use the `'-layout:columnMajor'` or `'-col'` option.

```
coder.ceval('-layout:columnMajor','myCFunctionCM',coder.ref(in),coder.ref(out))
```

You can perform row-major and column-major function calls in the same code. Consider the function `myMixedFn1` as an example:

```
function [E] = myMixedFn1(x,y)
%#codegen
% specify type of return arguments for ceval calls
D = zeros(size(x));
E = zeros(size(x));

% include external C functions that use row-major & column-major
coder.cinclude('addMatrixRM.h');
coder.updateBuildInfo('addSourceFiles', 'addMatrixRM.c');
coder.cinclude('addMatrixCM.h');
coder.updateBuildInfo('addSourceFiles', 'addMatrixCM.c');

% call C function that uses row-major order
coder.ceval('-layout:rowMajor','addMatrixRM', ...
    coder.rref(x),coder.rref(y),coder.wref(D));

% call C function that uses column-major order
coder.ceval('-layout:columnMajor','addMatrixCM', ...
    coder.rref(x),coder.rref(D),coder.wref(E));
end
```

The external files are:

addMatrixRM.h

```
extern void addMatrixRM(const double x[200], const double y[200], double z[200]);
```

addMatrixRM.c

```
#include "addMatrixRM.h"
```

```
void addMatrixRM(const double x[200], const double y[200], double z[200])
{
  int row;
  int col;

  /* add two matrices */
  for (row = 0; row < 20; row++) {
    /* row by row */
    for (col = 0; col < 10; col++) {
      /* each element in current row */
      z[col + 10 * row] = x[col + 10 * row] + y[col + 10 * row];
    }
  }
}
```

addMatrixCM.h

```
extern void addMatrixCM(const double x[200], const double y[200], double z[200]);
```

addMatrixCM.c

```
#include "addMatrixCM.h"

void addMatrixCM(const double x[200], const double y[200], double z[200])
{
  int row;
  int col;

  /* add two matrices */
  for (row = 0; row < 20; row++) {
    /* row by row */
    for (col = 0; col < 10; col++) {
      /* each element in current row */
      z[row + 20 * col] = x[row + 20 * col] + y[row + 20 * col];
    }
  }
}
```

To generate code, enter:

```
codegen -config:lib myMixedFn1 -args {ones(20,10),ones(20,10)} -rowmajor -launchreport
```

See Also

codegen | coder.ceval | coder.columnMajor | coder.isColumnMajor | coder.isRowMajor | coder.rowMajor

More About

- “Row-Major and Column-Major Array Layouts” on page 37-2
- “Specify Array Layout in Functions and Classes” on page 5-17
- “Code Design for Row-Major Array Layout” on page 5-21
- “Generate Code That Uses N-Dimensional Indexing” on page 27-133

Deep Learning with MATLAB Coder

- “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7
- “Networks and Layers Supported for Code Generation” on page 38-8
- “Load Pretrained Networks for Code Generation” on page 38-23
- “Generate Generic C/C++ Code for Deep Learning Networks” on page 38-26
- “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Cross-Compile Deep Learning Code That Uses ARM Compute Library” on page 38-37
- “Code Generation for Quantized Deep Learning Networks” on page 38-40
- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” on page 38-42
- “Deep Learning Prediction with ARM Compute Using codegen” on page 38-51
- “Code Generation for Deep Learning on ARM Targets” on page 38-56
- “Generate C++ Code for Object Detection Using YOLO v2 and Intel MKL-DNN” on page 38-61
- “Code Generation and Deployment of MobileNet-v2 Network to Raspberry Pi” on page 38-64
- “Code Generation for Semantic Segmentation Application on Intel CPUs That Uses U-Net” on page 38-68
- “Code Generation for Semantic Segmentation Application on ARM® Neon targets That Uses U-Net” on page 38-77
- “Code Generation for LSTM Network on Raspberry Pi” on page 38-86
- “Code Generation for LSTM Network That Uses Intel MKL-DNN” on page 38-93
- “Code Generation for Convolutional LSTM Network That Uses Intel MKL-DNN” on page 38-97
- “Cross Compile Deep Learning Code for ARM Neon Targets” on page 38-101
- “Code Generation for Quantized Deep Learning Network on Raspberry Pi” on page 38-107
- “Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning” on page 38-115
- “Generate Digit Images Using Variational Autoencoder on Intel CPUs” on page 38-124

Prerequisites for Deep Learning with MATLAB Coder

MathWorks Products

To use MATLAB Coder to generate code for deep learning networks, you must also install:

- Deep Learning Toolbox™
- MATLAB Coder Interface for Deep Learning Libraries

The MATLAB Coder Interface for Deep Learning Libraries is not supported for MATLAB Online.

Third-Party Hardware and Software

You can use MATLAB Coder to generate C++ code for deep learning networks that you deploy to Intel or ARM processors. The generated code takes advantage of deep learning libraries optimized for the target CPU. The hardware and software requirements depend on the target platform.

You can also use MATLAB Coder to generate generic C or C++ code for deep learning networks. Such C or C++ code does not depend on any third-party libraries. For more information, see “Generate Generic C/C++ Code for Deep Learning Networks” on page 38-26.

Note The paths to the required software libraries must not contain spaces or special characters, such as parentheses. On Windows operating systems, special characters and spaces are allowed only if 8.3 file names are enabled. For more information on 8.3 file names, refer to the Windows documentation.

	Intel CPUs	ARM CPUs
Hardware Requirements	Intel processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions.	ARM Cortex-A processors that support the NEON extension.

	Intel CPUs	ARM CPUs
Software Libraries	Intel Math Kernel Library for Deep Neural Networks (MKL-DNN), v1.4. See https://01.org/mkl-dnn Do not use a prebuilt library because some required files are missing. Instead, build the library from the source code. See instructions for building the library on GitHub®. For more information on build, see this post in MATLAB Answers™: https://www.mathworks.com/matlabcentral/answers/447387-matlab-coder-how-do-i-build-the-intel-mkl-dnn-library-for-deep-learning-c-code-generation-and-dep	ARM Compute Library for computer vision and machine learning, versions 18.11, 19.02, 19.05, and 20.02.1. See https://developer.arm.com/technologies/compute-library Specify the version number in a `coder.ARMNEONConfig` configuration object. The default version number is v20.02.1. Do not use a prebuilt library because it might be incompatible with the compiler on the ARM hardware. Instead, build the library from the source code. Build the library on either your host machine or directly on the target hardware. See instructions for building the library on GitHub. The folder that contains the library files such as `libarm_compute.so` should be named `lib`. If the folder is named `build`, rename the folder to `lib`. For more information on build, see this post in MATLAB Answers: https://www.mathworks.com/matlabcentral/answers/455590-matlab-coder-how-do-i-build-the-arm-compute-library-for-deep-learning-c-code-generation-and-deplo
Operating System Support	Windows, Linux, and macOS.	Windows and Linux only.

<table>
<tr><th></th><th>Intel CPUs</th><th>ARM CPUs</th></tr>
<tr><td>C++ Compiler</td><td colspan="2">MATLAB Coder locates and uses a supported installed compiler. For the list of supported compilers, see Supported and Compatible Compilers on the MathWorks website.

You can use <code>mex -setup</code> to change the default compiler. See “Change Default Compiler”.

The C++ compiler must support C++11.

On Windows, to generate code that uses the Intel MKL-DNN library by using the <code>codegen</code> command, use Microsoft Visual Studio 2015 or later.

On Windows, to generate generic C or C++ code that does not use any third-party libraries, use Microsoft Visual Studio or the MinGW® compiler. For more information, see “Generate Generic C/C++ Code for Deep Learning Networks” on page 38-26.</td></tr>
<tr><td>Other</td><td colspan="2">Open Source Computer Vision Library (OpenCV), v3.1.0 is required for the ARM based deep learning examples.

Note: The examples require separate libraries such as <code>opencv_core.lib</code> and <code>opencv_video.lib</code>. The <code>OpenCV</code> library that ships with Computer Vision Toolbox does not have the required libraries and the OpenCV installer does not install them. Therefore, you must download the <code>OpenCV</code> source and build the libraries.

For more information, refer to the OpenCV documentation.</td></tr>
</table>

Environment Variables

MATLAB Coder uses environment variables to locate the libraries required to generate code for deep learning networks.

<table>
<tr><th>Platform</th><th>Variable Name</th><th>Description</th></tr>
<tr><td rowspan="2">Windows</td><td><code>INTEL_MKLDNN</code></td><td>Path to the root folder of the Intel MKL-DNN library installation.

For example:

<code>C:\Program Files\mkl-dnn</code></td></tr>
<tr><td><code>ARM_COMPUTELIB</code></td><td>Path to the root folder of the ARM Compute Library installation on the ARM target hardware.

For example:

<code>/usr/local/arm_compute</code>

Set <code>ARM_COMPUTELIB</code> on the ARM target hardware.</td></tr>
</table>

Platform	Variable Name	Description
	`PATH`	Path to the Intel MKL-DNN library folder. For example: `C:\Program Files\mkl-dnn\lib`
Linux	`LD_LIBRARY_PATH`	Path to the Intel MKL-DNN library folder. For example: `/usr/local/mkl-dnn/lib/`
		Path to the ARM Compute Library folder on the target hardware. For example: `/usr/local/arm_compute/lib/` Set `LD_LIBRARY_PATH` on the ARM target hardware.
	`INTEL_MKLDNN`	Path to the root folder of the Intel MKL-DNN library installation. For example: `/usr/local/mkl-dnn/`
	`ARM_COMPUTELIB`	Path to the root folder of the ARM Compute Library installation on the ARM target hardware. For example: `/usr/local/arm_compute/` Set `ARM_COMPUTELIB` on the ARM target hardware.
macOS	`INTEL_MKLDNN`	Path to the root folder of the Intel MKL-DNN library installation. For example: `/usr/local/mkl-dnn`
UNIX based OS on ARM targets	`OPENCV_DIR`	Path to the build folder of OpenCV. Install OpenCV for deep learning examples that use OpenCV. For example: `/usr/local/opencv/build`

Note To generate code for Raspberry Pi using the MATLAB Support Package for Raspberry Pi Hardware, you must set the environment variables non-interactively. For instructions, see https://www.mathworks.com/matlabcentral/answers/455591-matlab-coder-how-do-i-setup-the-environment-variables-on-arm-targets-to-point-to-the-arm-compute-li

Note To build and run examples that use OpenCV, you must install the OpenCV libraries on the target board. For OpenCV installations on Linux, make sure that the path to the library files and the path to the header files are on the system path. By default, the library and header files are installed in a standard location such as `/usr/local/lib/` and `/usr/local/include/opencv`, respectively.

For OpenCV installations on the target board, set the `OPENCV_DIR` and `PATH` environment variables as described in the previous table.

See Also

More About

- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7

Workflow for Deep Learning Code Generation with MATLAB Coder

With MATLAB Coder, you can generate code for prediction from a pretrained convolutional neural network (CNN), targeting an embedded platform that uses an Intel processor or an ARM processor. The generated code calls the Intel MKL-DNN or ARM Compute Library to apply high performance.

You can also use MATLAB Coder to generate generic C or C++ code for deep learning networks. Such C or C++ code does not depend on any third-party libraries.

1 Get a trained network by using Deep Learning Toolbox. Construct and train the network or use a pretrained network. For more information, see:

   - “Deep Learning in MATLAB” (Deep Learning Toolbox).
   - “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

   The network must be supported for code generation. See “Networks and Layers Supported for Code Generation” on page 38-8.
2 Load a network object from the trained network.

   See “Load Pretrained Networks for Code Generation” on page 38-23.
3 Generate C++ code for the trained network by using `codegen` or the MATLAB Coder app. See:

   - “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
   - “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
   - “Generate Generic C/C++ Code for Deep Learning Networks” on page 38-26

See Also

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Learn About Convolutional Neural Networks” (Deep Learning Toolbox)
- “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2
- “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” on page 38-42
- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Deep Learning on ARM Targets” on page 38-56
- “Deep Learning Prediction with ARM Compute Using codegen” on page 38-51
- “Generate Generic C/C++ Code for Deep Learning Networks” on page 38-26
- “Deep Learning with GPU Coder” (GPU Coder)

Networks and Layers Supported for Code Generation

MATLAB Coder supports code generation for series, directed acyclic graph (DAG), and recurrent convolutional neural networks (CNNs or ConvNets). You can generate code for any trained convolutional neural network whose layers are supported for code generation. See "Supported Layers" on page 38-9.

Supported Pretrained Networks

The following pretrained networks, available in Deep Learning Toolbox, are supported for code generation.

Network Name	Description	ARM Compute Library	Intel MKL-DNN
AlexNet	AlexNet convolutional neural network. For the pretrained AlexNet model, see alexnet.	Yes	Yes
DarkNet	DarkNet-19 and DarkNet-53 convolutional neural networks. For the pretrained DarkNet models, see darknet19 and darknet53.	Yes	Yes
DenseNet-201	DenseNet-201 convolutional neural network. For the pretrained DenseNet-201 model, see densenet201.	Yes	Yes
EfficientNet-b0	EfficientNet-b0 convolutional neural network. For the pretrained EfficientNet-b0 model, see efficientnetb0.	Yes	Yes
GoogLeNet	GoogLeNet convolutional neural network. For the pretrained GoogLeNet model, see googlenet.	Yes	Yes
Inception-ResNet-v2	Inception-ResNet-v2 convolutional neural network. For the pretrained Inception-ResNet-v2 model, see inceptionresnetv2.	Yes	Yes
Inception-v3	Inception-v3 convolutional neural network. For the pretrained Inception-v3 model, see inceptionv3.	Yes	Yes
MobileNet-v2	MobileNet-v2 convolutional neural network. For the pretrained MobileNet-v2 model, see mobilenetv2.	Yes	Yes
NASNet-Large	NASNet-Large convolutional neural network. For the pretrained NASNet-Large model, see nasnetlarge.	Yes	Yes
NASNet-Mobile	NASNet-Mobile convolutional neural network. For the pretrained NASNet-Mobile model, see nasnetmobile.	Yes	Yes
ResNet	ResNet-18, ResNet-50, and ResNet-101 convolutional neural networks. For the pretrained ResNet models, see resnet18, resnet50, and resnet101.	Yes	Yes

Network Name	Description	ARM Compute Library	Intel MKL-DNN
`SegNet`	Multi-class pixelwise segmentation network. For more information, see `segnetLayers`.	No	Yes
`SqueezeNet`	Small, deep neural network. For the pretrained SqeezeNet models, see `squeezenet`.	Yes	Yes
`VGG-16`	VGG-16 convolutional neural network. For the pretrained VGG-16 model, see `vgg16`.	Yes	Yes
`VGG-19`	VGG-19 convolutional neural network. For the pretrained VGG-19 model, see `vgg19`.	Yes	Yes
`Xception`	Xception convolutional neural network. For the pretrained Xception model, see `xception`.	Yes	Yes

Supported Layers

The following layers are supported for code generation by MATLAB Coder for the target deep learning libraries specified in the table.

Once you install the support package MATLAB Coder Interface for Deep Learning Libraries, you can use `coder.getDeepLearningLayers` to see a list of the layers supported for a specific deep learning library. For example:

```
coder.getDeepLearningLayers('mkldnn')
```

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`additionLayer`	Addition layer	Yes	Yes	Yes
`anchorBoxLayer`	Anchor box layer	Yes	Yes	No
`averagePooling2dLayer`	Average pooling layer	Yes	Yes	No
`batchNormalizationLayer`	Batch normalization layer	Yes	Yes	No
`bilstmLayer`	Bidirectional LSTM layer	Yes	Yes	Yes
`classificationLayer`	Create classification output layer	Yes	Yes	Yes
`clippedReluLayer`	Clipped Rectified Linear Unit (ReLU) layer	Yes	Yes	No
`concatenationLayer`	Concatenation layer	Yes	Yes	No

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`convolution2dLayer`	2-D convolution layer • For code generation, the `PaddingValue` parameter must be equal to `0`, which is the default value.	Yes	Yes	No
`crop2dLayer`	Layer that applies 2-D cropping to the input	Yes	Yes	No
`CrossChannelNormalizationLayer`	Channel-wise local response normalization layer	Yes	Yes	No

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
Custom layers	Custom layers, with or without learnable parameters, that you define for your problem. See: • "Define Custom Deep Learning Layers" (Deep Learning Toolbox) • "Define Custom Deep Learning Layer for Code Generation" (Deep Learning Toolbox) • "Networks and Layers Supported for Code Generation" on page 38-8 The outputs of the custom layer must be fixed-size arrays. Custom layers in sequence networks are not supported for code generation. For code generation, custom layers must contain the `%#codegen` pragma.	Yes	Yes	Yes

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
Custom output layers	All output layers including custom classification or regression output layers created by using `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`. For an example showing how to define a custom classification output layer and specify a loss function, see “Define Custom Classification Output Layer” (Deep Learning Toolbox). For an example showing how to define a custom regression output layer and specify a loss function, see “Define Custom Regression Output Layer” (Deep Learning Toolbox).	Yes	Yes	Yes
`depthConcatenationLayer`	Depth concatenation layer	Yes	Yes	No
`depthToSpace2dLayer`	2-D depth to space layer	Yes	Yes	Yes
`dicePixelClassificationLayer`	A Dice pixel classification layer provides a categorical label for each image pixel or voxel using generalized Dice loss.	Yes	Yes	No

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`dropoutLayer`	Dropout layer	Yes	Yes	Yes
`eluLayer`	Exponential linear unit (ELU) layer	Yes	Yes	No
`featureInputLayer`	Feature input layer	Yes	Yes	Yes
`flattenLayer`	Flatten layer	Yes	Yes	No
`focalLossLayer`	A focal loss layer predicts object classes using focal loss.	Yes	Yes	No
`fullyConnectedLayer`	Fully connected layer	Yes	Yes	Yes
`globalAveragePooling2dLayer`	Global average pooling layer for spatial data	Yes	Yes	No
`globalMaxPooling2dLayer`	2-D global max pooling layer	Yes	Yes	No
`groupedConvolution2dLayer`	2-D grouped convolutional layer • For code generation, the `PaddingValue` parameter must be equal to `0`, which is the default value.	Yes • If you specify an integer for `numGroups`, then the value must be less than or equal to `2`.	Yes	No
`gruLayer`	Gated recurrent unit (GRU) layer	Yes	Yes	Yes
`imageInputLayer`	Image input layer • Code generation does not support `'Normalization'` specified using a function handle.	Yes	Yes	Yes
`leakyReluLayer`	Leaky Rectified Linear Unit (ReLU) layer	Yes	Yes	No
`lstmLayer`	Long short-term memory (LSTM) layer	Yes	Yes	Yes

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
maxPooling2dLayer	Max pooling layer	Yes	Yes	No
maxUnpooling2dLayer	Max unpooling layer	No	Yes	No
multiplicationLayer	Multiplication layer	Yes	Yes	Yes
pixelClassificationLayer	Create pixel classification layer for semantic segmentation	Yes	Yes	No
rcnnBoxRegressionLayer	Box regression layer for Fast and Faster R-CNN	Yes	Yes	No
rpnClassificationLayer	Classification layer for region proposal networks (RPNs)	Yes	Yes	No
regressionLayer	Create a regression output layer	Yes	Yes	Yes
reluLayer	Rectified Linear Unit (ReLU) layer	Yes	Yes	Yes
resize2dLayer	2-D resize layer	Yes	Yes	Yes
scalingLayer	Scaling layer for actor or critic network	Yes	Yes	No
sigmoidLayer	Sigmoid layer	Yes	Yes	Yes
sequenceFoldingLayer	Sequence folding layer	Yes	Yes	No

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`sequenceInputLayer`	Sequence input layer • For vector sequence inputs, the number of features must be a constant during code generation. • Code generation does not support `'Normalization'` specified using a function handle.	Yes	Yes	Yes
`sequenceUnfoldingLayer`	Sequence unfolding layer	Yes	Yes	No
`softmaxLayer`	Softmax layer	Yes	Yes	Yes
`softplusLayer`	Softplus layer for actor or critic network	Yes	Yes	Yes
`spaceToDepthLayer`	Space to depth layer	Yes	Yes	No
`ssdMergeLayer`	SSD merge layer for object detection	Yes	Yes	No
`nnet.keras.layer.FlattenCStyleLayer`	Flattens activations into 1-D assuming C-style (row-major) order	Yes	Yes	No
`nnet.keras.layer.GlobalAveragePooling2dLayer`	Global average pooling layer for spatial data	Yes	Yes	No
`nnet.keras.layer.SigmoidLayer`	Sigmoid activation layer	Yes	Yes	No
`nnet.keras.layer.TanhLayer`	Hyperbolic tangent activation layer	Yes	Yes	No
`nnet.keras.layer.ZeroPadding2dLayer`	Zero padding layer for 2-D input	Yes	Yes	No

Layer Name	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
nnet.onnx.layer.ElementwiseAffineLayer	Layer that performs element-wise scaling of the input followed by an addition	Yes	Yes	No
nnet.onnx.layer.FlattenLayer	Flatten layer for ONNX™ network	Yes	Yes	No
nnet.onnx.layer.IdentityLayer	Layer that implements ONNX identity operator	Yes	Yes	No
tanhLayer	Hyperbolic tangent (tanh) layer	Yes	Yes	Yes
transposedConv2dLayer	Transposed 2-D convolution layer Code generation does not support asymmetric cropping of the input. For example, specifying a vector [t b l r] for the 'Cropping' parameter to crop the top, bottom, left, and right of the input is not supported.	Yes	Yes	No
wordEmbeddingLayer	A word embedding layer maps word indices to vectors	Yes	Yes	No
yolov2OutputLayer	Output layer for YOLO v2 object detection network	Yes	Yes	No
yolov2ReorgLayer	Reorganization layer for YOLO v2 object detection network	Yes	Yes	No
yolov2TransformLayer	Transform layer for YOLO v2 object detection network	Yes	Yes	No

Supported Classes

Class	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`DAGNetwork`	Directed acyclic graph (DAG) network for deep learning • Only the `activations`, `predict`, and `classify` methods are supported.	Yes	Yes	Yes

Class	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`dlnetwork`	Deep learning network for custom training loops • Code generation supports only the `InputNames` and `OutputNames` properties. • Code generation does not support `dlnetwork` objects without input layers. • Code generation for `dlnetwork` objects with `sequenceInputLayer` objects is not supported. • Code generation supports only the `predict` object function. The `dlarray` input to the `predict` method must be a `single` datatype.	Yes	Yes	No

Class	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`SeriesNetwork`	Series network for deep learning • Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, `classifyAndUpdateState`, and `resetState` object functions are supported.	Yes	Yes	Yes

Class	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`yolov2ObjectDetector`	• Only the `detect` method of the `yolov2ObjectDetector` is supported for code generation. • The `roi` argument to the `detect` method must be a code generation constant (`coder.const()`) and a 1x4 vector. • Only the `Threshold`, `SelectStrongest`, `MinSize`, and `MaxSize` name-value pairs for `detect` are supported. • The `labels` output of `detect` is returned as a cell array of character vectors, for example, `{'car','bus'}`.	Yes	Yes	No

Class	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
`ssdObjectDetector`	Object to detect objects using the SSD-based detector. • Only the `detect` method of the `ssdObjectDetector` is supported for code generation. • The `roi` argument to the `detect` method must be a codegen constant (`coder.const()`) and a 1x4 vector. • Only the `Threshold`, `SelectStrongest`, `MinSize`, `MaxSize`, and `MiniBatchSize` Name-Value pairs are supported. All Name-Value pairs must be compile-time constants. • The channel and batch size of the input image must be fixed size. • The `labels` output is returned as a categorical array. • In the generated code, the input is	Yes	Yes	No

Class	Description	ARM Compute Library	Intel MKL-DNN	Generic C/C++
	rescaled to the size of the input layer of the network. But the bounding box that the `detect` method returns is in reference to the original input size.			

See Also

`coder.getDeepLearningLayers`

More About

- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)
- “Learn About Convolutional Neural Networks” (Deep Learning Toolbox)
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7

Load Pretrained Networks for Code Generation

You can generate code for a pretrained convolutional neural network (CNN). To provide the network to the code generator, load a `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object from the trained network.

Load a Network by Using coder.loadDeepLearningNetwork

You can load a network object from any network that is supported for code generation by using `coder.loadDeepLearningNetwork`. You can specify the network from a MAT-file. The MAT-file must contain only the network to be loaded.

For example, suppose that you create a trained network object called `myNet` by using the `trainNetwork` function. Then, you save the workspace by entering `save`. This creates a file called `matlab.mat` that contains the network object. To load the network object `myNet`, enter:

```
net = coder.loadDeepLearningNetwork('matlab.mat');
```

You can also specify the network by providing the name of a function that returns a pretrained `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, or `ssdObjectDetector` object, such as:

- `alexnet`
- `densenet201`
- `googlenet`
- `inceptionv3`
- `mobilenetv2`
- `resnet18`
- `resnet50`
- `resnet101`
- `squeezenet`
- `vgg16`
- `vgg19`
- `xception`

For example, load a network object by entering:

```
net = coder.loadDeepLearningNetwork('googlenet');
```

The Deep Learning Toolbox functions in the previous list require that you install a support package for the function. See “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

Specify a Network Object for Code Generation

If you generate code by using `codegen` or the app, load the network object inside of your entry-point function by using `coder.loadDeepLearningNetwork`. For example:

```
function out = myNet_predict(in) %#codegen

persistent mynet;

if isempty(mynet)
```

```
    mynet = coder.loadDeepLearningNetwork('matlab.mat');
end
out = predict(mynet,in);
```

For pretrained networks that are available as support package functions such as `alexnet`, `inceptionv3`, `googlenet`, and `resnet`, you can directly specify the support package function, for example, by writing `mynet = googlenet`.

Next, generate code for the entry-point function. For example:

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -args {ones(224,224,3,'single')} -config cfg myNet_predict
```

Specify a dlnetwork Object for Code Generation

Suppose you have a pretrained `dlnetwork` network object in the `mynet.mat` MAT-file. To predict the responses for this network, create an entry-point function in MATLAB as shown in this code.

```
function a = myDLNet_predict(in)
dlIn = dlarray(in, 'SSC');

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dlA = predict(dlnet, dlIn);

a = extractdata(dlA);

end
```

In this example, the input and output to `myDLNet_predict` are of simpler datatypes and the `dlarray` object is created within the function. The `extractdata` method of the `dlarray` object returns the data in the `dlarray` `dlA` as the output of `myDLNet_predict`. The output `a` has the same data type as the underlying data type in `dlA`. This entry-point design has the following advantages:

- Easier integration with standalone code generation workflows such as static, dynamic libraries, or executables.
- The data format of the output from the `extractdata` function has the same order (`'SCBTU'`) in both the MATLAB environment and the generated code.
- Improves performance for MEX workflows.
- Simplifies Simulink workflows using MATLAB Function blocks as Simulink does not natively support `dlarray` objects.

Next, generate code for the entry-point function. For example:

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -args {ones(224,224,3,'single')} -config cfg myDLNet_predict
```

See Also

Functions
`codegen` | `coder.loadDeepLearningNetwork` | `trainNetwork`

Objects
`DAGNetwork` | `SeriesNetwork` | `dlarray` | `dlnetwork` | `ssdObjectDetector` | `yolov2ObjectDetector`

More About

- “Networks and Layers Supported for Code Generation” on page 38-8
- “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32

Generate Generic C/C++ Code for Deep Learning Networks

With MATLAB Coder, you can generate generic C or C++ code for prediction from an already trained convolutional neural network (CNN). The generated C/C++ code does not depend on any third-party libraries. The generated code implements a CNN with the architecture, layers, and parameters specified in the input `SeriesNetwork` or `DAGNetwork` network object. See “Networks and Layers Supported for Code Generation” on page 38-8.

Generate code by using one of these methods:

- The standard `codegen` command for C/C++ code generation from MATLAB code.
- The MATLAB Coder app.

Requirements

- On Windows, code generation for deep learning networks with the `codegen` function requires Microsoft Visual Studio or the MinGW compiler.
- MATLAB Coder Interface for Deep Learning Libraries. To install this support package, select it from the MATLAB **Add-Ons** menu.
- Deep Learning Toolbox.

Code Generation by Using codegen

1 Write an entry-point function in MATLAB that:

- Uses the `coder.loadDeepLearningNetwork` function to construct and set up a CNN network object. For more information, see “Load Pretrained Networks for Code Generation” on page 38-23.
- Calls the `predict` method of the network on the entry-point function input.
- Specifies a `MiniBatchSize` in the `predict` method to manage memory usage for prediction on multiple input images or observations.

For example:

```
function out = my_predict(in) %#codegen

% A persistent object mynet is used to load the series network object.
% At the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is reused
% to call predict on inputs, thus avoiding reconstructing and reloading the
% network object.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('myNetwork.mat');
end

% pass in input
out = predict(mynet,in,'MiniBatchSize',2);
```

2 Create a deep learning configuration object `dlconfig` that is configured for generating generic C/C++ code by using the `coder.DeepLearningConfig` function.

```
dlconfig = coder.DeepLearningConfig(TargetLibrary='none');
```

Create a code generation configuration object for MEX or for a static or dynamically linked library. By default, the code generator produces generic C code. To produce generic C++ code,

in your code generation configuration object, set the `TargetLang` parameter to `'C++'`. Set the `DeepLearningConfig` parameter to the previously created object `dlconfig`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = dlconfig;
```

3. Run the `codegen` command. Use the `-config` option to specify the configuration object. Use the `-args` option to specify the input type.

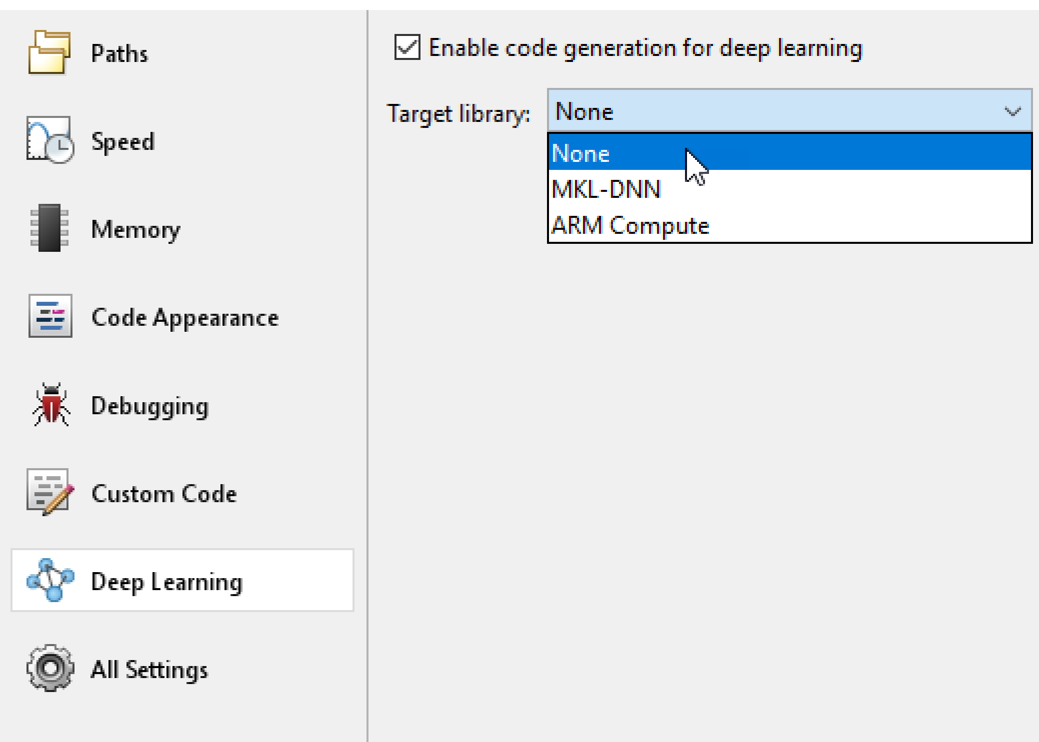
```
codegen -config cfg my_predict -args {myInput} -report
```

Note You can specify half-precision inputs for code generation. However, the code generator type casts the inputs to single-precision. The Deep Learning Toolbox uses single-precision, floating-point arithmetic for all computations in MATLAB.

Code Generation by Using the MATLAB Coder App

1. Follow the usual steps for specifying the entry-point function and specifying input types. See “Generate C Code by Using the MATLAB Coder App”.
2. In the **Generate Code** step:

   - Set **Language** to either **C** or **C++**.
   - Click **More Settings**. In the **Deep Learning** pane, set **Target library** to `None`.



3. Generate code.

See Also

`codegen` | `coder.DeepLearningConfig` | `coder.loadDeepLearningNetwork`

Related Examples

- “Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning” on page 38-115
- “Generate C Code by Using the MATLAB Coder App”

Code Generation for Deep Learning Networks with MKL-DNN

With MATLAB Coder, you can generate code for prediction from an already trained convolutional neural network (CNN), targeting an embedded platform that uses an Intel processor. The code generator takes advantage of the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). The generated code implements a CNN with the architecture, layers, and parameters specified in the input `SeriesNetwork` or `DAGNetwork` network object.

Generate code by using one of these methods:

- The standard `codegen` command for C/C++ code generation from MATLAB code.
- The MATLAB Coder app.

Requirements

- On Windows, code generation for deep learning networks with the `codegen` function requires Microsoft Visual Studio 2015 or later.
- MATLAB Coder Interface for Deep Learning Libraries. To install this support package, select it from the MATLAB **Add-Ons** menu.
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Deep Learning Toolbox.
- Environment variables for the compilers and libraries. For more information, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

Code Generation by Using codegen

1. Write an entry-point function in MATLAB that:

   - Uses the `coder.loadDeepLearningNetwork` function to construct and set up a CNN network object. For more information, see “Load Pretrained Networks for Code Generation” on page 38-23.
   - Calls the `predict` method of the network on the entry-point function input.
   - Specifies a `MiniBatchSize` in the `predict` method to manage memory usage for prediction on multiple input images or observations.

   For example:

   ```
   function out = googlenet_predict(in) %#codegen

   % A persistent object mynet is used to load the series network object.
   % At the first call to this function, the persistent object is constructed and
   % setup. When the function is called subsequent times, the same object is reused
   % to call predict on inputs, thus avoiding reconstructing and reloading the
   % network object.

   persistent mynet;

   if isempty(mynet)
       mynet = coder.loadDeepLearningNetwork('googlenet');
   end

   % pass in input
   out = predict(mynet,in,'MiniBatchSize',2);
   ```
2. Create a code generation configuration object for MEX or for a static or dynamically linked library. To specify code generation parameters for MKL-DNN, set the `DeepLearningConfig` property to a `coder.MklDNNConfig` object that you create with `coder.DeepLearningConfig`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

3. Run the `codegen` command. Use the `-config` option to specify the configuration object. Use the `-args` option to specify the input type. The input size corresponds to the input layer size of the GoogLeNet network with 16 different images or observations.

```
codegen -config cfg googlenet_predict -args {ones(224,224,3,16)} -report
```

Note You can specify half-precision inputs for code generation. However, the code generator type casts the inputs to single-precision. The Deep Learning Toolbox uses single-precision, floating-point arithmetic for all computations in MATLAB.

Generated Code

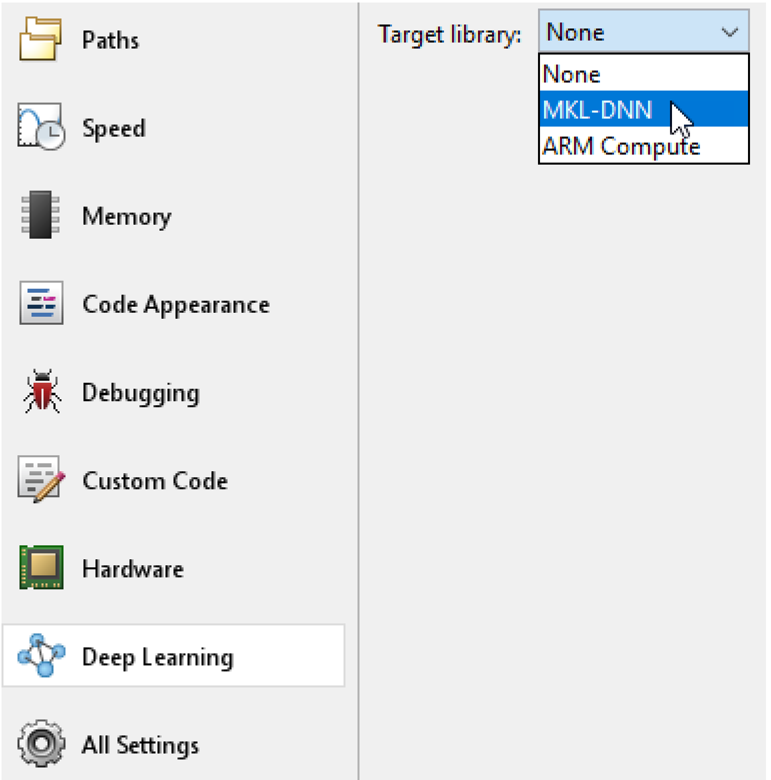
The network is generated as a C++ class containing an array of layer classes. The `setup()` method of the class sets up handles and allocates memory for each layer of the network object. The `predict()` method invokes prediction for each of the layers in the network. The code generator produces the function `googlenet_predict()` in `googlenet_predict.cpp` that corresponds to the MATLAB entry-point function. This function constructs the static object for the network and invokes the setup and predict methods.

Binary files are exported for layers with parameters such as fully connected and convolution layers in the network. For example, files `cnn_googlenet_conv*_w` and `cnn_googlenet_conv*_b` correspond to weights and bias parameters for the convolution layers in the network.

Code Generation by Using the MATLAB Coder App

1. Follow the usual steps for specifying the entry-point function and specifying input types. See “Generate C Code by Using the MATLAB Coder App”.
2. In the **Generate Code** step:

   - Set **Language** to **C++**.
   - Click **More Settings**. In the **Deep Learning** pane, set **Target library** to `MKL-DNN`.



3 Generate code.

See Also

`codegen` | `coder.DeepLearningConfig` | `coder.MklDNNConfig` | `coder.loadDeepLearningNetwork`

More About

- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” on page 38-42
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7
- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Deep Learning Networks by Using cuDNN” (GPU Coder)
- “Code Generation for Deep Learning Networks by Using TensorRT” (GPU Coder)
- “Generate C Code by Using the MATLAB Coder App”

Code Generation for Deep Learning Networks with ARM Compute Library

With MATLAB Coder, you can generate code for prediction from an already trained convolutional neural network (CNN), targeting an embedded platform that uses an ARM processor that supports the NEON extension. The code generator takes advantage of the ARM Compute Library for computer vision and machine learning. The generated code implements a CNN that has the architecture, layers, and parameters specified in the input `SeriesNetwork` or `DAGNetwork` network object.

Generate code by using one of these methods:

- codegen on page 38-32
- MATLAB Coder app on page 38-35

Requirements

- MATLAB Coder Interface for Deep Learning Libraries. To install the support package, select it from the MATLAB **Add-Ons** menu.
- ARM Compute Library for computer vision and machine learning must be installed on the target hardware.
- Deep Learning Toolbox.
- Environment variables for the compilers and libraries.

Note The ARM Compute library version that the examples in this help topic uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see "Prerequisites for Deep Learning with MATLAB Coder" on page 38-2.

Code Generation by Using codegen

To generate code for deep learning on an ARM target by using `codegen`:

- Write an entry-point function that loads the pretrained CNN and calls `predict`. For example:

```
function out = squeezenet_predict(in)
%#codegen

persistent net;
opencv_linkflags = '`pkg-config --cflags --libs opencv`';
coder.updateBuildInfo('addLinkFlags',opencv_linkflags);
if isempty(net)
    net = coder.loadDeepLearningNetwork('squeezenet', 'squeezenet');
end

out = net.predict(in);
end
```

- If your target hardware is Raspberry Pi, you can take advantage of the MATLAB Support Package for Raspberry Pi Hardware. With the support package, `codegen` moves the generated code to the Raspberry Pi and builds the executable program on the Raspberry Pi. When you generate code for

a target that does not have a hardware support package, you must run commands to move the generated files and build the executable program.

- MEX generation is not supported for code generation for deep learning on ARM targets.
- For ARM, for inputs to `predict` with multiple images or observations (`N > 1`), a `MiniBatchSize` of greater than 1 is not supported. Specify a `MiniBatchSize` of 1.

Code Generation for Deep Learning on a Raspberry Pi

When you have the MATLAB Support Package for Raspberry Pi Hardware, to generate code for deep learning on a Raspberry Pi:

1. To connect to the Raspberry Pi, use `raspi`. For example:

```
r = raspi('raspiname','username','password');
```

2. Create a code generation configuration object for a library or executable by using `coder.config`. Set the `TargetLang` property to `'C++'`.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

3. Create a deep learning configuration object by using `coder.DeepLearningConfig`. Set the `ArmComputeVersion` and `ArmArchitecture` properties. Set the `DeepLearningConfig` property of the code generation configuration object to the `coder.ARMNEONConfig` object. For example:

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
cfg.DeepLearningConfig = dlcfg;
```

4. To configure code generation hardware settings for the Raspberry Pi, create a `coder.Hardware` object, by using `coder.hardware`. Set the `Hardware` property of the code generation configuration object to the `coder.Hardware` object .

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

5. If you are generating an executable program, provide a C++ main program. For example:

```
cfg.CustomSource = 'main.cpp';
```

6. To generate code, use `codegen`. Specify the code generation configuration object by using the `-config` option. For example:

```
codegen -config cfg squeezenet_predict -args {ones(227, 227, 3,'single')} -report
```

Note You can specify half-precision inputs for code generation. However, the code generator type casts the inputs to single-precision. The Deep Learning Toolbox uses single-precision, floating-point arithmetic for all computations in MATLAB.

Code Generation When You Do Not Have a Hardware Support Package

To generate code for deep learning when you do not have a hardware support package for the target:

1. Generate code on a Linux host only.
2. Create a configuration object for a library. For example:

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

Do not use a configuration object for an executable program.

3. Configure code generation to generate C++ code and to generate source code only.

```
cfg.GenCodeOnly = true;
cfg.TargetLang = 'C++';
```

4. To specify code generation with the ARM Compute Library, create a `coder.ARMNEONConfig` object by using `coder.DeepLearningConfig`. Set the `ArmComputeVersion` and `ArmArchitecture` properties. Set the `DeepLearningConfig` property of the code generation configuration object to the `coder.ARMNEONConfig` object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
cfg.DeepLearningConfig = dlcfg;
```

5. To configure code generation parameters that are specific to the target hardware, set the `ProdHWDeviceType` property of the `HardwareImplementation` object.

   - For the ARMv7 architecture, use `'ARM Compatible->ARM Cortex'`.
   - for the ARMv8 architecture, use `'ARM Compatible->ARM 64-bit (LP64)'`.

   For example:

```
cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible->ARM 64-bit (LP64)';
```

6. To generate code, use `codegen`. Specify the code generation configuration object by using the `-config` option. For example:

```
codegen -config cfg squeezenet_predict -args {ones(227, 227, 3, 'single')} -d arm_compute
```

For an example, see “Code Generation for Deep Learning on ARM Targets” on page 38-56.

Generated Code

The series network is generated as a C++ class containing an array of layer classes.

```
class b_squeezenet_0
{
 public:
  int32_T batchSize;
  int32_T numLayers;
  real32_T *inputData;
  real32_T *outputData;
  MWCNNLayer *layers[68];
 private:
  MWTargetNetworkImpl *targetImpl;
 public:
  b_squeezenet_0();
  void presetup();
  void postsetup();
  void setup();
  void predict();
  void cleanup();
  real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
  ~b_squeezenet_0();
};
```

The `setup()` method of the class sets up handles and allocates memory for each layer of the network object. The `predict()` method invokes prediction for each of the layers in the network. Suppose that

you generate code for an entry-point function, `squeezenet_predict`. In the generated"for you" file, `squeezenet_predict.cpp`, the entry-point function `squeeznet_predict()` constructs a static object of *b_squeezenet_0* class type and invokes `setup` and `predict` on the network object.

```
static b_squeezenet_0 net;
static boolean_T net_not_empty;

// Function Definitions
//
// A persistent object net is used to load the DAGNetwork object.
//  At the first call to this function, the persistent object is constructed and
//  set up. When the function is called subsequent times, the same object is reused
//  to call predict on inputs, avoiding reconstructing and reloading the
//  network object.
// Arguments    : const real32_T in[154587]
//                real32_T out[1000]
// Return Type  : void
//
void squeezenet_predict(const real32_T in[154587], real32_T out[1000])
{
  //  Copyright 2018 The MathWorks, Inc.
  if (!net_not_empty) {
    DeepLearningNetwork_setup(&net);
    net_not_empty = true;
  }

  DeepLearningNetwork_predict(&net, in, out);
}
```

Binary files are exported for layers that have parameters, such as fully connected and convolution layers in the network. For example, the files with names having the pattern `cnn_squeezenet_*_w` and `cnn_squeezenet_*_b` correspond to weights and bias parameters for the convolution layers in the network.

```
cnn_squeezenet_conv10_b
cnn_squeezenet_conv10_w
cnn_squeezenet_conv1_b
cnn_squeezenet_conv1_w
cnn_squeezenet_fire2-expand1x1_b
cnn_squeezenet_fire2-expand1x1_w
cnn_squeezenet_fire2-expand3x3_b
cnn_squeezenet_fire2-expand3x3_w
cnn_squeezenet_fire2-squeeze1x1_b
cnn_squeezenet_fire2-squeeze1x1_w
...
```

Code Generation for Quantized Deep Learning Networks

See “Code Generation for Quantized Deep Learning Networks” on page 38-40.

Code Generation by Using the MATLAB Coder App

1. Complete the **Select Source Files** and **Define Input Types** steps.
2. Go to the **Generate Code** step. (Skip the **Check for Run-Time Issues** step because MEX generation is not supported for code generation with the ARM Compute Library.)
3. Set **Language** to **C++**.
4. Specify the target ARM hardware.

   If your target hardware is Raspberry Pi and you installed the MATLAB Support Package for Raspberry Pi Hardware:

   - For **Hardware Board**, select `Raspberry Pi`.

- To access the Raspberry Pi settings, click **More Settings**. Then, click **Hardware**. Specify the **Device Address**, **Username**, **Password**, and **Build directory**.

When you do not have a support package for your ARM target:

- Make sure that **Build type** is `Static Library` or `Dynamic Library` and select the **Generate code only** check box.
- For **Hardware Board**, select `None - Select device below`.
- For **Device vendor**, select `ARM Compatible`.
- For the **Device type**:
  - For the ARMv7 architecture, select `ARM Cortex`.
  - For the ARMv8 architecture, select `ARM 64-bit (LP64)`.

Note If you generate code for deep learning on an ARM target, and do not use a hardware support package, generate code on a Linux host only.

---

5. In the **Deep Learning** pane, set **Target library** to `ARM Compute`. Specify **ARM Compute Library version** and **ARM Compute Architecture**.
6. Generate the code.

See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.loadDeepLearningNetwork`

More About

- “Deep Learning Prediction with ARM Compute Using codegen” on page 38-51
- “Code Generation for Deep Learning on ARM Targets” on page 38-56
- “Code Generation for Quantized Deep Learning Networks” on page 38-40
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7
- “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
- “Code Generation for Deep Learning Networks by Using cuDNN” (GPU Coder)
- “Code Generation for Deep Learning Networks by Using TensorRT” (GPU Coder)

Cross-Compile Deep Learning Code That Uses ARM Compute Library

On the computer that hosts your MATLAB session, you can generate deep learning source code and compile it to create a library or an executable that runs on a target ARM hardware device. The compilation of source code on one platform to create binary code for another platform is known as cross-compilation. This workflow is supported only for the Linux host platform and target devices that have armv7 (32-bit) or armv8 (64-bit) ARM architecture.

Use this workflow to deploy deep learning code on ARM devices that do not have hardware support packages.

Note The ARM Compute library version that the examples in this help topic uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

Prerequisites

These are the prerequisites specific to the cross-compilation workflow. For the general prerequisites, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

- The target device must have armv7 (32-bit) or armv8 (64-bit) ARM architecture. To verify the architecture of your device run this command in the terminal of the device:

  ```
  arch
  ```
- You must have the Linaro AArch32 or AArch64 toolchain installed on the host computer.

  - For armv7 target, install the GNU/GCC `g++-arm-linux-gnueabihf` toolchain on the host.
  - For armv8 target, install the GNU/GCC `g++-aarch64-linux-gnu` toolchain on the host.

  For example, to install the Linaro AArch64 toolchain on the host, run this command in the terminal:

  ```
  sudo apt-get install g++-aarch64-linux-gnu
  ```
- At the MATLAB command line, set the environment variable `LINARO_TOOLCHAIN_AARCH32` or `LINARO_TOOLCHAIN_AARCH64` for the path of the toolchain binaries. You must set the path once per MATLAB session.

  Suppose that the toolchain is installed at the location `/usr/bin` in the host.

  - For armv7 target, run this command:

    ```
    setenv('LINARO_TOOLCHAIN_AARCH32', '/usr/bin')
    ```
  - For armv8 target, run this command:

    ```
    setenv('LINARO_TOOLCHAIN_AARCH64', '/usr/bin')
    ```
- Cross-compile the ARM Compute library on the host:

  - Clone the Git™ repository for ARM Compute library and check out the version you need. For example, to check out v19.05, run these commands in the host terminal:

```
git clone https://github.com/Arm-software/ComputeLibrary.git
cd ComputeLibrary
git tag -l
git checkout v19.05
```

- Install scons on the host. For example, run this commands in the host terminal:

  ```
  sudo apt-get install scons
  ```

- Use scons to cross-compile the ARM Compute library on host. For example, to build the library to run on armv8 architecture, run this command in the host terminal:

  ```
  scons Werror=0 -j8 debug=0 neon=1 opencl=0 os=linux arch=arm64-v8a openmp=1 cppthreads=1 ex
  ```

- At the MATLAB command line, set the environment variable `ARM_COMPUTELIB` for the path of the ARM Compute library. You must set the path once per MATLAB session.

  Suppose that the ARM Compute library is installed at the location `/home/$(USER)/Desktop/ComputeLibrary`. Run this command at the MATLAB command line:

  ```
  setenv('ARM_COMPUTELIB','/home/$(USER)/Desktop/ComputeLibrary')
  ```

Generate and Deploy Deep Learning Code

There are two possible workflows for cross-compiling deep learning code on your host computer and then deploying the code on target ARM hardware. Here is a summary of the two workflows. For an example that demonstrates both workflows, see “Cross Compile Deep Learning Code for ARM Neon Targets” on page 38-101.

- On the host computer, you generate a static or dynamic library for deep learning code. Follow these steps:

  - On the host, use the `codegen` command to generate and build deep learning code to create a static or dynamic library.
  - Copy the generated library, the ARM Compute library files, the makefile, and other supporting files to the target hardware.
  - Compile the copied makefile on the target to create an executable.
  - Run the generated executable on the target hardware.
- On the host computer, you generate an executable for deep learning code. Follow these steps:

  - On the host, use the `codegen` command to generate and build deep learning code to create an executable.
  - Copy the generated executable, the ARM Compute library files, and other supporting files to the target hardware.
  - Run the executable on the target hardware.

See Also

`codegen`

More About

- “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2
- “Networks and Layers Supported for Code Generation” on page 38-8

- “Cross Compile Deep Learning Code for ARM Neon Targets” on page 38-101

Code Generation for Quantized Deep Learning Networks

Deep learning uses neural network architectures that contain many processing layers, including convolutional layers. Deep learning models typically work on large sets of labeled data. Performing inference on these models is computationally intensive, consuming significant amount of memory. Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. Deep Neural networks trained in MATLAB use single-precision floating point data types. Even networks that are small in size require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and smaller memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

You can use Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. Then, you can use MATLAB Coder to generate optimized code for the quantized network. The generated code takes advantage of ARM processor SIMD by using the ARM Compute library. The generated code can be integrated into your project as source code, static or dynamic libraries, or executables that you can deploy to a variety of ARM CPU platforms such as Raspberry Pi.

Supported Layers and Classes

You can generate C++ code for these convolution layers that uses the ARM Compute Library and performs inference computations in 8-bit integers:

- 2-D convolution layer (`convolution2dLayer`)
- 2-D grouped convolution layer (`groupedConvolution2dLayer`). The value of the `NumGroups` input argument must be equal to `2`.

C++ code generation for quantized deep learning networks supports `DAGNetwork` and `SeriesNetwork` objects.

Generating Code

To generate code that performs inference computations in 8-bit integers, in your `coder.ARMNEONConfig` object `dlcfg`, set these additional properties:

```
dlcfg.CalibrationResultFile = 'dlquantizerObjectMatFile';
dlcfg.DataType = 'int8';
```

Alternatively, in the MATLAB Coder app, on the **Deep Learning** tab, set **Target library** to `ARM Compute`. Then set the **Data type** and **Calibration result file path** parameters.

Here `'dlquantizerObjectMatFile'` is the name of the MAT-file that `dlquantizer` generates for specific calibration data. For the purpose of calibration, set the `ExecutionEnvironment` property of the `dlquantizer` object to `'CPU'`.

Otherwise, follow the steps described in “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32.

For an example, see “Code Generation for Quantized Deep Learning Network on Raspberry Pi” on page 38-107.

See Also

Apps
Deep Network Quantizer

Functions
`calibrate` | `codegen` | `coder.loadDeepLearningNetwork` | `dlquantizationOptions` | `dlquantizer` | `validate`

Objects
`coder.ARMNEONConfig`

More About

- “Quantization of Deep Neural Networks” (Deep Learning Toolbox)
- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Quantized Deep Learning Network on Raspberry Pi” on page 38-107

Deep Learning Code Generation on Intel Targets for Different Batch Sizes

This example shows how to use the `codegen` command to generate code for an image classification application that uses deep learning on Intel® processors. The generated code uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). This example consists of two parts:

- The first part shows how to generate a MEX function that accepts a batch of images as input.
- The second part shows how to generate an executable that accepts a batch of images as input.

Prerequisites

- Intel processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Environment variables for the compilers and libraries. For information on the supported versions of compilers, see Supported Compilers. For setting up the environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

This example is supported on Linux® , Windows® and Mac® platforms and not supported for MATLAB Online.

Download input video File

Download a sample video file.

```
if ~exist('./object_class.avi', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpucoder/media/object_class.avi.zip';
    websave('object_class.avi.zip',url);
    unzip('object_class.avi.zip');
end
```

Define the `resnet_predict` Function

This example uses the DAG network ResNet-50 to show image classification on Intel desktops. A pretrained ResNet-50 model for MATLAB is available as part of the support package Deep Learning Toolbox Model for ResNet-50 Network.

The `resnet_predict` function loads the ResNet-50 network into a persistent network object and then performs prediction on the input. Subsequent calls to the function reuse the persistent network object.

```
type resnet_predict
```

```
% Copyright 2020 The MathWorks, Inc.

function out = resnet_predict(in)
%#codegen

% A persistent object mynet is used to load the series network object.
% At the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is reused
% to call predict on inputs, avoiding reconstructing and reloading the
% network object.
```

```
persistent mynet;

if isempty(mynet)
    % Call the function resnet50 that returns a DAG network
    % for ResNet-50 model.
    mynet = coder.loadDeepLearningNetwork('resnet50','resnet');
end

% pass in input
out = mynet.predict(in);
```

Generate MEX for `resnet_predict`

To generate a MEX function for the `resnet_predict` function, use `codegen` with a deep learning configuration object for the MKL-DNN library. Attach the deep learning configuration object to the MEX code generation configuration object that you pass to `codegen`. Run the `codegen` command and specify the input as a 4D matrix of size [224,224,3,|batchSize|]. This value corresponds to the input layer size of the ResNet-50 network.

```
batchSize = 5;
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg resnet_predict -args {ones(224,224,3,batchSize,'single')} -report
```

```
Code generation successful: To view the report, open('codegen\mex\resnet_predict\html\report.mlda
```

Perform Prediction on a Batch of Images

Presuming the Object_class.avi video file is already downloaded. Create the videoReader object and read five frames using videoReader read function.Since batchSize is set to 5 read 5 images .Resize the batch of input images to size needed by resnet50 size expected by ResNet50 network.

```
videoReader = VideoReader('Object_class.avi');
imBatch = read(videoReader,[1 5]);
imBatch = imresize(imBatch, [224,224]);
```

Call the generated `resnet_predict_mex` function which outputs classification results for the inputs that you provide.

```
predict_scores = resnet_predict_mex(single(imBatch));
```

Get top 5 probability scores and their labels for each image in the batch.

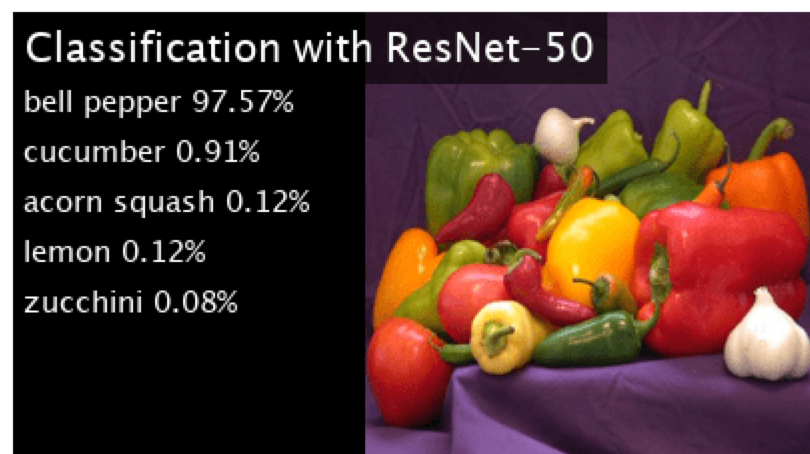
```
[val,indx] = sort(transpose(predict_scores), 'descend');
scores = val(1:5,:)*100;
net = resnet50;
classnames = net.Layers(end).ClassNames;
for i = 1:batchSize
    labels = classnames(indx(1:5,i));
    disp(['Top 5 predictions on image, ', num2str(i)]);
    for j=1:5
        disp([labels{j},' ',num2str(scores(j,i), '%2.2f'),'%'])
    end
end
```

For predictions on the first image, map the top five prediction scores to words in the `synset` dictionary.

```
fid = fopen('synsetWords.txt');
synsetOut = textscan(fid,'%s', 'delimiter', '\n');
synsetOut = synsetOut{1};
fclose(fid);
[val,indx] = sort(transpose(predict_scores), 'descend');
scores = val(1:5,1)*100;
top5labels = synsetOut(indx(1:5,1));
```

Display the top five classification labels on the image.

```
outputImage = zeros(224,400,3, 'uint8');
for k = 1:3
    outputImage(:,177:end,k) = imBatch(:,:,k,1);
end

scol = 1;
srow = 1;
outputImage = insertText(outputImage, [scol, srow], 'Classification with ResNet-50', 'TextColo
srow = srow + 30;
for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], [top5labels{k},' ',num2str(scores(k),
    srow = srow + 25;
end

imshow(outputImage);
```



Clear the persistent network object from memory.

```
clear mex;
```

Define the resnet_predict_exe Entry-Point Function

To generate an executable from MATLAB code, define a new entry-point function `resnet_predict_exe`. This function is similar to the previous entry-point function `resent_predict` but, in addition, includes code for preprocessing and postprocessing. The API that `resnet_predict_exe` uses is platform independent. This function accepts a video and the batch size as input arguments. These arguments are compile-time constants.

```
type resnet_predict_exe
```

```
% Copyright 2020 The MathWorks, Inc.

function resnet_predict_exe(inputVideo,batchSize)
%#codegen

    % A persistent object mynet is used to load the series network object.
    % At the first call to this function, the persistent object is constructed and
    % setup. When the function is called subsequent times, the same object is reused
    % to call predict on inputs, avoiding reconstructing and reloading the
    % network object.
    persistent mynet;

    if isempty(mynet)
        % Call the function resnet50 that returns a DAG network
        % for ResNet-50 model.
        mynet = coder.loadDeepLearningNetwork('resnet50','resnet');
    end

    % Create video reader and video player objects %
    videoReader = VideoReader(inputVideo);
    depVideoPlayer = vision.DeployableVideoPlayer;


    % Read the classification label names %
    synsetOut = readImageClassLabels('synsetWords.txt');

    i=1;
    % Read frames until end of video file %
    while ~(i+batchSize > (videoReader.NumFrames+1))
        % Read and resize batch of frames as specified by input argument%
        reSizedImagesBatch = readImageInputBatch(videoReader,batchSize,i);

        % run predict on resized input images %
        predict_scores = mynet.predict(reSizedImagesBatch);


        % overlay the prediction scores on images and display %
        overlayResultsOnImages(predict_scores,synsetOut,reSizedImagesBatch,batchSize,depVideoPlay

        i = i+ batchSize;
    end
    release(depVideoPlayer);
end

function synsetOut = readImageClassLabels(classLabelsFile)
% Read the classification label names from the file
%
```

```
% Inputs :
% classLabelsFile - supplied by user
%
% Outputs :
% synsetOut       - cell array filled with 1000 image class labels

    synsetOut = cell(1000,1);
    fid = fopen(classLabelsFile);
    for i = 1:1000
        synsetOut{i} = fgetl(fid);
    end
    fclose(fid);
end

function reSizedImagesBatch = readImageInputBatch(videoReader,batchSize,i)
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
% videoReader - Object used for reading the images from video file
% batchSize   - Number of images in batch to process. Supplied by user
% i           - index to track frames read from video file
%
% Outputs :
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize

    img = read(videoReader,[i (i+batchSize-1)]);
    reSizedImagesBatch = coder.nullcopy(ones(224,224,3,batchSize,'like',img));
    resizeTo  = coder.const([224,224]);
    reSizedImagesBatch(:,:,:,:) = imresize(img,resizeTo);
end


function overlayResultsOnImages(predict_scores,synsetOut,reSizedImagesBatch,batchSize,depVideoPla
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
% predict_scores  - classification results for given network
% synsetOut       - cell array filled with 1000 image class labels
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize
% batchSize       - Number of images in batch to process. Supplied by user
% depVideoPlayer  - Object for displaying results
%
% Outputs :
% Predicted results overlayed on input images

    % sort the predicted scores  %
    [val,indx] = sort(transpose(predict_scores), 'descend');

    for j = 1:batchSize
        scores = val(1:5,j)*100;
        outputImage = zeros(224,400,3, 'uint8');
        for k = 1:3
            outputImage(:,177:end,k) = reSizedImagesBatch(:,:,k,j);
        end

        % Overlay the results on image %
        scol = 1;
        srow = 1;
```

```
        outputImage = insertText(outputImage, [scol, srow], 'Classification with ResNet-50', 'Te
        srow = srow + 30;
        for k = 1:5
            scoreStr = sprintf('%2.2f',scores(k));
            outputImage = insertText(outputImage, [scol, srow], [synsetOut{indx(k,j)},' ',scoreS
            srow = srow + 25;
        end

        depVideoPlayer(outputImage);
    end
end
```

Structure of the `resnet_predict_exe` Function

The function `resnet_predict_exe` contains four subsections that perform these actions:

- Read the classification labels from supplied input text file
- Read the input batch of images and resize them as needed by the network
- Run inference on input image batch
- Overlay the results on the images

For more information each of these steps, see the subsequent sections.

The `readImageClassLabels` Function

This function accepts the `synsetWords.txt` file as an input argument. It reads the classification labels and populates a cell array.

```
function synsetOut = readImageClassLabels(classLabelsFile)
% Read the classification label names from the file
%
% Inputs :
% classLabelsFile - supplied by user
%
% Outputs :
% synsetOut       - cell array filled with 1000 image class labels

    synsetOut = cell(1000,1);
    fid = fopen(classLabelsFile);
    for i = 1:1000
        synsetOut{i} = fgetl(fid);
    end
    fclose(fid);
end
```

The `readImageInputBatch` Function

This function reads and resizes the images from the video input file that is passed to the function as an input argument. It reads the specified input images and resizes them to 224x224x3 which is the size the resnet50 network expects.

```
function reSizedImagesBatch = readImageInputBatch(videoReader,batchSize,i)
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
```

```
% videoReader - Object used for reading the images from video file
% batchSize   - Number of images in batch to process. Supplied by user
% i           - index to track frames read from video file
%
% Outputs :
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize

    img = read(videoReader,[i (i+batchSize-1)]);
    reSizedImagesBatch = coder.nullcopy(ones(224,224,3,batchSize,'like',img));
    resizeTo  = coder.const([224,224]);
    reSizedImagesBatch(:,:,:,:) = imresize(img,resizeTo);
end
```

The mynet.predict Function

This function accepts the resized batch of images as input and returns the prediction results.

```
% run predict on resized input images %
predict_scores = mynet.predict(reSizedImagesBatch);
```

The overlayResultsOnImages Function

This function accepts the prediction results and sorts them in descending order. It overlays these results on the input images and displays them.

```
function overlayResultsOnImages(predict_scores,synsetOut,reSizedImagesBatch,batchSize,depV
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
% predict_scores  - classification results for given network
% synsetOut       - cell array filled with 1000 image class labels
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize
% batchSize       - Number of images in batch to process. Supplied by user
% depVideoPlayer  - Object for displaying results
%
% Outputs :
% Predicted results overlayed on input images

    % sort the predicted scores  %
    [val,indx] = sort(transpose(predict_scores), 'descend');

    for j = 1:batchSize
        scores = val(1:5,j)*100;
        outputImage = zeros(224,400,3, 'uint8');
        for k = 1:3
            outputImage(:,177:end,k) = reSizedImagesBatch(:,:,k,j);
        end

        % Overlay the results on image %
        scol = 1;
        srow = 1;
        outputImage = insertText(outputImage, [scol, srow], 'Classification with ResNet-50
        srow = srow + 30;
        for k = 1:5
            scoreStr = sprintf('%2.2f',scores(k));
            outputImage = insertText(outputImage, [scol, srow], [synsetOut{indx(k,j)},' '
            srow = srow + 25;
        end
```

```
            depVideoPlayer(outputImage);
        end
    end
```

Build and Run Executable

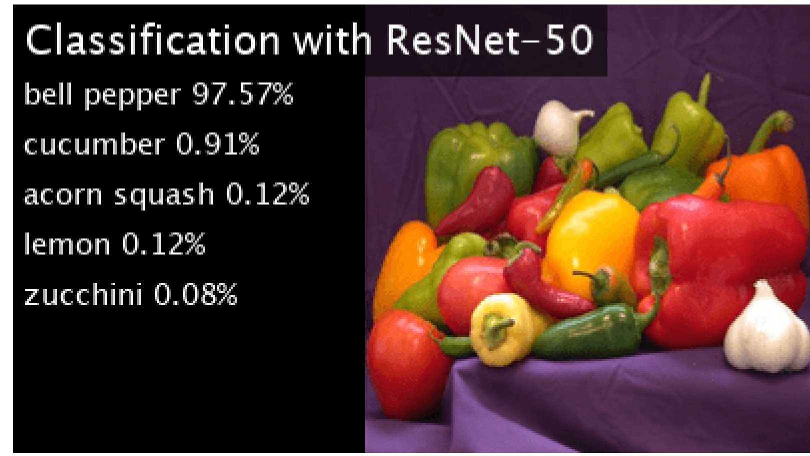
Create a code configuration object for generating an executable. Attach a deep learning configuration object to it. Set the `batchSize` and `inputVideoFile` variables.

If you do not intend to create a custom C++ main function and use the generated example C++ main instead, set the `GenerateExampleMain` parameter to `'GenerateCodeAndCompile'`. Also, disable cfg.EnableOpenMP to make sure there are no openmp library dependencies when you run your executable from the desktop terminal.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
batchSize = 5;
inputVideoFile = 'object_class.avi';
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
cfg.EnableOpenMP = 0;
```

Run the `codegen` command to build the executable. Run the generated executable resnet_predict_exe either at the MATLAB command line or at the desktop terminal.

```
codegen -config cfg resnet_predict_exe -args {coder.Constant(inputVideoFile), coder.Consta
system('./resnet_predict_exe')
```



See Also

`codegen` | `coder.DeepLearningConfig` | `coder.MklDNNConfig` | `coder.loadDeepLearningNetwork`

More About

- “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
- “Deep Learning Prediction with ARM Compute Using codegen” on page 38-51
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7

Deep Learning Prediction with ARM Compute Using codegen

This example shows how to use `codegen` to generate code for a Logo classification application that uses deep learning on ARM® processors. The logo classification application uses the `LogoNet` series network to perform logo recognition from images. The generated code takes advantage of the ARM Compute library for computer vision and machine learning.

Prerequisites

- ARM processor that supports the NEON extension
- Open Source Computer Vision Library (OpenCV) v3.1
- Environment variables for ARM Compute and OpenCV libraries
- MATLAB® Coder™ for C++ code generation
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™ for using the `SeriesNetwork` object

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

This example is supported on Linux® and Windows® platforms and not supported for MATLAB Online.

Get the Pretrained SeriesNetwork

Download the pretrained `LogoNet` network and save it as `logonet.mat`, if it does not exist. The network was developed in MATLAB® and its architecture is similar to that of AlexNet. This network can recognize 32 logos under various lighting conditions and camera angles.

```
net = getLogonet();
```

The network contains 22 layers including convolution, fully connected, and the classification output layers.

```
net.Layers
```

```
ans = 

  22×1 Layer array with layers:

     1   'imageinput'    Image Input             227×227×3 images with 'zerocenter' normalizatio
     2   'conv_1'        Convolution             96 5×5×3 convolutions with stride [1  1] and pa
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     Max Pooling             3×3 max pooling with stride [2  2] and padding 
     5   'conv_2'        Convolution             128 3×3×96 convolutions with stride [1  1] and 
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     Max Pooling             3×3 max pooling with stride [2  2] and padding 
     8   'conv_3'        Convolution             384 3×3×128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
    10   'maxpool_3'     Max Pooling             3×3 max pooling with stride [2  2] and padding 
    11   'conv_4'        Convolution             128 3×3×384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                    ReLU
    13   'maxpool_4'     Max Pooling             3×3 max pooling with stride [2  2] and padding 
```

```
    14   'fc_1'          Fully Connected         2048 fully connected layer
    15   'relu_5'        ReLU                    ReLU
    16   'dropout_1'     Dropout                 50% dropout
    17   'fc_2'          Fully Connected         2048 fully connected layer
    18   'relu_6'        ReLU                    ReLU
    19   'dropout_2'     Dropout                 50% dropout
    20   'fc_3'          Fully Connected         32 fully connected layer
    21   'softmax'       Softmax                 softmax
    22   'classoutput'   Classification Output   crossentropyex with 'adidas' and 31 other classe
```

Set Environment Variables

On the ARM target hardware, make sure that ARM_COMPUTELIB is set and that LD_LIBRARY_PATH contains the path to the ARM Compute Library folder.

See “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

logonet_predict Function

The `logonet_predict.m` entry-point function takes an image input and performs prediction on the image using the deep learning network saved in the `LogoNet` MAT-file. The function loads the network object from `LogoNet.mat` into a persistent network variable *logonet*. On subsequent calls to the function, the persistent object is reused.

```
type logonet_predict
```

```
function out = logonet_predict(in)
%#codegen

% Copyright 2017-2020 The MathWorks, Inc.

persistent logonet;

if isempty(logonet)

    logonet = coder.loadDeepLearningNetwork('LogoNet.mat','logonet');
end

out = logonet.predict(in);

end
```

Set Up a Code Generation Configuration Object for a Static Library

When you generate code targeting an ARM-based device and do not use a hardware support package, create a configuration object for a library. Do not create a configuration object for an executable program.

Set up the configuration object for generation of C++ code and generation of code only.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenCodeOnly = true;
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the library version and the architecture of the target ARM processor. For example, suppose that the target board is a HiKey/Rock960 board with ARMv8 architecture and ARM Compute Library version 19.05.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv8';
```

Attach the Deep Learning Configuration Object to the Code Generation Configuration Object

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Generate Source C++ Code by Using codegen

```
codegen -config cfg logonet_predict -args {ones(227, 227, 3, 'single')} -d arm_compute
```

The code is generated in the `arm_compute` folder in the current working folder on the host computer.

Generate the Zip File Using the packNGo function

The packNGo function packages all relevant files in a compressed zip file.

```
zipFileName = 'arm_compute.zip';
bInfo = load(fullfile('arm_compute','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'fileName', zipFileName,'minimalHeaders', false, 'ignoreFileMissing',tr
```

Copy the Generated Zip File to the Target Hardware

Copy the Zip file and extract into a folder. Remove the Zip file from the target hardware.

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Run these commands to copy and extract zip file from Linux.

```
if isunix, system(['sshpass -p password scp -r '  fullfile(pwd,zipFileName) ' username@targetname
if isunix, system('sshpass -p password ssh username@targetname "if [ -d targetloc/arm_compute ];
if isunix, system(['sshpass -p password ssh username@targetname "unzip targetloc/' zipFileName '
if isunix, system(['sshpass -p password ssh username@targetname "rm -rf  targetloc' zipFileName
```

Run these commands to copy and extract zip file from Windows.

```
if ispc, system(['pscp.exe -pw password -r '  fullfile(pwd,zipFileName) ' username@targetname:ta
if ispc, system('plink.exe -l username -pw password targetname "if [ -d targetloc/arm_compute ];
if ispc, system(['plink.exe -l username -pw password targetname "unzip targetloc/' zipFileName '
if ispc, system(['plink.exe -l username -pw password targetname "rm -rf  targetloc' zipFileName
```

Copy Example Files to the Target Hardware

Copy these supporting files from the host computer to the target hardware:

- Input image, `coderdemo_google.png`
- Makefile for generating the library, `logonet_predict_rtw.mk`
- Makefile for building the executable program, `makefile_arm_logo.mk`
- Synset dictionary, `synsetWordsLogoDet.txt`

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Perform the steps below to copy all the required files when running from Linux

```
if isunix, system('sshpass -p password scp logonet_predict_rtw.mk username@targetname:targetloc/a
if isunix, system('sshpass -p password scp coderdemo_google.png username@targetname:targetloc/arm
if isunix, system('sshpass -p password scp makefile_arm_logo.mk username@targetname:targetloc/arm
if isunix, system('sshpass -p password scp synsetWordsLogoDet.txt username@targetname:targetloc/a
```

Perform the steps below to copy all the required files when running from Windows

```
if ispc, system('pscp.exe -pw password logonet_predict_rtw.mk username@targetname:targetloc/arm_c
if ispc, system('pscp.exe -pw password coderdemo_google.png username@targetname:targetloc/arm_com
if ispc, system('pscp.exe -pw password makefile_arm_logo.mk username@targetname:targetloc/arm_com
if ispc, system('pscp.exe -pw password synsetWordsLogoDet.txt username@targetname:targetloc/arm_c
```

Build the Library on the Target Hardware

To build the library on the target hardware, execute the generated makefile on the ARM hardware.

Make sure that you set the environment variables ARM_COMPUTELIB and LD_LIBRARY_PATH on the target hardware. See “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2. The ARM_ARCH variable is used in the Makefile to pass compiler flags based on Arm Architecture. ARM_VER variable is used in the Makefile to compile the code based on Arm Compute Version. Replace the hardware credentials and paths in these commands similar to previous section.

Perform the below steps to build the library from Linux.

```
if isunix, system('sshpass -p password scp main_arm_logo.cpp username@targetname:targetloc/arm_co
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -
```

Perform the below steps to build the library from windows.

```
if ispc, system('pscp.exe -pw password main_arm_logo.cpp username@targetname:targetloc/arm_comput
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -
```

Create Executable from the Library on the Target Hardware

Build the library with the source main wrapper file to create the executable. `main_arm_logo.cpp` is the C++ main wrapper file which invokes the `logonet_predict` function.

Run the below command to create the executable from Linux.

```
if isunix, system('sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -
```

Run the below command to create the executable from Windows.

```
if ispc, system('plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -
```

Run the Executable on the Target Hardware

Run the executable from Linux using below command.

```
if isunix, system('sshpass -p password ssh username@targetname "cd targetloc/arm_compute/; ./logo
```

Run the executable from Windows using below command.

```
if ispc, system('plink.exe -l username -pw password targetname "cd targetloc/arm_compute/; ./logo
```

```
Top 5 Predictions:
-----------------------------
99.992% google
0.003% corona
0.003% singha
0.001% esso
0.000% fedex
```

See Also

cnncodegen | coder.DeepLearningConfig | coder.loadDeepLearningNetwork

More About

- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” on page 38-42
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7
- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Deep Learning on ARM Targets” on page 38-56

Code Generation for Deep Learning on ARM Targets

This example shows how to generate and deploy code for prediction on an ARM®-based device without using a hardware support package.

When you generate code for prediction using the ARM Compute Library and a hardware support package, `codegen` generates code on the host computer, copies the generated files to the target hardware, and builds the executable on the target hardware. Without a hardware support package, `codegen` generates code on the host computer. You must run commands to copy the files and build the executable program on the target hardware.

This example uses the `packNGo` function to package all relevant files into a compressed zip file. Use this example to learn how to deploy the generated code on ARM Neon targets that do not have a hardware support package by using `packNGo`.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- Open Source Computer Vision Library(Open CV)
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

This example is not supported for MATLAB Online.

squeezenet_predict Function

This example uses the DAG network SqueezeNet to show image classification with the ARM Compute Library. A pretrained SqueezeNet for MATLAB is available in the Deep Learning Toolbox. The `squeezenet_predict` function loads the SqueezeNet network into a persistent network object. On subsequent calls to the function, the persistent object is reused.

```
type squeezenet_predict
```

```
% Copyright 2018 The MathWorks, Inc.

function out = squeezenet_predict(in)
%#codegen

% A persistent object mynet is used to load the DAG network object.
% At the first call to this function, the persistent object is constructed and
% set up. When the function is called subsequent times, the same object is reused
% to call predict on inputs, avoiding reconstructing and reloading the
% network object.

persistent mynet;
```

```
if isempty(mynet)
       mynet = coder.loadDeepLearningNetwork('squeezenet','squeezenet');
end

out = mynet.predict(in);
```

Set Up a Code Generation Configuration Object for a Static Library

When you generate code targeting an ARM-based device and do not use a hardware support package, create a configuration object for a library. Do not create a configuration object for an executable program.

Set up the configuration object for generation of C++ code and generation of code only.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenCodeOnly = true;
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the library version and the architecture of the target ARM processor. For example, suppose that the target board is a HiKey/Rock960 board with ARMv8 architecture and ARM Compute Library version 19.05.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv8';
```

Attach the Deep Learning Configuration Object to the Code Generation Configuration Object

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Generate Source C++ Code by Using codegen

```
codegen -config cfg squeezenet_predict -args {ones(227, 227, 3, 'single')} -d arm_compute
```

The code is generated in the arm_compute folder in the current working folder on the host computer.

Generate the Zip File using packNGo function

The packNGo function packages all relevant files in a compressed zip file.

```
zipFileName = 'arm_compute.zip';
bInfo = load(fullfile('arm_compute','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'fileName', zipFileName,'minimalHeaders', false, 'ignoreFileMissing',tr
```

The code is generated as zip file.

Copy the Generated Zip file to the Target Hardware

Copy the Zip file and extract into folder and remove the Zip file in the hardware

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Perform the steps below to copy and extract zip file from Linux.

```
if isunix, system(['sshpass -p password scp -r '  fullfile(pwd,zipFileName) ' username@targetname
if isunix, system('sshpass -p password ssh username@targetname "if [ -d targetloc/arm_compute ];
if isunix, system(['sshpass -p password ssh username@targetname "unzip targetloc/' zipFileName '
if isunix, system(['sshpass -p password ssh username@targetname "rm -rf  targetloc' zipFileName
```

Perform the steps below to copy and extract zip file from Windows.

```
if ispc, system(['pscp.exe -pw password -r '  fullfile(pwd,zipFileName) ' username@targetname:ta
if ispc, system('plink.exe -l username -pw password targetname "if [ -d targetloc/arm_compute ];
if ispc, system(['plink.exe -l username -pw password targetname "unzip targetloc/' zipFileName '
if ispc, system(['plink.exe -l username -pw password targetname "rm -rf  targetloc' zipFileName
```

Copy Example Files to the Target Hardware

Copy these supporting files from the host computer to the target hardware:

- Input image, `coffeemug.png`
- Makefile for generating the library, `squeezenet_predict_rtw.mk`
- Makefile for building the executable program, `makefile_squeezenet_arm_generic.mk`
- Synset dictionary, `synsetWords.txt`

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Perform the steps below to copy all the required files when running from Linux

```
if isunix, system('sshpass -p password scp squeezenet_predict_rtw.mk username@targetname:targetlo
if isunix, system('sshpass -p password scp coffeemug.png username@targetname:targetloc/arm_comput
if isunix, system('sshpass -p password scp makefile_squeezenet_arm_generic.mk username@targetname
if isunix, system('sshpass -p password scp synsetWords.txt username@targetname:targetloc/arm_comp
```

Perform the steps below to copy all the required files when running from Windows

```
if ispc, system('pscp.exe -pw password squeezenet_predict_rtw.mk username@targetname:targetloc/ar
if ispc, system('pscp.exe -pw password coffeemug.png username@targetname:targetloc/arm_compute/')
if ispc, system('pscp.exe -pw password makefile_squeezenet_arm_generic.mk username@targetname:tar
if ispc, system('pscp.exe -pw password synsetWords.txt username@targetname:targetloc/arm_compute/
```

Build the Library on the Target Hardware

To build the library on the target hardware, execute the generated makefile on the ARM hardware.

Make sure that you set the environment variables ARM_COMPUTELIB and LD_LIBRARY_PATH on the target hardware. See “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

ARM_ARCH variable is used in Makefile to pass compiler flags based on Arm Architecture. ARM_VER variable is used in Makefile to compile the code based on Arm Compute Version. Replace the hardware credentials and paths in similar to above steps.

Perform the below steps to build the library from Linux.

```
if isunix, system('sshpass -p password scp main_squeezenet_arm_generic.cpp username@targetname:ta
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -
```

Perform the below steps to build the library from windows.

```
if ispc, system('pscp.exe -pw password main_squeezenet_arm_generic.cpp username@targetname:target
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -
```

Create Executable from the Library on the Target Hardware

Build the library with the source main wrapper file to create the executable. `main_squeezenet_arm_generic.cpp` is the C++ main wrapper file which invokes squeezenet_predict function to create the executable.

Run the below command to create the executable from Linux.

```
if isunix, system('sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -f
```

Run the below command to create the executable from Windows.

```
if ispc, system('plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -f
```

Run the Executable on the Target Hardware

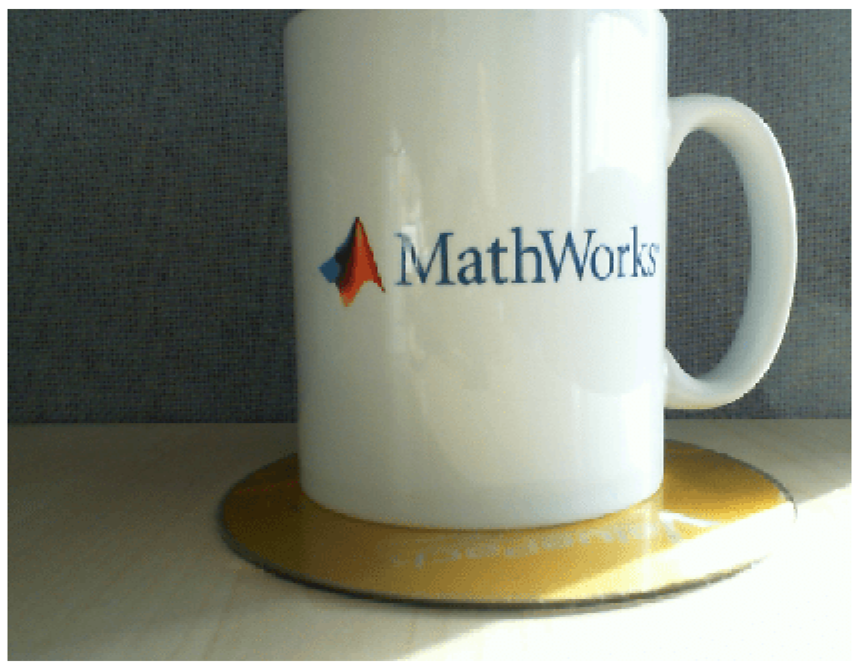
Run the executable from Linux using below command.

```
if isunix, system('sshpass -p password ssh username@targetname "cd targetloc/arm_compute/; ./sque
```

Run the executable from Windows using below command.

```
if ispc, system('plink.exe -l username -pw password targetname "cd targetloc/arm_compute/; ./sque
```

```
Top 5 Predictions:
-----------------------------
88.299% coffee mug
7.309% cup
1.098% candle
0.634% paper towel
0.591% water jug
```



See Also

coder.ARMNEONConfig | coder.DeepLearningConfig | coder.HardwareImplementation | packNGo

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32

Generate C++ Code for Object Detection Using YOLO v2 and Intel MKL-DNN

This example shows how to generate C++ code for the YOLO v2 Object detection network on an Intel® processor. The generated code uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).

For more information, see “Object Detection Using YOLO v2 Deep Learning” (Computer Vision Toolbox).

Prerequisites

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Refer MKLDNN CPU Support to know the list of processors that supports MKL-DNN library
- MATLAB® Coder™ for C++ code generation
- MATLAB Coder Interface for Deep Learning support package
- Deep Learning Toolbox™ for using the `DAGNetwork` object
- Computer Vision Toolbox™ for video I/O operations

For more information on the supported versions of the compilers and libraries, see “Third-Party Hardware and Software” on page 38-2.

This example is supported on Linux®, Windows®, and macOS platforms and not supported for MATLAB Online.

Get the Pretrained `DAGNetwork` Object

The DAG network contains 150 layers including convolution, ReLU, and batch normalization layers and the YOLO v2 transform and YOLO v2 output layers.

```
net = getYOLOv2();
```

```
Downloading pretrained detector (98 MB)...
```

Use the command `net.Layers` to see all the layers of the network.

```
net.Layers
```

Code Generation for `yolov2_detection` Function

The `yolov2_detection` function attached with the example takes an image input and runs the detector on the image using the network saved in `yolov2ResNet50VehicleExample.mat`. The function loads the network object from `yolov2ResNet50VehicleExample.mat` into a persistent variable `yolov2Obj`. Subsequent calls to the function reuse the persistent object for detection.

```
type('yolov2_detection.m')
```

```
function outImg = yolov2_detection(in)

%   Copyright 2018-2019 The MathWorks, Inc.

% A persistent object yolov2Obj is used to load the YOLOv2ObjectDetector object.
% At the first call to this function, the persistent object is constructed and
```

```
% set up. Subsequent calls to the function reuse the same object to call detection
% on inputs, thus avoiding having to reconstruct and reload the
% network object.
persistent yolov2Obj;

if isempty(yolov2Obj)
    yolov2Obj = coder.loadDeepLearningNetwork('yolov2ResNet50VehicleExample.mat');
end

% pass in input
[bboxes,~,labels] = yolov2Obj.detect(in,'Threshold',0.5);
outImg = in;

% convert categorical labels to cell array of character vectors for MATLAB
% execution
if coder.target('MATLAB')
    labels = cellstr(labels);
end

if ~(isempty(bboxes) && isempty(labels))
% Annotate detections in the image.
    outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
end
```

To generate code, create a code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a MKL-DNN deep learning configuration object. Assign this object to the `DeepLearningConfig` property of the code configuration object. Specify the input size as an argument to the `codegen` command. In this example, the input layer size of the YOLO v2 network is `[224,224,3]`.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg yolov2_detection -args {ones(224,224,3,'uint8')} -report
```

```
Code generation successful: To view the report, open('codegen\mex\yolov2_detection\html\report.m
```

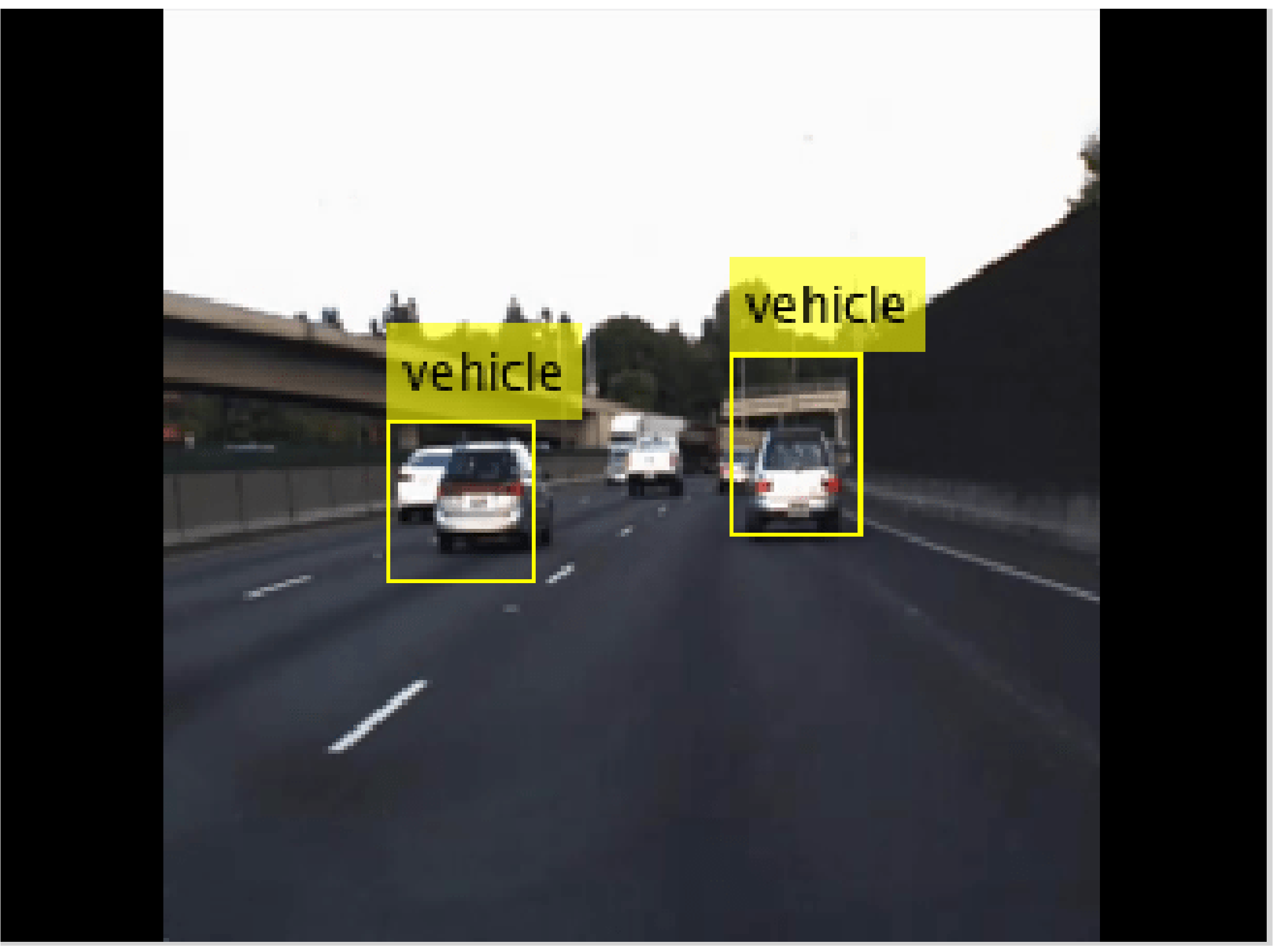
Run the Generated MEX Function on Example Input

Set up a video file reader and read the example input video `highway_lanechange.mp4`. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';
videoFreader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
depVideoPlayer = vision.DeployableVideoPlayer('Size','Custom','CustomSize',[640 480]);
```

Read the video input frame by frame and detect the vehicles in the video by using the detector.

```
cont = ~isDone(videoFreader);
while cont
    I = step(videoFreader);
    in = imresize(I,[224,224]);
    out = yolov2_detection_mex(in);
    depVideoPlayer(out);
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer); % Exit the loop if the video player 
end
```



References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517–25. Honolulu, HI: IEEE, 2017.

See Also

`coder.DeepLearningConfig` | `coder.hardware`

More About

- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” on page 38-42
- “Workflow for Deep Learning Code Generation with MATLAB Coder” on page 38-7

Code Generation and Deployment of MobileNet-v2 Network to Raspberry Pi

This example shows how to generate and deploy C++ code that uses the MobileNet-v2 pretrained network for object prediction.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- Open Source Computer Vision Library(OpenCV) v2.4 (on the target ARM hardware)
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- MATLAB Coder Interface for Deep Learning Libraries support package
- Deep Learning Toolbox™
- Deep Learning Toolbox Model for MobileNet-v2 Network support package
- Image Processing Toolbox™
- MATLAB Support Package for Raspberry Pi Hardware

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

This example is not supported for MATLAB online.

This example uses the DAG network MobileNet-v2 to perform image classification with the ARM® Compute Library. The pretrained MobileNet-v2 network for MATLAB is available in the Deep Learning Toolbox Model for MobileNet-v2 Network support package.

When you generate code that uses the ARM Compute Library and a hardware support package, `codegen` generates code on the host computer, copies the generated files to the target hardware, and builds the executable on the target hardware.

Configure Code Generation for the `mobilenet_predict` Function

The `mobilenet_predict` function calls the predict method of the MobileNet-v2 network object on an input image and returns the prediction score output. The function calls `coder.updateBuildInfo` to specify linking options for the generated makefile.

```
type mobilenet_predict
```

```
function out = mobilenet_predict(in)

persistent net;
opencv_linkflags = '`pkg-config --cflags --libs opencv`';
coder.updateBuildInfo('addLinkFlags',opencv_linkflags);
if isempty(net)
    net = coder.loadDeepLearningNetwork('mobilenetv2', 'mobilenet');
end
```

```
out = net.predict(in);

end
```

Create a C++ code generation configuration object.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Specify Use of the ARM Compute Library. The ARM Compute Library provides optimized functionality for the Raspberry Pi hardware. To generate code that uses the ARM Compute Library, create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute Library installed on your Raspberry Pi and the architecture of the Raspberry Pi. Attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
supportedVersions = dlcfg.getARMComputeSupportedVersions;
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
cfg.DeepLearningConfig = dlcfg;
```

Create a Connection to the Raspberry Pi

Use the MATLAB Support Package for Raspberry Pi Hardware function `raspi` to create a connection to the Raspberry Pi. In this code, replace:

- `raspiname` with the host name of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
r = raspi('raspiname','username','password');
```

Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for the Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify a build folder on the Raspberry Pi:

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Provide a C++ Main File

Specify the main file `main_mobilenet.cpp` in the code generation configuration object. The file calls the generated C++ code for the `mobilenet_predict` function. The file reads the input image, passes the data to the generated function calls, retrieves the predictions on the image, and prints the prediction scores to a file.

```
cfg.CustomSource = 'main_mobilenet.cpp';
```

Generate the Executable Program on the Raspberry Pi

Generate C++ code. When you use `codegen` with the MATLAB Support Package for Raspberry PI Hardware, the executable is built on the Raspberry Pi.

For code generation, you must set the “Environment Variables” on page 38-4 ARM_COMPUTELIB and LD_LIBRARY_PATH on the Raspberry Pi.

```
codegen -config cfg mobilenet_predict -args {ones(224, 224, 3,'single')} -report
```

Fetch the Generated Executable Folder

To test the generated code on the Raspberry Pi, copy the input image to the generated code folder. You can find this folder manually or by using the raspi.utils.getRemoteBuildDirectory API. This function lists the folders of the binary files that are generated by using codegen. Assuming that the binary is found in only one folder, enter:

```
applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName','mobilenet_predict');
targetDirPath = applicationDirPaths{1}.directory;
```

Copy Example Files to the Raspberry Pi

To copy files required to run the executable program, use putFile.

```
r.putFile('peppers_raspi_mobilenet.png',targetDirPath);
```

Run the Executable Program on the Raspberry Pi

Run the executable program on the Raspberry Pi from MATLAB and direct the output back to MATLAB.

```
exeName = 'mobilenet_predict.elf';
argsforexe = ' peppers_raspi_mobilenet.png '; % Provide the input image;
command = ['cd ' targetDirPath ';sudo ./' exeName argsforexe];
output = system(r,command);
```

Get the Prediction Scores for the 1000 Output Classes of the Network

```
outputfile = [targetDirPath, '/output.txt'];
r.getFile(outputfile);
```

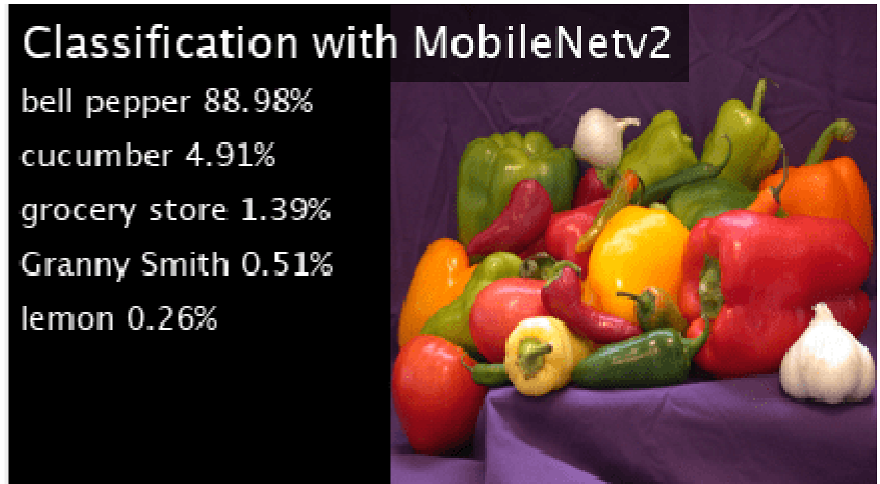
Map the Prediction Scores to Labels and Display Output

Map the top five prediction scores to the corresponding labels in the trained network, and display the output.

```
type mapPredictedScores_mobilenet
```

```
%% Map the Prediction Scores to Labels and Display Output
net = mobilenetv2;
ClassNames = net.Layers(end).ClassNames;

%% Read the classification
fid = fopen('output.txt') ;
S = textscan(fid,'%s');
fclose(fid) ;
S = S{1} ;
predict_scores = cellfun(@(x)str2double(x), S);

%% Remove NaN values that were strings
predict_scores(isnan(predict_scores))=[];
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
```

```
top5labels = ClassNames(indx(1:5));

%% Display classification labels on the image
im = imread('peppers_raspi_mobilenet.png');
im = imresize(im, [224 224]);
outputImage = zeros(224,400,3, 'uint8');
for k = 1:3
    outputImage(:,177:end,k) = im(:,:,k);
end
scol = 1;
srow = 1;
outputImage = insertText(outputImage, [scol, srow], 'Classification with MobileNetv2', 'TextColo
srow = srow + 30;
for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], [top5labels{k},' ',num2str(scores(k), '%
    srow = srow + 25;
end
imshow(outputImage);
```



See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware`

More About

- "Code Generation for Deep Learning Networks with ARM Compute Library" on page 38-32
- "Code Generation for Deep Learning on ARM Targets" on page 38-56

Code Generation for Semantic Segmentation Application on Intel CPUs That Uses U-Net

This example demonstrates code generation for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction by using the deep learning network U-Net for image segmentation.

For a similar example that demonstrates segmentation of images by using U-Net but does not use the `codegen` command, see “Semantic Segmentation of Multispectral Images Using Deep Learning” (Image Processing Toolbox).

Third-Party Prerequisites

- Xeon processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions

This example is supported on Linux®, Windows®, and macOS platforms.

This example uses the Intel MKL-DNN library that ships with MATLAB and generates a MEX function for semantic segmentation.

This example is not supported in MATLAB Online.

Overview of U-Net

U-Net [1] is a type of convolutional neural network (CNN)that is designed for semantic image segmentation. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. The combination of these two series paths forms a U-shaped graph. The network was originally trained to perform prediction for biomedical image segmentation applications. This example demonstrates the ability of the network to track changes in forest cover over time. Environmental agencies track deforestation to assess and qualify the environmental and ecological health of a region.

Deep-learning-based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One of the challenges is differentiating classes that have similar visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.

This example uses the Hamlin Beach State Park Data [2] along with a pretrained U-Net network in order to correctly classify each pixel.

The U-Net this example uses is trained to segment pixels belonging to 18 classes which includes:

```
0. Other Class/Image Border      7. Picnic Table         14. Grass
1. Road Markings                 8. Black Wood Panel     15. Sand
2. Tree                          9. White Wood Panel     16. Water (Lake)
3. Building                     10. Orange Landing Pad   17. Water (Pond)
4. Vehicle (Car, Truck, or Bus) 11. Water Buoy           18. Asphalt (Parking Lot/Walkway)
5. Person                       12. Rocks
6. Lifeguard Chair              13. Other Vegetation
```

Get Pretrained U-Net DAG Network Object

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUnet_url,pwd);
```

```
Downloading Pre-trained U-net for Hamlin Beach dataset...
This will take several minutes to download...
done.
```

```
ld = load("trainedUnet/multispectralUnet.mat");
net = ld.net;
```

The DAG network contains 58 layers including convolution, max pooling, depth concatenation, and pixel classification output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
%   analyzeNetwork(net);
```

The `segmentImageUnet` Entry-Point Function

The segmentImageUnet.m entry-point function performs semantic segmentation on the input image for each patch of a fixed size by using the multispectralUnet network contained in the `multispectralUnet.mat` file. This function loads the network object from the `multispectralUnet.mat` file into a persistent variable `mynet`. The function reuses this persistent variable in subsequent prediction calls.

```
type('segmentImageUnet.m')
```

```
%  OUT = segmentImageUnet(IM, PATCHSIZE) returns a semantically segmented
%  image, segmented using the network multispectralUnet. The segmentation
%  is performed over each patch of size PATCHSIZE.
%
% Copyright 2019-2020 The MathWorks, Inc.
function out = segmentImageUnet(im, patchSize)

%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('trainedUnet/multispectralUnet.mat');
end

[height, width, nChannel] = size(im);
patch = coder.nullcopy(zeros([patchSize, nChannel-1]));

% pad image to have dimensions as multiples of patchSize
padSize = zeros(1,2);
padSize(1) = patchSize(1) - mod(height, patchSize(1));
padSize(2) = patchSize(2) - mod(width, patchSize(2));

im_pad = padarray (im, padSize, 0, 'post');
[height_pad, width_pad, ~] = size(im_pad);

out = zeros([size(im_pad,1), size(im_pad,2)], 'uint8');
```

```
for i = 1:patchSize(1):height_pad
    for j =1:patchSize(2):width_pad
        for p = 1:nChannel-1
            patch(:,:,p) = squeeze( im_pad( i:i+patchSize(1)-1,...
                                            j:j+patchSize(2)-1,...
                                            p));
        end

        % pass in input
        segmentedLabels = activations(mynet, patch, 'Segmentation-Layer');

        % Takes the max of each channel (6 total at this point)
        [~,L] = max(segmentedLabels,[],3);
        patch_seg = uint8(L);

        % populate section of output
        out(i:i+patchSize(1)-1, j:j+patchSize(2)-1) = patch_seg;

    end
end

% Remove the padding
out = out(1:height, 1:width);
```

Prepare Data

Download the Hamlin Beach State Park data.

```
if ~exist(fullfile(pwd,'data'),'dir')
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

```
Downloading Hamlin Beach dataset...
This will take several minutes to download...
done.
```

Load and examine the data in MATLAB.

```
load(fullfile(pwd,'data','rit18_data','rit18_data.mat'));

% Examine data
whos test_data
```

```
  Name            Size                          Bytes  Class     Attributes

  test_data       7x12446x7654             1333663576  uint16
```

The image has seven channels. The RGB color channels are the fourth, fifth, and sixth image channels. The first three channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. Channel 7 is a mask that indicates the valid segmentation region.

The multispectral image data is arranged as numChannels-by-width-by-height arrays. In MATLAB, multichannel images are arranged as width-by-height-by-numChannels arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
test_data  = switchChannelsToThirdPlane(test_data);
```

Confirm data has the correct structure (channels last).

```
whos test_data
```

```
  Name                Size                      Bytes  Class     Attributes

  test_data      12446x7654x7            1333663576  uint16
```

This example uses a cropped version of the full Hamlin Beach State Park dataset that the `test_data` variable contains. Crop the height and width of `test_data` to create the variable `input_data` that this example uses.

```
test_datacropRGB = imcrop(test_data(:,:,1:3),[2600, 3000, 2000, 2000]);
test_datacropInfrared = imcrop(test_data(:,:,4:6),[2600, 3000, 2000, 2000]);
test_datacropMask = imcrop(test_data(:,:,7),[2600, 3000, 2000, 2000]);
input_data(:,:,1:3) = test_datacropRGB;
input_data(:,:,4:6) = test_datacropInfrared;
input_data(:,:,7) = test_datacropMask;
```

Examine the input_data variable.

```
whos('input_data');
```

```
  Name                Size                    Bytes  Class     Attributes

  input_data      2001x2001x7            56056014  uint16
```

Generate MEX

To generate a MEX function for the segmentImageUnet.m entry-point function, create a code configuration object `cfg` for MEX code generation. Set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create an MKL-DNN deep learning configuration object and assign it to the `DeepLearningConfig` property of `cfg`. Run the `codegen` command specifying an input size of [12446,7654,7] and a patch size of [1024,1024]. These values correspond to the size of the entire `input_data` variable. The smaller patch sizes speed up inference. To see how the patches are calculated, see the `segmentImageUnet` entry-point function.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg segmentImageUnet -args {ones(size(input_data),'uint16'),coder.Constant([1024
```

```
Code generation successful: To view the report, open('codegen\mex\segmentImageUnet\html\report.m
```

Run Generated MEX to Predict Results for input_data

The `segmentImageUnet` function accepts `input_data` and a vector containing the dimensions of the patch size as inputs. The function divides the image into patches, predicts the pixels in a particular patch, and finally combines all the patches. Because of the large size of `input_data` (12446x7654x7), it is easier to process the image in patches.

```
segmentedImage = segmentImageUnet_mex(input_data,[1024 1024]);
```

To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the test data.

```
segmentedImage = uint8(input_data(:,:,7)~=0) .* segmentedImage;
```

Remove the noise and stray pixels by using the `medfilt2` function.

```
segmentedImage = medfilt2(segmentedImage,[5,5]);
```

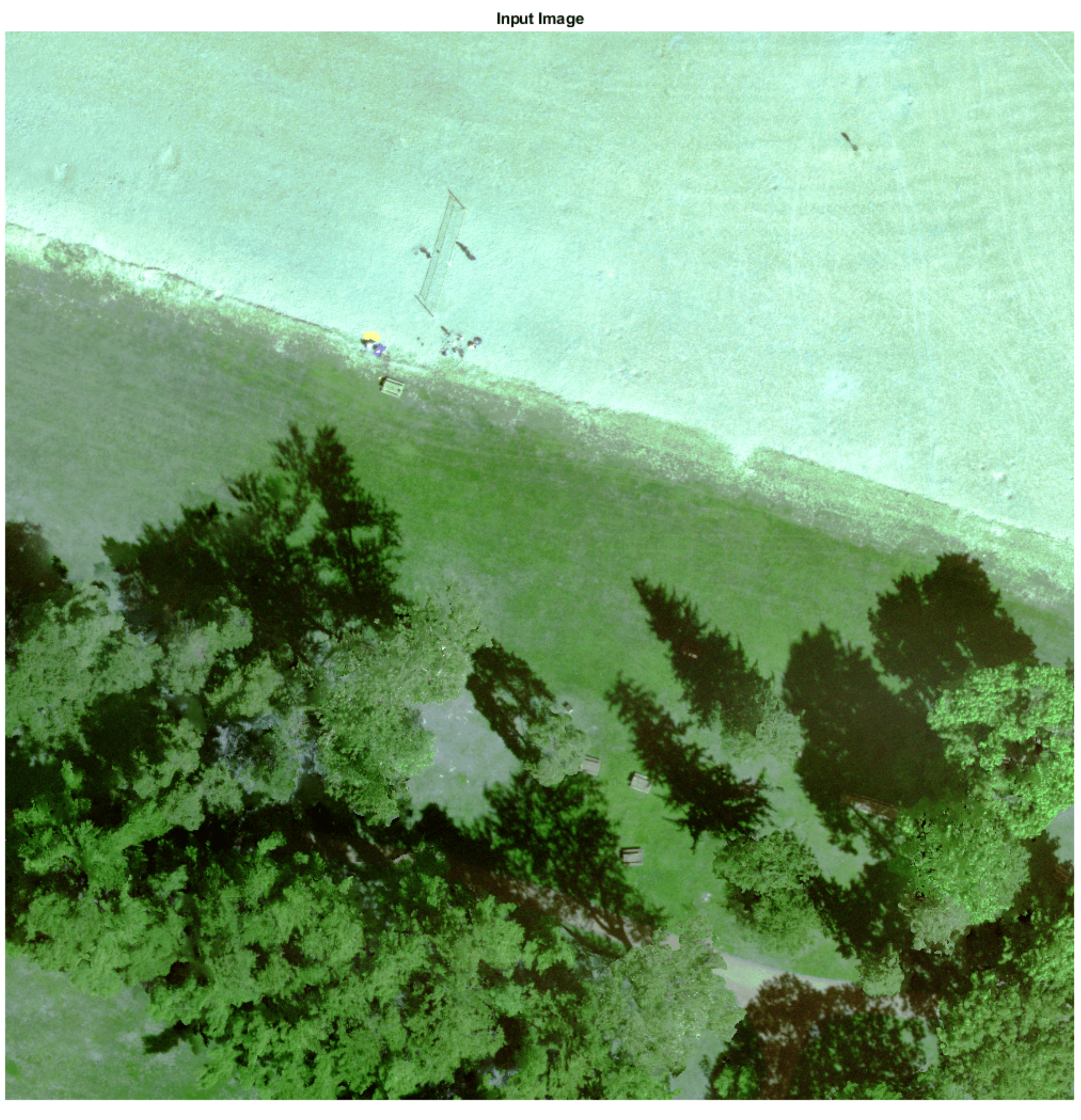
Display U-Net Segmented input_data

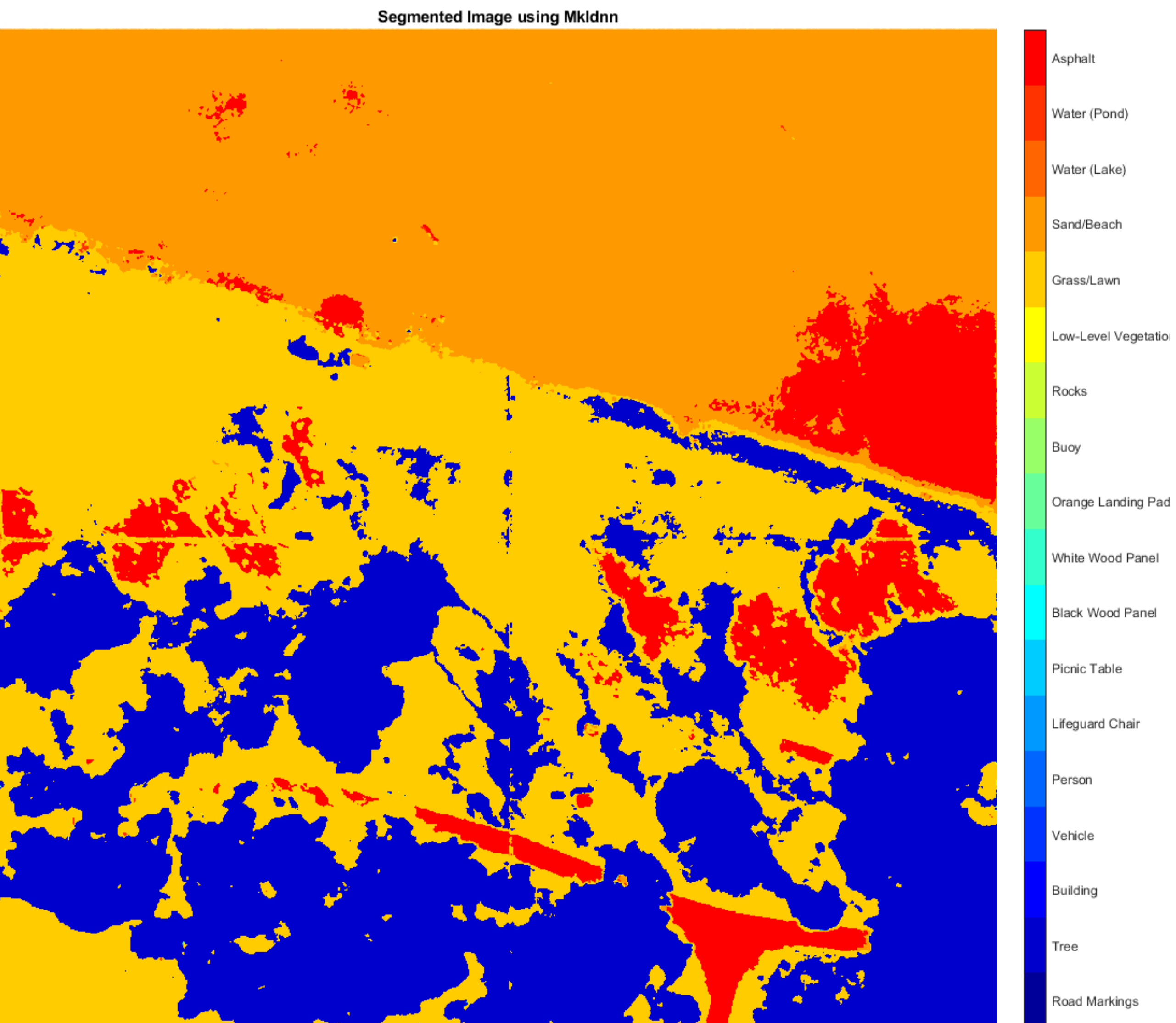
This line of code creates a vector of the class names:

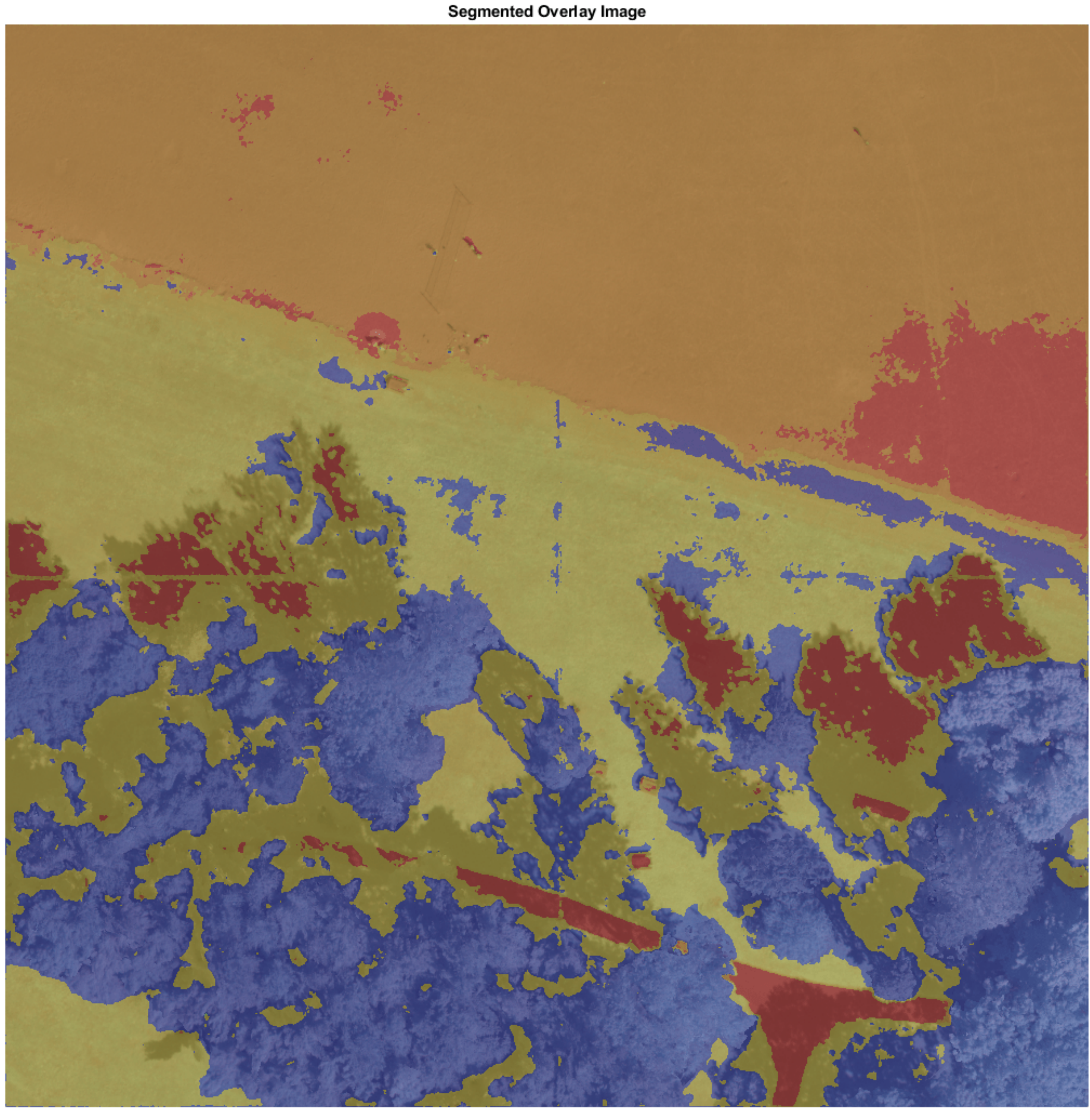
```
classNames = net.Layers(end).Classes;
```

Overlay the labels on the segmented RGB test image and add a color bar to the segmentation image.

```
% Display input data

figure(1);
imshow(histeq(input_data(:,:,1:3)));
title('Input Image');
cmap = jet(numel(classNames));
segmentedImageOut = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmentedIm

% Display segmented data

figure(2);
imshow(segmentedImageOut);
title('Segmented Image Output');
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
title('Segmented Image using Mkldnn');
segmentedImageOverlay = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segment
figure(3);
imshow(segmentedImageOverlay);
title('Segmented Overlay Image');
```


Input Image


Segmented Image using Mkldnn
Asphalt
Water (Pond)
Water (Lake)
Sand/Beach
Grass/Lawn
Low-Level Vegetatio
Rocks
Buoy
Orange Landing Pad
White Wood Panel
Black Wood Panel
Picnic Table
Lifeguard Chair
Person
Vehicle
Building
Tree
Road Markings



References

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *arXiv preprint arXiv:1505.04597,* 2015.

[2] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918, 2017.

See Also

`analyzeNetwork` (Deep Learning Toolbox) | `coder.DeepLearningConfig` | `coder.hardware`

More About

- "Deep Learning Code Generation on Intel Targets for Different Batch Sizes" on page 38-42
- "Workflow for Deep Learning Code Generation with MATLAB Coder" on page 38-7

- Semantic Segmentation of Multispectral Images Using Deep Learning

Code Generation for Semantic Segmentation Application on ARM® Neon targets That Uses U-Net

This example shows how to generate code for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a static library that performs prediction on a DAG Network object for U-Net. U-Net a deep learning network for image segmentation.

For a similar example that uses U-Net for image segmentation but does not use the `codegen` command, see “Semantic Segmentation of Multispectral Images Using Deep Learning” (Image Processing Toolbox).

Prerequisites

- ARM processor that supports the NEON extension and has a RAM of at least 3GB
- ARM Compute Library (on the target ARM hardware)
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- MATLAB Coder Interface for Deep Learning Libraries support package
- Deep Learning Toolbox™

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For information about supported versions of libraries and about environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

This example is not supported in MATLAB Online.

Overview of U-Net

U-Net [1] is a type of convolutional neural network (CNN) designed for semantic image segmentation. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. The combination of these two series paths forms a U-shaped graph. The U-Net network was originally trained to perform prediction on biomedical image segmentation applications. This example demonstrates the ability of the network to track changes in forest cover over time. Environmental agencies track deforestation to assess and qualify the environmental and ecological health of a region.

Deep learning based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One of the challenges of such computation is to differentiating classes that have similar visual characteristics, such as classifying a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.

This example uses the Hamlin Beach State Park Data [2] along with a pretrained U-Net network to correctly classify each pixel.

The U-Net that this example uses is trained to segment pixels belonging to a set of 18 classes which includes:

```
0. Other Class/Image Border      7. Picnic Table       14. Grass
1. Road Markings                 8. Black Wood Panel   15. Sand
2. Tree                          9. White Wood Panel   16. Water (Lake)
3. Building                     10. Orange Landing Pad 17. Water (Pond)
4. Vehicle (Car, Truck, or Bus) 11. Water Buoy         18. Asphalt (Parking Lot/Walkway)
5. Person                       12. Rocks
6. Lifeguard Chair              13. Other Vegetation
```

The segmentationUnetARM Entry-Point Function

The segmentationUnetARM.m entry-point function performs patchwise semantic segmentation on the input image by using the multispectralUnet network contained in the `multispectralUnet.mat` file. The function loads the network object from the `multispectralUnet.mat` file into a persistent variable `mynet` and reuses the persistent variable on subsequent prediction calls.

```
type('segmentationUnetARM.m')
```

```
%  OUT = segmentationUnetARM(IM) returns a semantically segmented
%  image, which is segmented using the network multispectralUnet. This segmentation
%  is performed on the input image patchwise on patches of size 256,256.
%
% Copyright 2019-2020 The MathWorks, Inc.
function out = segmentationUnetARM(im)

%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('trainedUnet/multispectralUnet.mat');
end

% The input data has to be padded to the size compatible
% with the network Input Size. This input_data is padded inorder to
% perform semantic segmentation on each patch of size (Network Input Size)
[height, width, nChannel] = size(im);
patch = coder.nullcopy(zeros([256, 256, nChannel-1]));
%
padSize = zeros(1,2);
padSize(1) = 256 - mod(height, 256);
padSize(2) = 256 - mod(width, 256);
%
% Pad image must have have dimensions as multiples of network input dimensions
im_pad = padarray (im, padSize, 0, 'post');
[height_pad, width_pad, ~] = size(im_pad);
%
out = zeros([size(im_pad,1), size(im_pad,2)], 'uint8');

for i = 1:256:height_pad
    for j =1:256:width_pad
        for p = 1:nChannel -1
            patch(:,:,p) = squeeze( im( i:i+255,...
                                                j:j+255,...
                                                p));
        end

        % pass in input
        segmentedLabels = activations(mynet, patch, 'Segmentation-Layer');
```

```
        % Takes the max of each channel (6 total at this point)
        [~,L] = max(segmentedLabels,[],3);
        patch_seg = uint8(L);

        % populate section of output
        out(i:i+255, j:j+255) = patch_seg;

    end
end

% Remove the padding
out = out(1:height, 1:width);
```

Get Pretrained U-Net DAG Network Object

Download the `multispectralUnet.mat` file and load the U-Net DAG network object.

```
if ~exist('trainedUnet/multispectralUnet.mat','file')
    trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
    downloadUNet(trainedUnet_url,pwd);
end

ld = load("trainedUnet/multispectralUnet.mat");
net = ld.net;
```

The DAG network contains 58 layers that include convolution, max pooling, depth concatenation, and pixel classification output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

Prepare Input Data

Download the Hamlin Beach State Park data.

```
if ~exist(fullfile(pwd,'data'),'dir')
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

Load and examine the data in MATLAB.

```
load(fullfile(pwd,'data','rit18_data','rit18_data.mat'));
```

Examine data

```
whos test_data
```

The image has seven channels. The RGB color channels are the fourth, fifth, and sixth image channels. The first three channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. Channel 7 is a mask that indicates the valid segmentation region.

The multispectral image data is arranged as numChannels-by-width-by-height arrays. In MATLAB, multichannel images are arranged as width-by-height-by-numChannels arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
test_data = switchChannelsToThirdPlane(test_data);
```

Confirm data has the correct structure (channels last).

```
whos test_data
```

This example uses a cropped version of the full Hamlin Beach State Park dataset that the `test_data` variable contains. Crop the height and width of `test_data` to create the variable `input_data` that this example uses.

```
test_datacropRGB = imcrop(test_data(:,:,1:3),[2600, 3000, 2000, 2000]);
test_datacropInfrared = imcrop(test_data(:,:,4:6),[2600, 3000, 2000, 2000]);
test_datacropMask = imcrop(test_data(:,:,7),[2600, 3000, 2000, 2000]);
```

```
input_data(:,:,1:3) = test_datacropRGB;
input_data(:,:,4:6) = test_datacropInfrared;
input_data(:,:,7) = test_datacropMask;
```

Examine the `input_data` variable.

```
whos('input_data');
```

Write the input data into a text file that is passed as input to the generated executable.

```
WriteInputDatatoTxt(input_data);
[height, width, channels] = size(input_data);
```

Set Up a Code Generation Configuration Object for a Static Library

To generate code that targets an ARM-based device, create a configuration object for a library. Do not create a configuration object for an executable program. Set up the configuration object for generation of C++ source code only.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenCodeOnly = true;
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the library version and the architecture of the target ARM processor. For example, suppose that the target board is a HiKey/Rock960 board with ARMv8 architecture and ARM Compute Library version 19.05.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv8';
```

Assign the `DeepLearningConfig` property of the code generation configuration object `cfg` to the deep learning configuration object `dlcfg`.

```
cfg.DeepLearningConfig = dlcfg;
```

Generate C++ Source Code by Using codegen

```
codegen -config cfg segmentationUnetARM -args {ones(size(input_data),'uint16')} -d unet_predict
```

The code gets generated in the `unet_predict` folder that is located in the current working directory on the host computer.

Generate Zip File by Using packNGo

The `packNGo` function packages all relevant files into a compressed zip file.

```
zipFileName = 'unet_predict.zip';
bInfo = load(fullfile('unet_predict','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'fileName', zipFileName,'minimalHeaders', false, 'ignoreFileMissing',tr
```

The name of the generated zip file is `unet_predict.zip`.

Copy Generated Zip file to the Target Hardware

Copy the zip file into the target hardware board. Extract the contents of the zip file into a folder and delete the zip file from the hardware.

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetDir` with the destination folder for the files

On the Linux® platform, to transfer and extract the zip file on the target hardware, run these commands:

```
if isunix, system(['sshpass -p password scp -r '  fullfile(pwd,zipFileName) ' username@targetname
if isunix, system('sshpass -p password ssh username@targetname "if [ -d targetDir/unet_predict ]
if isunix, system(['sshpass -p password ssh username@targetname "unzip targetDir/' zipFileName '
if isunix, system(['sshpass -p password ssh username@targetname "rm -rf  targetDir/' zipFileName
```

On the Windows® platform, to transfer and extract the zip file on the target hardware, run these commands:

```
if ispc, system(['pscp.exe -pw password -r '  fullfile(pwd,zipFileName) ' username@targetname:ta
if ispc, system('plink.exe -l username -pw password targetname "if [ -d targetDir/unet_predict ]
if ispc, system(['plink.exe -l username -pw password targetname "unzip targetDir/' zipFileName '
if ispc, system(['plink.exe -l username -pw password targetname "rm -rf  targetDir/' zipFileName
```

Copy Supporting Files to the Target Hardware

Copy these files from the host computer to the target hardware:

- Input data, `input_data.txt`
- Makefile for creating the library, `unet_predict_rtw.mk`
- Makefile for building the executable program, `makefile_unet_arm_generic.mk`

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetDir` with the destination folder for the files

On the Linux® platform, to transfer the supporting files to the target hardware, run these commands:

```
if isunix, system('sshpass -p password scp unet_predict_rtw.mk username@targetname:targetDir/unet
if isunix, system('sshpass -p password scp input_data.txt username@targetname:targetDir/unet_pred
if isunix, system('sshpass -p password scp makefile_unet_arm_generic.mk username@targetname:targe
```

On the Windows® platform, to transfer the supporting files to the target hardware, run these commands:

```
if ispc, system('pscp.exe -pw password unet_predict_rtw.mk username@targetname:targetDir/unet_pre
if ispc, system('pscp.exe -pw password input_data.txt username@targetname:targetDir/unet_predict/
if ispc, system('pscp.exe -pw password makefile_unet_arm_generic.mk username@targetname:targetDi
```

Build the Library on the Target Hardware

To build the library on the target hardware, execute the generated makefile on the ARM hardware.

Make sure that you set the environment variables `ARM_COMPUTELIB` and `LD_LIBRARY_PATH` on the target hardware. See "Prerequisites for Deep Learning with MATLAB Coder" on page 38-2. The `ARM_ARCH` variable is used in Makefile to pass compiler flags based on the ARM Architecture. The `ARM_VER` variable is used in Makefile to compile the code based on the version of the ARM Compute library.

On the Linux host platform, run this command to build the library:

```
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetDir/unet_predict/
```

On the Windows host platform, run this command to build the library:

```
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetDir/unet_predict/
```

Create Executable on the Target

In these commands, replace `targetDir` with the destination folder where the library is generated. The variables `height`, `width`, and `channels` represent the dimensions of the input data.

`main_unet_arm_generic.cpp` is the C++ main wrapper file which invokes the segmentationUnetARM function and passes the input image to it. Build the library with the wrapper file to create the executable.

On the Linux host platform, to create the executable, run these commands:

```
if isunix, system('sshpass -p password scp main_unet_arm_generic.cpp username@targetname:targetD
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetDir/unet_predict/
```

On the Windows host platform, to create the executable, run these commands:

```
if ispc, system('pscp.exe -pw password main_unet_arm_generic.cpp username@targetname:targetDir/un
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetDir/unet_predict/
```

Run the Executable on the Target Hardware

Run the Executable on the target hardware with the input image file `input_data.txt`.

On the Linux host platform, run this command:

```
if isunix, system('sshpass -p password ssh username@targetname "cd targetDir/unet_predict/; ./une
```

On the Windows host platform, run this command:

```
if ispc, system('plink.exe -l username -pw password targetname "cd targetDir/unet_predict/; ./une
```

The `unet` executable accepts the input data. Because of the large size of `input_data` (2001x2001x7), it is easier to process the input image in patches. The executable splits the input image into multiple patches, each corresponding to network input size. The executable performs prediction on the pixels in one particular patch at a time and then combines all the patches together.

Transfer the Output from Target Hardware to MATLAB

Copy the generated output file `output_data.txt` back to the current MATLAB session. On the Linux platform, run:

```
if isunix, system('sshpass -p password scp username@targetname:targetDir/unet_predict/output_data
```

To perform the same action on the Windows platform, run:

```
if ispc, system('pscp.exe -pw password username@targetname:targetDir/unet_predict/output_data.tx
```

Store the output data in the variable `segmentedImage`:

```
segmentedImage = uint8(importdata('output_data.txt'));
segmentedImage = reshape(segmentedImage,[height,width]);
```

To extract only the valid portion of the segmented image, multiply it by the mask channel of the input data.

```
segmentedImage = uint8(input_data(:,:,7)~=0) .* segmentedImage;
```

Remove the noise and stray pixels by using the `medfilt2` function.

```
segmentedImageCodegen = medfilt2(segmentedImage,[5,5]);
```

Display U-Net Segmented data

This line of code creates a vector of the class names. classNames = net.Layers(end).Classes; disp(classNames);

Overlay the labels on the segmented RGB test image and add a color bar to the segmented image.

Display input data

```
figure(1);
imshow(histeq(input_data(:,:,1:3)));
title('Input Image');
```

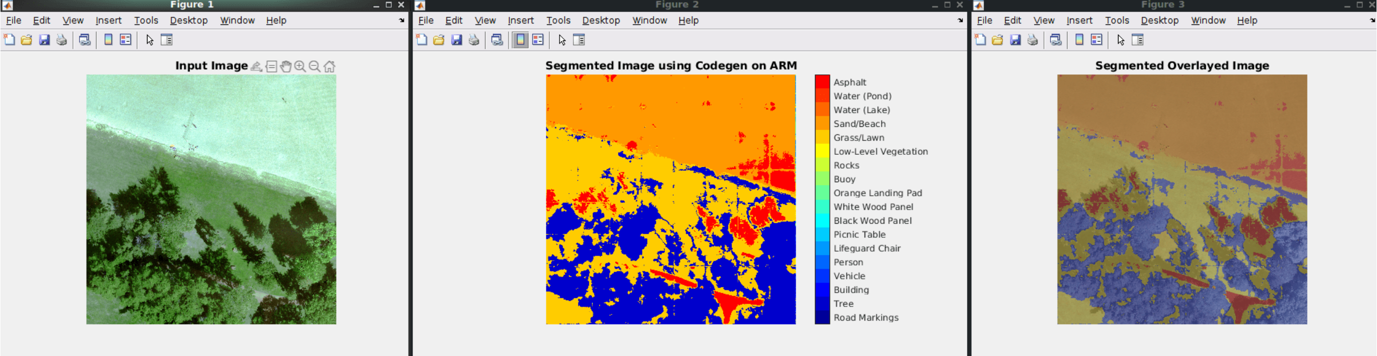
```
cmap = jet(numel(classNames));
segmentedImageOut = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmentedIm
figure(2);
imshow(segmentedImageOut);
```

Display segmented data

```
title('Segmented Image using Codegen on ARM');
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
```

Display segmented overlay Image

```
segmentedImageOverlay = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segment
figure(3);
imshow(segmentedImageOverlay);
title('Segmented Overlayed Image');
```



References

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *arXiv preprint arXiv:1505.04597,* 2015.

[2] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918, 2017.

[3] Reference Input Data used is part of the Hamlin Beach State Park data. The following steps can be used to download the data for further evaluation.

```
if ~exist(fullfile(pwd,'data'))
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware` | `packNGo`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Deep Learning on ARM Targets” on page 38-56
- Semantic Segmentation of Multispectral Images Using Deep Learning

Code Generation for LSTM Network on Raspberry Pi

This example shows how to generate code for a pretrained long short-term memory (LSTM) network that uses the ARM® Compute Library and deploy the code on a Raspberry Pi™ target. In this example, the LSTM network predicts the Remaining Useful Life (RUL) of a machine. The network takes as input time series data sets that represent various sensors in the engine. The network returns the Remaining Useful Life of an engine, measured in cycles, as its output.

This example uses the Turbofan Engine Degradation Simulation Data Set as described in [1]. This data set contains 100 training observations and 100 test observations. The training data contains simulated time series data for 100 engines. Each sequence has 17 features, varies in length, and corresponds to a full run to failure (RTF) instance. The test data contains 100 partial sequences and corresponding values of the Remaining Useful Life at the end of each sequence.

This example uses a pretrained LSTM network. For more information on how to train an LSTM network, see the example “Sequence Classification Using Deep Learning” (Deep Learning Toolbox).

This example demonstrates two different approaches for performing prediction by using an LSTM network:

- The first approach uses a standard LSTM network and runs inference on a set of time series data.
- The second approach leverages the stateful behavior of the same LSTM network. In this method, you pass a single timestep of data at a time, and have the network update its state at each time step.

This example uses the PIL based workflow to generate a MEX function, which in turn calls the executable generated in the target hardware from MATLAB.

Notes:

- The code lines in this example are commented out. Uncomment them before you run the example.
- The ARM Compute library version that this example uses might not be the latest version that code generation supports. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware and Software” on page 38-2.
- This example is not supported in MATLAB Online.

Prerequisites

- MATLAB® Coder™
- Embedded Coder®
- Deep Learning Toolbox™
- MATLAB Coder Interface for Deep Learning Libraries. To install this support package, use the Add-On Explorer.
- MATLAB Support Package for Raspberry Pi Hardware. To install this support package, use the Add-On Explorer.
- Raspberry Pi hardware
- ARM Compute Library (on the target ARM hardware)
- Environment variables for the compilers and libraries. For setting up the environment variables, see “Environment Variables” on page 38-4.

Set Up a Code Generation Configuration Object for a Static Library

To generate a PIL MEX function for a specified entry-point function, create a code configuration object for a static library and set the verification mode to 'PIL'. Set the target language to C++.

```
% cfg = coder.config('lib', 'ecoder', true);
% cfg.VerificationMode = 'PIL';
% cfg.TargetLang = 'C++';
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the Compute Library version. For this example, suppose that the ARM Compute Library in the Raspberry Pi hardware is version 19.05.

```
% dlcfg = coder.DeepLearningConfig('arm-compute');
% dlcfg.ArmComputeVersion = '19.05';
```

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object.

```
% cfg.DeepLearningConfig = dlcfg;
```

Create a Connection to the Raspberry Pi

Use the MATLAB Support Package for Raspberry Pi Support Package function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiname` with the name of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
% r = raspi('raspiname','username','password');
```

Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for Raspberry Pi and attach it to the code generation configuration object.

```
% hw = coder.hardware('Raspberry Pi');
% cfg.Hardware = hw;
```

First Approach: Generate PIL MEX Function for LSTM Network

In this approach, you generate code for the entry-point function `rul_lstmnet_predict`.

The rul_lstmnet_predict.m entry-point function takes an entire time series data set as an input and passes it to the network for prediction. Specifically, the function uses the LSTM network that is trained in the example “Sequence Classification Using Deep Learning” (Deep Learning Toolbox). The function loads the network object from the `rul_lstmnet.mat` file into a persistent variable and reuses this persistent object in subsequent prediction calls. A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of a data sequence.

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
type('rul_lstmnet_predict.m')
```

```
function out =  rul_lstmnet_predict(in) %#codegen

% Copyright 2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rul_lstmnet.mat');
end


out = mynet.predict(in);
```

To generate code by using the `codegen` command, use the `coder.typeof` function to specify the type and size of the input argument to the entry-point function. In this example, the input is of double data type with a feature dimension value of 17 and a variable sequence length. Specify the sequence length as variable-size to perform prediction on an input sequence of any length.

```
% matrixInput = coder.typeof(double(0),[17 Inf],[false true]);
```

Run the codegen command to generate a PIL based mex function `rul_lstmnet_predict_pil` on the host platform.

```
% codegen -config cfg rul_lstmnet_predict -args {matrixInput} -report
```

Run Generated PIL MEX Function on Test Data

Load the MAT-file `RULTestData`. This MAT-file stores the variables `XTest` and `YTest` that contain sample timeseries of sensor readings on which you can test the generated code. This test data is taken from the example “Sequence Classification Using Deep Learning” (Deep Learning Toolbox) after data pre-processing.

```
load RULTestData;
```

The `XTest` variable contains 100 input observations. Each observation has 17 features with varying sequence length.

```
XTest(1:5)
```

```
ans=5×1 cell array
    {17×31  double}
    {17×49  double}
    {17×126 double}
    {17×106 double}
    {17×98  double}
```

The `YTest` variable contains 100 output observations that correspond to the `XTest` input variable. Each output observation is a Remaining Useful Life (RUI) value, measured in cycles, for each time step data in entire sequence.

```
YTest(1:5)
```

```
ans=5×1 cell array
    {[
    {[
    {[150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150
    {[                                                                    150 150 150 150 150 150
```

```
{[
```

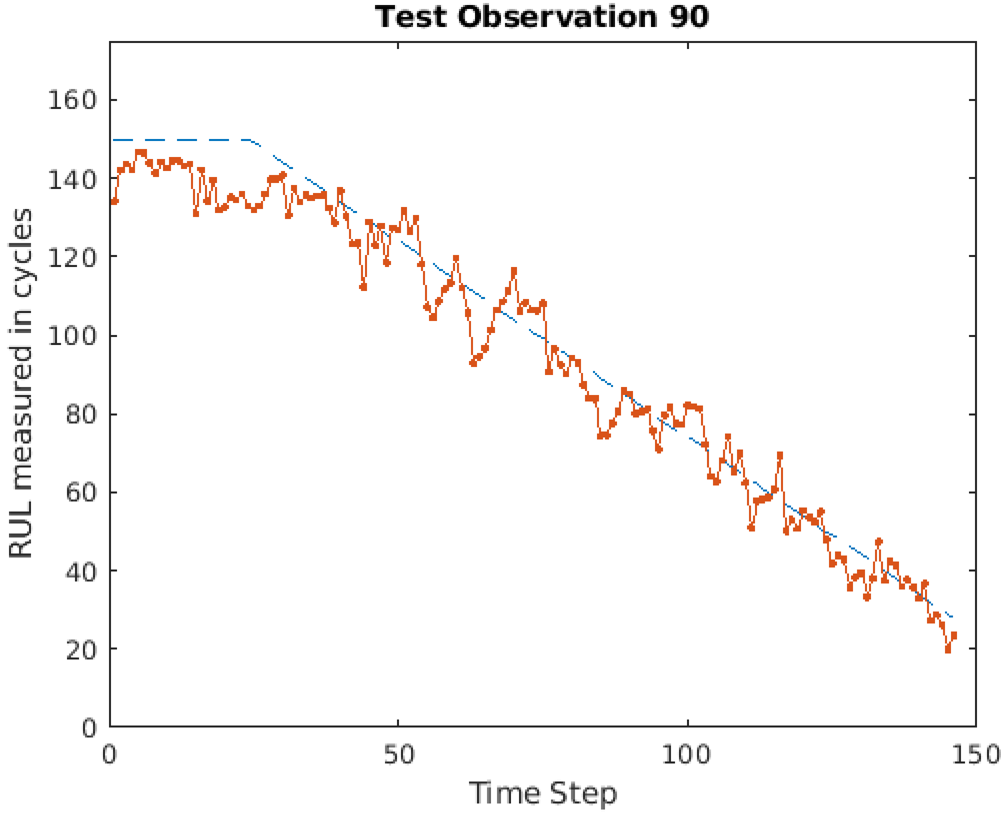
Run the generated MEX function `rul_lstmnet_predict_pil` on a random test data set.

```
% idx = randperm(numel(XTest), 1);
% inputData = XTest{idx};

% YPred1 = rul_lstmnet_predict_pil(inputData);
```

Compare Predictions with Test Data

Use a plot to compare the MEX output data with the test data.

```
% figure('Name', 'Standard LSTM', 'NumberTitle', 'off');
%
% plot(YTest{idx},'--')
% hold on
% plot(YPred1,'.-')
% hold off
%
% ylim([0 175])
% title("Test Observation " + idx)
% xlabel("Time Step")
% ylabel("RUL measured in cycles")
```



Clear PIL

```
% clear rul_lstmnet_predict_pil;
```

Second Approach: Generate PIL MEX Function for Stateful LSTM Network

Instead of passing the entire timeseries data all at once to `predict`, you can run prediction by streaming the input data segment-wise by using the `predictAndUpdateState` function.

The entry-point function rul_lstmnet_predict_and_update.m accepts a single-timestep input and processes it by using the `predictAndUpdateState` (Deep Learning Toolbox) function. `predictAndUpdateState` returns a prediction for the input timestep and updates the network so that subsequent parts of the input are treated as subsequent timesteps of the same sample.

```
type('rul_lstmnet_predict_and_update.m')
```

```
function out = rul_lstmnet_predict_and_update(in) %#codegen

% Copyright 2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rul_lstmnet.mat');
end

[mynet, out] = predictAndUpdateState(mynet, in);

end
```

Create the input type for the `codegen` command. Because `rul_lstmnet_predict_and_update` accepts a single timestep data in each call, specify the input type `matrixInput` to have a fixed sequence length of 1 instead of a variable sequence length.

```
% matrixInput = coder.typeof(double(0),[17 1]);
```

Run the `codegen` command to generate PIL based mex function `rul_lstmnet_predict_and_update_pil` on the host platform.

```
% codegen -config cfg rul_lstmnet_predict_and_update -args {matrixInput} -report
```

Run Generated PIL MEX Function on Test Data

```
% Run generated MEX function(|rul_lstmnet_predict_and_update_pil|) for each
% time step data in the inputData sequence.

% sequenceLength = size(inputData,2);
% YPred2 = zeros(1, sequenceLength);
% for i=1:sequenceLength
%     inTimeStep = inputData(:,i);
%     YPred2(:, i) = rul_lstmnet_predict_and_update_pil(inTimeStep);
% end
```

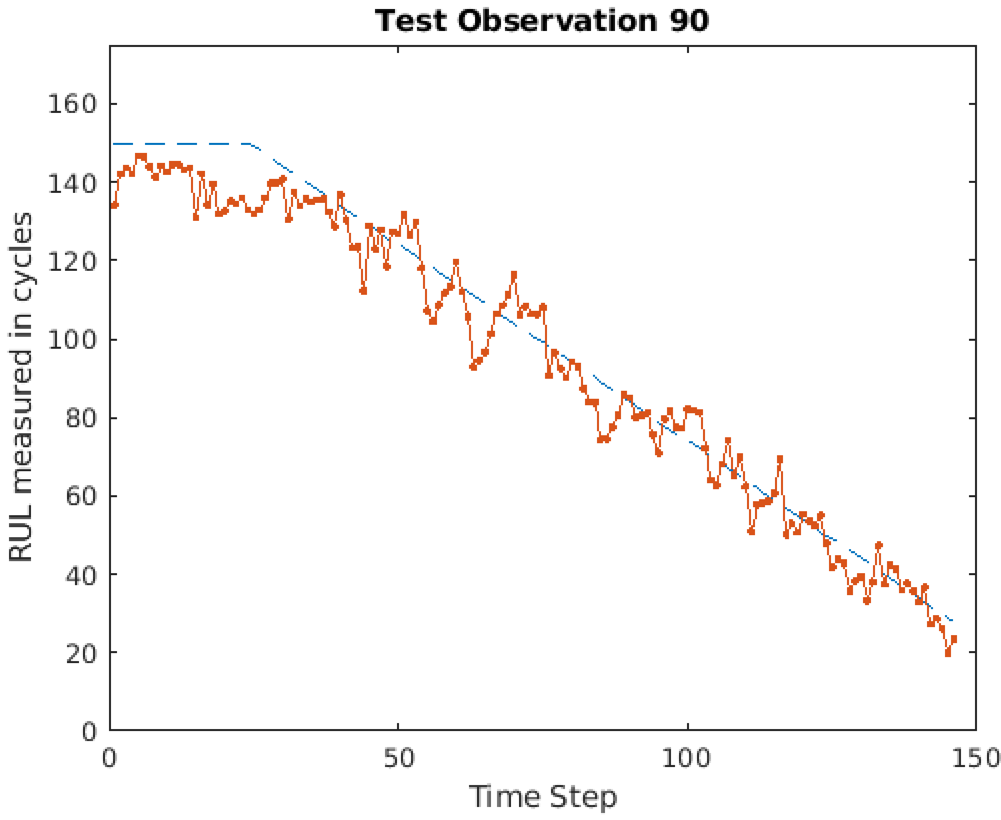
After you pass all timesteps, one at a time, to the `rul_lstmnet_predict_and_update` function, the resulting output is the same as that in the first approach in which you passed all inputs at once.

Compare Predictions with Test Data

Use a plot to compare the MEX output data with the test data.

```
% figure('Name', 'Statefull LSTM', 'NumberTitle', 'off');
%
```

```
%
% plot(YTest{idx},'--')
% hold on
% plot(YPred2,'.-')
% hold off
%
% ylim([0 175])
% title("Test Observation " + idx)
% xlabel("Time Step")
% ylabel("RUL measured in cycles")
```



Clear PIL

```
% clear rul_lstmnet_predict_and_update_pil;
```

References

[1] Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In Prognostics and Health Management, 2008. PHM 2008. International Conference on, pp. 1-9. IEEE, 2008.

See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware` | `predictAndUpdateState`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Deep Learning on ARM Targets” on page 38-56
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)

Code Generation for LSTM Network That Uses Intel MKL-DNN

This example shows how to generate code for a pretrained long short-term memory (LSTM) network that uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).This example generates a MEX function that makes predictions for each step of an input timeseries. The example demonstrates two approaches. The first approach uses a standard LSTM network. The second approach leverages the stateful behavior of the same LSTM network. This example uses textual descriptions of factory events that can be classified into one of these four categories: Electronic Failure, Leak, Mechanical Failure, and Software Failure. The example uses a pretrained LSTM network. For more information on training a network, see the “Classify Text Data Using Deep Learning” (Text Analytics Toolbox).

Third-Party Prerequisites

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- For a list of processors that support the MKL-DNN library, see MKLDNN CPU Support
- For more information on the supported versions of the compilers and libraries, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2

This example is supported on Mac®, Linux® and Windows® platforms and not supported for MATLAB Online.

Prepare Input

Load the `wordEncoding` MAT-file. This MAT-file stores the words encoded as numerical indices. This encoding was performed during the training of the network. For more information, see “Classify Text Data Using Deep Learning” (Text Analytics Toolbox).

```
load("wordEncoding.mat");
```

Create a string array containing the new reports to classify the event type.

```
reportsNew = [ ...
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."
    "At times mechanical arrangement software freezes."
    "Mixer output is stuck."];
```

Tokenize the input string by using the `preprocessText` function.

```
documentsNew = preprocessText(reportsNew);
```

Use the `doc2sequence` (Text Analytics Toolbox) function to convert documents to sequences.

```
XNew = doc2sequence(enc,documentsNew);
labels = categorical({'Electronic Failure', 'Leak', 'Mechanical Failure', 'Software Failure'});
```

The `lstm_predict` Entry-Point Function

A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of a data sequence. The `lstm_predict.m` entry-point function takes an input sequence and passes it to a trained LSTM network for prediction. Specifically, the function uses the LSTM network that is trained in the example “Classify Text Data Using Deep Learning” (Text Analytics Toolbox). The function loads the network object from the `textClassifierNetwork.mat` file into a

persistent variable and then performs prediction. On subsequent calls, the function reuses the persistent object.

```
type('lstm_predict.m')
```

```
function out = lstm_predict(in)
%#codegen

%   Copyright 2020 The MathWorks, Inc.

    persistent mynet;

    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork('textClassifierNetwork.mat');
    end

    out = predict(mynet, in);
end
```

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` (Deep Learning Toolbox) function.

Generate MEX

To generate code, create a code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a MKL-DNN deep learning configuration object. Assign it to the `DeepLearningConfig` property of the code configuration object.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Use the `coder.typeof` function to specify the type and size of the input argument to the entry-point function. In this example, the input is of double data type with a feature dimension value of 1 and a variable sequence length.

```
matrixInput = coder.typeof(double(0),[1 Inf],[false true]);
```

Generate a MEX function by running the `codegen` command.

```
codegen -config cfg lstm_predict -args {matrixInput} -report
```

```
Code generation successful: View report
```

Run Generated MEX

Call `lstm_predict_mex` on the first observation.

```
YPred1 = lstm_predict_mex(XNew{1});
```

`YPred1` contains the probabilities for the four classes. Find the predicted class by calculating the index of the maximum probability.

```
[~, maxIndex] = max(YPred1);
```

Associate the indices of max probability to the corresponding label. Display the classification. From the results, you can see that the network predicted the first event to be a Leak.

```
predictedLabels1 = labels(maxIndex);
disp(predictedLabels1)
```

```
     Leak
```

Generate MEX that Accepts Multiple Observations

If you want to perform prediction on many observations at once, you can group the observations together in a cell array and pass the cell array for prediction. The cell array must be a column cell array, and each cell must contain one observation. The sequence lengths of the inputs might vary. In this example, `XNew` contains five observations. To generate a MEX function that can accept `XNew` as input, specify the input type to be a 5-by-1 cell array. Specify that each cell be of the same type as `matrixInput`.

```
matrixInput = coder.typeof(double(0),[1 Inf],[false true]);
cellInput = coder.typeof({matrixInput}, [5 1]);
codegen -config cfg lstm_predict -args {cellInput} -report
```

```
Code generation successful: View report
```

Run the generated MEX function with `XNew` as input.

```
YPred2 = lstm_predict_mex(XNew);
```

`YPred2` is 5-by-4 cell array. Find the indices that have maximum probability for each of the five inputs and classify them.

```
[~, maxIndex] = max(YPred2, [], 2);
predictedLabels2 = labels(maxIndex);
disp(predictedLabels2)
```

```
     Leak      Mechanical Failure      Mechanical Failure      Software Failure      Electronic F
```

Generate MEX with Stateful LSTM

Instead of passing the entire timeseries to `predict` in a single step, you can run prediction on an input by streaming in one timestep at a time and using the function `predictAndUpdateState` (Deep Learning Toolbox). This function accepts an input, produces an output prediction, and updates the internal state of the network so that future predictions take this initial input into account.

The entry-point function `lstm_predict_and_update.m` accepts a single-timestep input and processes the input using the `predictAndUpdateState` function. The `predictAndUpdateState` function returns a prediction for the input timestep and updates the network so that subsequent inputs are treated as subsequent timesteps of the same sample. After passing in all timesteps, one at a time, the resulting output is identical to the case where all timesteps were passed in as a single input.

```
type('lstm_predict_and_update.m')
```

```
function out = lstm_predict_and_update(in)
%#codegen

%   Copyright 2020 The MathWorks, Inc.

    persistent mynet;

    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork('textClassifierNetwork.mat');
```

```
    end

    [mynet, out] = predictAndUpdateState(mynet,in);
end
```

Generate code for `lstm_predict_and_update`. Because this function accepts a single timestep at each call, specify `matrixInput` to have a fixed sequence dimension of 1 instead of a variable sequence length.

```
matrixInput = coder.typeof(double(0),[1 1]);
codegen -config cfg lstm_predict_and_update -args {matrixInput} -report
```

```
Code generation successful: View report
```

Run the generated MEX on the first observation.

```
sequenceLength = size(XNew{1},2);
for i=1:sequenceLength
    inTimeStep = XNew{1}(:,i);
    YPred3 = lstm_predict_and_update_mex(inTimeStep);
end
clear mex;
```

Find the index that has the highest probability and map it to the labels.

```
[~, maxIndex] = max(YPred3);
predictedLabels3 = labels(maxIndex);
disp(predictedLabels3)
```

```
     Leak
```

See Also

`codegen` | `coder.DeepLearningConfig` | `coder.typeof` | `doc2sequence`

More About

- “Classify Text Data Using Deep Learning” (Text Analytics Toolbox)
- “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2

Code Generation for Convolutional LSTM Network That Uses Intel MKL-DNN

This example shows how to generate a MEX function for a deep learning network containing both convolutional and bidirectional long short-term memory (BiLSTM) layers that uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). The generated MEX function reads the data from a specified video file as a sequence of video frames and outputs a label that classifies the activity in the video. For more information on the training of this network, see the example “Classify Videos Using Deep Learning” (Deep Learning Toolbox).

Third-Party Prerequisites

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- For a list of processors that support the MKL-DNN library, see MKLDNN CPU Support
- For more information on the supported versions of the compilers and libraries, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2

This example is supported on Mac®, Linux® and Windows® platforms and not supported for MATLAB Online.

Prepare Input

Read the video file `pushup.mp4` by using the `readvideo` helper function included with this example in a supporting file. To view the video, loop over the individual frames of the video file and use the `imshow` function.

```
filename = "pushup.mp4";
video = readVideo(filename);
numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:,:,i);
    imshow(frame/255);
    drawnow
end
```

Center-crop the input video frames to the input size of the trained network by using the the `centerCrop` helper function attached as a supporting file.

```
inputSize = [224 224 3];
video = centerCrop(video,inputSize);
```

The `video_classify` Entry-Point Function

The `video_classify.m` entry-point function takes image sequences and passes it to a trained network for prediction. This function uses the convolutional LSTM network that is trained in the example “Classify Videos Using Deep Learning” (Deep Learning Toolbox). The function loads the network object from the file `net.mat` file into a persistent variable and then uses the `classify` (Deep Learning Toolbox) function to perform the prediction. On subsequent calls, the function reuses the already loaded persistent object.

```
type('video_classify.m')
```

```
function out = video_classify(in) %#codegen

% During the execution of the first function call, the network object is
% loaded in the persistent variable mynet. In subsequent calls, this loaded
% object is reused.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('net.mat');
end
```

```
% Provide input and perform prediction
out = classify(mynet,in);
```

Generate MEX

To generate a MEX function, create a `coder.MexCodeConfig` object `cfg`. Set the `TargetLang` property of `cfg` to C++. Use the `coder.DeepLearningConfig` function to create a deep learning configuration object for MKL-DNN. Assign it to the `DeepLearningConfig` property of the `cfg`.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Run the `getVideoClassificationNetwork` helper function to download the video classification network and save the network in the MAT file `net.mat`.

```
getVideoClassificationNetwork();
```

Use the `coder.typeof` function to specify the type and size of the input argument to the entry-point function. In this example, the input is of double type with size `[224 224 3]` and a variable sequence length.

```
Input = coder.typeof(double(0),[224 224 3 Inf],[false false false true]);
```

Generate a MEX function by running the `codegen` command.

```
codegen -config cfg video_classify -args {Input} -report
```

```
Code generation successful: View report
```

Run generated MEX

Run the generated MEX function with center-cropped video input.

```
output = video_classify_mex(video)
```

```
output = categorical
     pushup
```

Overlay the prediction on to the input video.

```
video = readVideo(filename);
numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:,:,i);
    frame = insertText(frame, [1 1], char(output), 'TextColor', [255 255 255],'FontSize',30, 'Bo
    imshow(frame/255);
    drawnow
end
```

See Also

`codegen` | `coder.DeepLearningConfig` | `coder.typeof`

More About

- “Classify Videos Using Deep Learning” (Deep Learning Toolbox)
- “Code Generation for Deep Learning Networks with MKL-DNN” on page 38-29
- “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2

Cross Compile Deep Learning Code for ARM Neon Targets

This example shows how to cross-compile the generated deep learning code to create a library or an executable, and then deploy the library or executable on an ARM® target such as Hikey 960 or Rock 960. This example uses the `codegen` command.

Cross compiling the deep learning code for ARM® targets involves these steps:

- Configure the installed cross-compiler toolchain to perform compilation on the host MATLAB®. The compilation happens when you run the `codegen` command in MATLAB in the host computer.
- Use the `codegen` command to build the generated code and create a library or an executable on the host computer.
- Copy the generated library or executable and other supporting files to the target hardware. If you generate a library on the host computer, compile the copied makefile on the target to create an executable.
- Run the generated executable on the target ARM hardware.

You can use this workflow for any ARM Neon target that supportes the Neon|SIMD instruction set. This example is supported only for host Linux® platforms.

Prerequisites

- ARM processor that supports the Neon|SIMD extension
- ARM Compute Library (on the host computer)
- MATLAB® Coder™
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™
- The support package Deep Learning Toolbox Model for Inception-v3 Network
- Image Processing Toolbox™
- For deployment on armv7 (32 bit Arm Architecture) target, GNU/GCC `g++-arm-linux-gnueabihf` toolchain
- For deployment on armv8 (64 bit Arm Architecture) target, GNU/GCC `g++-aarch64-linux-gnu` toolchain
- Environment variables for the cross compilers and libraries

For information about how to install the cross-compiler toolchain and set up the associated environment variable, see “Cross-Compile Deep Learning Code That Uses ARM Compute Library” on page 38-37.

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For information about supported versions of libraries and about environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” on page 38-2.

The code lines in this example are commented out. Uncomment them before you run the example.

This example in not supported in MATLAB Online.

The `inception_predict_arm` Entry-Point Function

This example uses the Inception-V3 image classification network. A pretrained Inception-V3 network for MATLAB is available in the support package Deep Learning Toolbox Model for Inception-V3

Network. The `inception_predict_arm` entry-point function loads the Inception-V3 network into a persistent network object. On subsequent calls to the function, the persistent object is reused.

```
type inception_predict_arm
```

```
function out = inception_predict_arm(in)

persistent net;
if isempty(net)
    net = coder.loadDeepLearningNetwork('inceptionv3','inceptionv3');
end

out = net.predict(in);

end
```

Set up a Deep Learning Configuration Object

Create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute library.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
```

For classifying the input image `peppers.png`, convert the image to a text file.

```
% generateImagetoTxt('peppers.png');
```

First Approach: Create Static Library for Entry-Point Function on Host

In this approach, you first cross-compile the generated code to create a static library on the host computer. You then transfer the generated static library, the ARM Compute library files, the makefile, and other supporting files to the target hardware. You run the makefile on the target hardware to generate the executable. Finally, you run the executable on the target hardware.

Set Up a Code Generation Configuration Object

Create a code generation configuration object for a static library. Specify the target language as C++.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
```

Attach the deep learning configuration object to the code generation configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Configure the Cross-Compiler Toolchain

Configure the cross-compiler toolchain based on the ARM Architecture of the target device.

```
%   cfg.Toolchain =  'Linaro AArch64 Linux v6.3.1';% When the Arm Architecture is armv8

%   cfg.Toolchain =  'Linaro AArch32 Linux v6.3.1';% When the Arm Architecture is armv7
```

Generate Static Library on Host Computer by Using codegen

Use the `codegen` command to generate code for the entry-point function, build the generated code, and create static library for the target ARM architecture.

```
% codegen -config cfg inception_predict_arm -args {ones(299,299,3,'single')} -d arm_compute_cc_l
```

Copy the Generated Cross-Compiled Static Library to Target hardware

Copy the static library, the bin files, and the header files from the generated folder `arm_compute_cc_lib` to the target ARM hardware. In this code line and other code lines that follow, replace:

- password with your password
- username with your username
- hostname with the name of your device
- targetDir with the destination folder for the files

```
% system('sshpass -p password scp -r arm_compute_cc_lib/*.bin arm_compute_cc_lib/*.lib arm_comput
```

Copy the ARM Compute Library Files to Target Hardware

The executable uses the ARM Compute library files during runtime. The target board does not need header files while generating the executable and running the executable. Copy the library to the desired path.

```
% system(['sshpass -p password scp -r ' fullfile(getenv('ARM_COMPUTELIB'),'lib') ' username@hostn
```

Copy Supporting Files to Target Hardware

Copy these files to the target ARM hardware:

- Makefile `Makefile_Inceptionv3` to generate executable from static library.
- Input Image `inputimage.txt` that you want to classify.
- The text file `synsetWords.txt` that contains the ClassNames returned by `net.Layers(end).Classes`
- The main wrapper file `main_inception_arm.cpp` that calls the code generated for the `inception_predict_arm` function.

```
% system('sshpass -p password scp synsetWords.txt ./Makefile_Inceptionv3 ./inputimage.txt ./main_
```

Create the Executable on the Target

Compile the makefile on the target to generate the executable from the static library. This makefile links the static library with the main wrapper file `main_inception_arm.cpp` and generates the executable.

```
% system('sshpass -p password ssh username@hostname "make -C targetDir -f Makefile_Inceptionv3 ar
```

Run the Executable on the Target

Run the generated executable on the target. Make sure to export LD_LIBRARY_PATH that points to the ARM Compute library files while running executable.

```
% system('sshpass -p password ssh username@hostname "export LD_LIBRARY_PATH=targetDir/lib; cd tar
```

Second Approach: Create Executable for Entry-Point function on Host

In this approach, you first cross-compile the generated code to create an executable on the host computer. You then transfer the generated executable, the ARM Compute library files, and other supporting files to the target hardware. Finally, you run the executable on the target hardware.

Set Up a Code Generation Configuration Object

Create a code generation configuration object for an generating an executable. Set the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Attach the deep learning configuration object to the code generation configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Declare the main wrapper file `main_inception_arm.cpp` as the custom source file.

```
cfg.CustomSource = 'main_inception_arm.cpp';
```

Configure the Cross-Compiler Toolchain

Configure the cross-compiler toolchain based on the ARM Architecture of the target device.

```
%   cfg.Toolchain =  'Linaro AArch64 Linux v6.3.1'; % When the Arm Architecture is armv8,

%   cfg.Toolchain =  'Linaro AArch32 Linux v6.3.1';% When the Arm Architecture is armv7,
```

Generate Executable on the Host Computer by Using codegen

Use the `codegen` command to generate code for the entry-point function, build the generated code, and create an executablle for the target ARM architecture.

```
% codegen -config cfg inception_predict_arm -args {ones(299,299,3,'single')} -d arm_compute_cc_e
```

Copy the Generated Executable to the Target Hardware

Copy the generated executable and the bin files to the target ARM hardware. In this code line and other code lines that follow, replace:

- password with your password
- username with your username
- hostname with the name of your device
- `targetDir` with the destination folder for the files

```
% system('sshpass -p password scp -r arm_compute_cc_exe/*.bin username@hostname:targetDir/');
% system('sshpass -p password scp inception_predict_arm.elf username@hostname:targetDir/');
```

Copy the ARM Compute Library Files to the Target Hardware

The executable uses the ARM Compute library files during runtime. It does not use header files at runtime. Copy the library files to the desired path.

```
% system(['sshpass -p password scp -r ' fullfile(getenv('ARM_COMPUTELIB'),'lib') ' username@host
```

Copy Supporting Files to the Target Hardware

Copy these files to the target ARM hardware:

- Input Image `inputimage.txt` that you want to classify.

- The text file `synsetWords.txt` that contains the ClassNames returned by `net.Layers(end).Classes`
- The main wrapper file `main_inception_arm.cpp` that calls the code generated for the `inception_predict_arm` function.

```
% system('sshpass -p password scp synsetWords.txt ./inputimage.txt ./main_inception_arm.cpp usern
```

Run the Executable on the Target Hardware

Run the generated executable on the target. Make sure to export LD_LIBRARY_PATH that points to the ARM Compute library files while running executable.

```
% system('sshpass -p password ssh username@hostname "export LD_LIBRARY_PATH=targetDir/lib; cd tar
```

Transfer the Output Data from Target to MATLAB

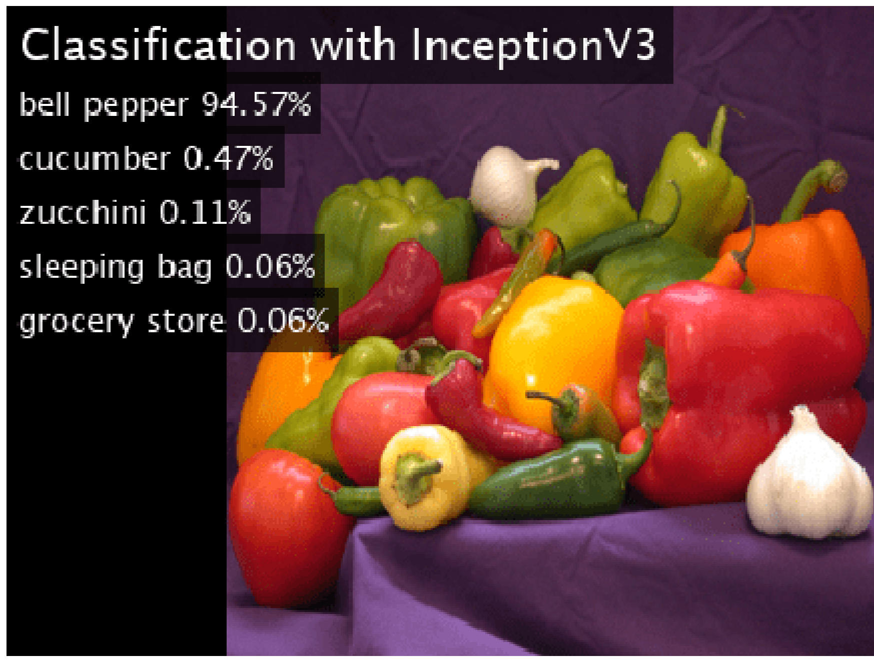
Copy the generated output back to the current MATLAB session on the host computer.

```
% system('sshpass -p password scp username@hostname:targetDir/out.txt ./');
```

Map Prediction Scores to Labels

Map the top five prediction scores to corresponding labels in the trained network.

```
% outputImage = mapPredictionScores;

% Display the overlayed Image with Classification Scores.

% imshow(outputImage);
```



See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” on page 38-32
- “Code Generation for Deep Learning on ARM Targets” on page 38-56
- “Cross-Compile Deep Learning Code That Uses ARM Compute Library” on page 38-37

Code Generation for Quantized Deep Learning Network on Raspberry Pi

Deep learning uses neural network architectures that contain many processing layers, including convolutional layers. Deep learning models typically work on large sets of labeled data. Performing inference on these models is computationally intensive, consuming significant amount of memory. Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. Deep Neural networks trained in MATLAB use single-precision floating point data types.. Even networks that are small in size require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and smaller memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

You can use Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. Then, you can use MATLAB Coder™ to generate optimized code for the quantized network. The generated code takes advantage of ARM® processor SIMD by using the ARM Compute library. The generated code can be integrated into your project as source code, static or dynamic libraries, or executables that you can deploy to a variety of ARM CPU platforms such as Raspberry Pi™.

This example shows how to generate C++ code for a convolutional neural network that uses the ARM Compute Library and performs inference computations in 8-bit integers.

This example is not supported for MATLAB Online.

Third-Party Prerequisites

- Raspberry Pi hardware
- ARM Compute Library (on the target ARM hardware)
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see Third-Party Hardware and Software. For setting up the environment variables, see Environment Variables.

Example: Classify Images Using SqueezeNet

In this example, you use MATLAB Coder to generate optmized C++ code for a quantized deep convolutional neural network and classify an image. The example uses the pretrained `squeezenet` (Deep Learning Toolbox) convolutional neural network.

SqueezeNet has been trained on the ImageNet dataset containing images of 1000 object categories. The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

This example consists of four steps:

1. Modify the SqueezeNet neural network to classify a smaller subset of images containing five object categories using transfer learning..
2. Quantize the modified SqueezeNet network.

3. Generate code for the quantized network by using the codegen command. The generated code runs on Raspberry Pi target via PIL execution.
4. Execute the generated PIL MEX on Raspberry Pi.

Transfer Learning Using SqueezeNet

To perform classification on a new set of images, you must fine-tune a pretrained SqueezeNet convolutional neural network by using transfer learning. In transfer learning, you take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network by using transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task by using a smaller number of training images.

Load Training Data

Unzip and load the new images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');

numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,4);
img = imtile(imds, 'Frames', idx);

figure
imshow(img)
title('Random Images from Training Dataset');
```

Random Images from Training Dataset

Load Pretrained Network

Load the pretrained SqueezeNet network.

```
net=squeezenet;
```

The object `net` contains the `DAGNetwork` object. The first layer is the image input layer that accepts input images of size 227-by-227-by-3, where 3 is the number of color channels. Use the `analyzeNetwork` (Deep Learning Toolbox) function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
inputSize = net.Layers(1).InputSize;
analyzeNetwork(net);
```

Replace Final Layers

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers, `'conv10'` and `'ClassificationLayer_predictions'` in SqueezeNet, contain information about how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels.

To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set. You can do this manually or use the helper function `findLayersToReplace` to find these layers automatically.

This is the `findLayersToReplace` helper Function:

```
type findLayersToReplace.m
```

```
% findLayersToReplace(lgraph) finds the single classification layer and the
% preceding learnable (fully connected or convolutional) layer of the layer
% graph lgraph.

function [learnableLayer,classLayer] = findLayersToReplace(lgraph)

if ~isa(lgraph,'nnet.cnn.LayerGraph')
    error('Argument must be a LayerGraph object.')
end

% Get source, destination, and layer names.
src = string(lgraph.Connections.Source);
dst = string(lgraph.Connections.Destination);
layerNames = string({lgraph.Layers.Name}');

% Find the classification layer. The layer graph must have a single
% classification layer.
isClassificationLayer = arrayfun(@(l) ...
    (isa(l,'nnet.cnn.layer.ClassificationOutputLayer')|isa(l,'nnet.layer.ClassificationLayer')),
    lgraph.Layers);

if sum(isClassificationLayer) ~= 1
    error('Layer graph must have a single classification layer.')
end
classLayer = lgraph.Layers(isClassificationLayer);


% Traverse the layer graph in reverse starting from the classification
% layer. If the network branches, throw an error.
currentLayerIdx = find(isClassificationLayer);
while true

    if numel(currentLayerIdx) ~= 1
        error('Layer graph must have a single learnable layer preceding the classification layer
    end

    currentLayerType = class(lgraph.Layers(currentLayerIdx));
    isLearnableLayer = ismember(currentLayerType, ...
        ['nnet.cnn.layer.FullyConnectedLayer','nnet.cnn.layer.Convolution2DLayer']);

    if isLearnableLayer
```

```
        learnableLayer =  lgraph.Layers(currentLayerIdx);
        return
    end

    currentDstIdx = find(layerNames(currentLayerIdx) == dst);
    currentLayerIdx = find(src(currentDstIdx) == layerNames);

end



end
```

To use this function to replace the final layers, run these commands:

```
lgraph = layerGraph(net);
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
numClasses = numel(categories(imdsTrain.Labels));

newConvLayer =  convolution2dLayer([1, 1],numClasses,'WeightLearnRateFactor',...
10,'BiasLearnRateFactor',10,"Name",'new_conv');
lgraph = replaceLayer(lgraph,'conv10',newConvLayer);

newClassificatonLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassificatonLayer);
```

Train Network

The network requires all input images to have the size 227-by-227-by-3, but each image in the image datastores has a differnet size. Use an augmented image datastore to automatically resize the training images. Specify these additional augmentation operations to be performed on the training images: randomly flip the training images about the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from over-fitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the convolutional layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size to be 11 so that in each epoch you consider all of the data. During training, the software validates the network after every `ValidationFrequency` iterations.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',11, ...
    'MaxEpochs',7, ...
    'InitialLearnRate',2e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'Verbose',false);
```

Train the network that consists of the transferred and new layers.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
classNames = netTransfer.Layers(end).Classes;
save('mySqueezenet.mat','netTransfer');
```

Quantize the Network

Create a `dlquantizer` object and specify the network to quantize.

```
quantObj = dlquantizer(netTransfer, 'ExecutionEnvironment', 'CPU');
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
calResults = quantObj.calibrate(augimdsTrain);
save('squeezenetCalResults.mat','calResults');
save('squeezenetQuantObj.mat','quantObj');
```

Generate PIL MEX Function

In this example, you generate code for the entry-point function `predict_int8`. This function uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. Then the entry-point function predicts the responses by using the `predict` (Deep Learning Toolbox) function.

```
type predict_int8.m
```

```
function out = predict_int8(netFile, in)

    persistent mynet;
    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork(netFile);
    end
    out = predict(mynet,in);
end
```

To generate a PIL MEX function, create a code configuration object for a static library and set the verification mode to `'PIL'`. Set the target language to C++.

```
cfg = coder.config('lib', 'ecoder', true);
cfg.VerificationMode = 'PIL';
cfg.TargetLang = 'C++';
```

Create a deep learning configuration object for the ARM Compute library and specify the library version. For this example, suppose that the ARM Compute Library in the Raspberry Pi hardware is version 20.02.1.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '20.02.1';
```

Set the properties of `dlcfg` to generate code for low precision/INT8 inference.

```
dlcfg.CalibrationResultFile = 'squeezenetQuantObj.mat';
dlcfg.DataType = 'int8';
```

6. Set the `DeepLearningConfig` property of `cfg` to `dlcfg`.

```
cfg.DeepLearningConfig = dlcfg;
```

7. Use the MATLAB Support Package for Raspberry Pi function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiname` with the name of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
%  r = raspi('raspiname','username','password');
```

8. Create a `coder.Hardware` object for Raspberry Pi and attach it to the code generation configuration object.

```
% hw = coder.hardware('Raspberry Pi');
% cfg.Hardware = hw;
```

9. Generate a PIL MEX function by using the `codegen` command

```
% codegen -config cfg predict_int8 -args {coder.Constant('mySqueezenet.mat'), ones(227,227,3,'ui
```

Run Generated PIL MEX Function on Raspberry Pi

Input image is expected to be of same size as the input size of the network. Read the image that you want to classify and resize it to the input size of the network. This resizing slightly changes the aspect ratio of the image.

```
% testImage = imread("MerchDataTest.jpg");
% testImage = imresize(testImage,inputSize(1:2));
```

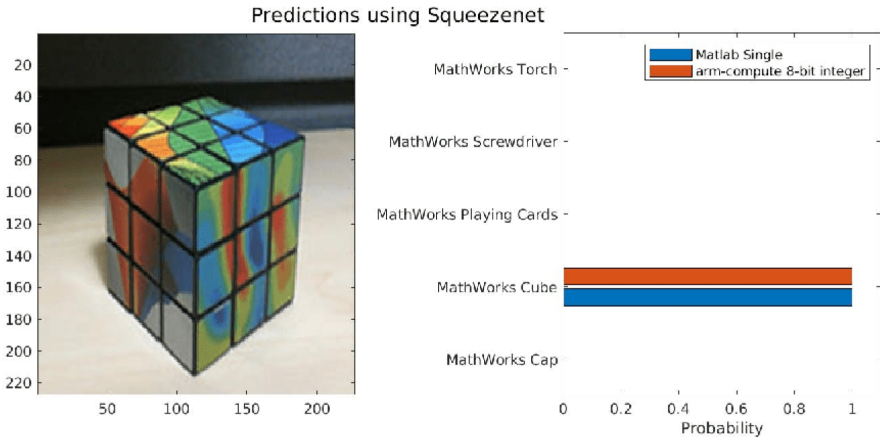
To compare the predictions of the Deep Learning Toolbox `predict` function and the generared PIL MEX function `predict_int8_pil`, call both these funcions on the input image separately.

```
% predictScores(:,1) =  predict(netTransfer,testImage)';
% predictScores(:,2) = predict_int8_pil('mySqueezenet.mat',testImage);
```

Display the predicted labels and their associated probabilities as a histogram.

```
% h = figure;
% h.Position(3) = 2*h.Position(3);
% ax1 = subplot(1,2,1);
% ax2 = subplot(1,2,2);
% image(ax1,testImage);
```

```
% barh(ax2,predictScores)
% xlabel(ax2,'Probability')
% yticklabels(ax2,classNames)
% ax2.XLim = [0 1.1];
% ax2.YAxisLocation = 'left';
% legend('Matlab Single','arm-compute 8-bit integer');
% sgtitle('Predictions using Squeezenet')
% saveas(gcf,'SqueeznetPredictionComparison.jpg');
% close(gcf);
imshow('SqueeznetPredictionComparison.jpg');
```



See Also

Apps
Deep Network Quantizer

Functions
`calibrate` | `codegen` | `coder.loadDeepLearningNetwork` | `dlquantizationOptions` | `dlquantizer` | `validate`

Objects
`coder.ARMNEONConfig`

More About

- "Quantization of Deep Neural Networks" (Deep Learning Toolbox)
- "Code Generation for Deep Learning Networks with ARM Compute Library" on page 38-32

Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning

This example demonstrates how to generate plain C/C++ code that does not depend on any third-party deep learning libraries for a long short-term memory (LSTM) network. You generate a MEX function that accepts time series data representing various sensors in an engine. The MEX function then makes predictions for each step of the input timeseries to predict the remaining useful life (RUL) of the engine measured in cycles.

This example uses the Turbofan Engine Degradation Simulation Data Set as described in [1] and a pretrained LSTM network to predict the remaining useful life of an engine. The network was trained on simulated time series sequence data for 100 engines and corresponding values of the remaining useful life at the end of each sequence. Each sequence in this training data has a different length and corresponds to a full run to failure (RTF) instance. For more information on training the network, see the example Sequence-to-Sequence Regression Using Deep Learning.

Define Entry-Point Function `rulPredict`

The `rulPredict` entry-point function takes an input sequence and passes it to a trained sequence-to-sequence LSTM network for prediction. The function loads the network object from the `rulNetwork.mat` file into a persistent variable and reuses the persistent object on subsequent prediction calls. The LSTM network makes predictions on the partial sequence one time step at a time. At each time step, the network predicts using the value at this time step, and the network state calculated from the previous time steps only. The network updates its state between each prediction. The `predict` function returns a sequence of these predictions. The last element of the prediction corresponds to the predicted RUL for the partial sequence.

To display an interactive visualization of the network architecture and information about the network layers, use the analyzeNetwork function.

```
type rulPredict.m
```

```
function out = rulPredict(in)
%#codegen

% Copyright 2020 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rulNetwork.mat');
end

% pass in input to predict method
% To prevent the function from adding padding to the data, specify the mini-batch size 1.
out = predict(mynet,in,'MiniBatchSize',1);
```

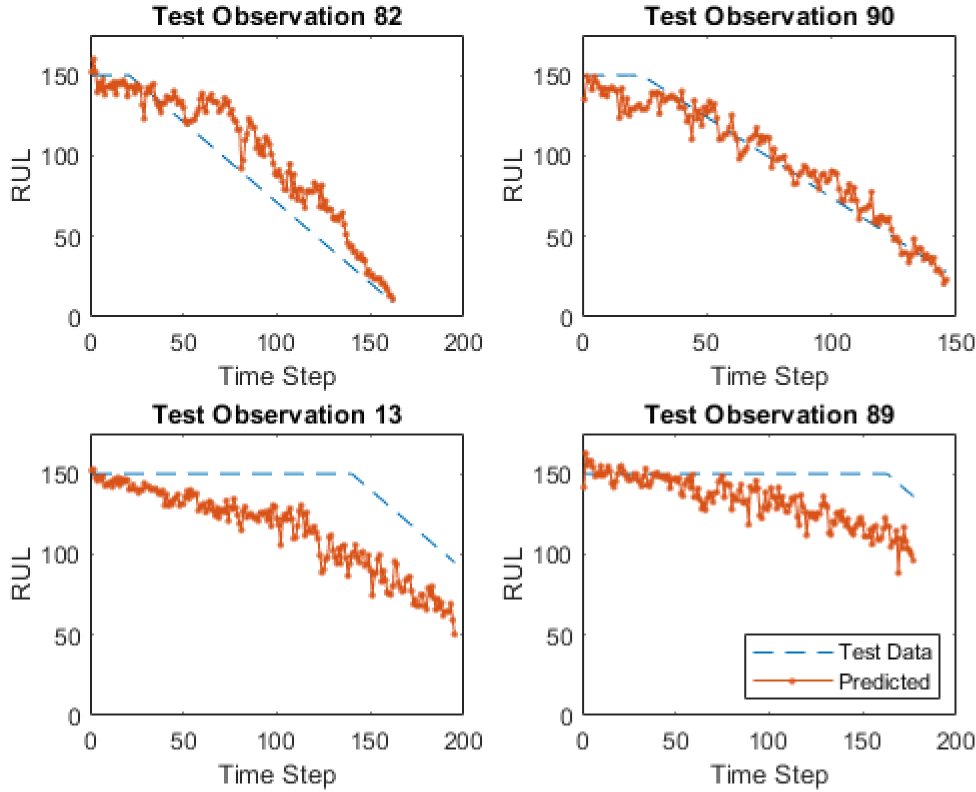
Run `rulPredict` on Test Data

Load the `TurboFanRULValidate` MAT-file. This MAT-file stores the variable `XValidate` that contains sample timeseries data for sensor readings the you use to test the entry-point function in MATLAB. Make predictions on the test data by calling the `rulPredict` method.

```
load TurboFanRULValidate.mat
YPred = rulPredict(XValidate);
```

Visualize some of the predictions in a plot.

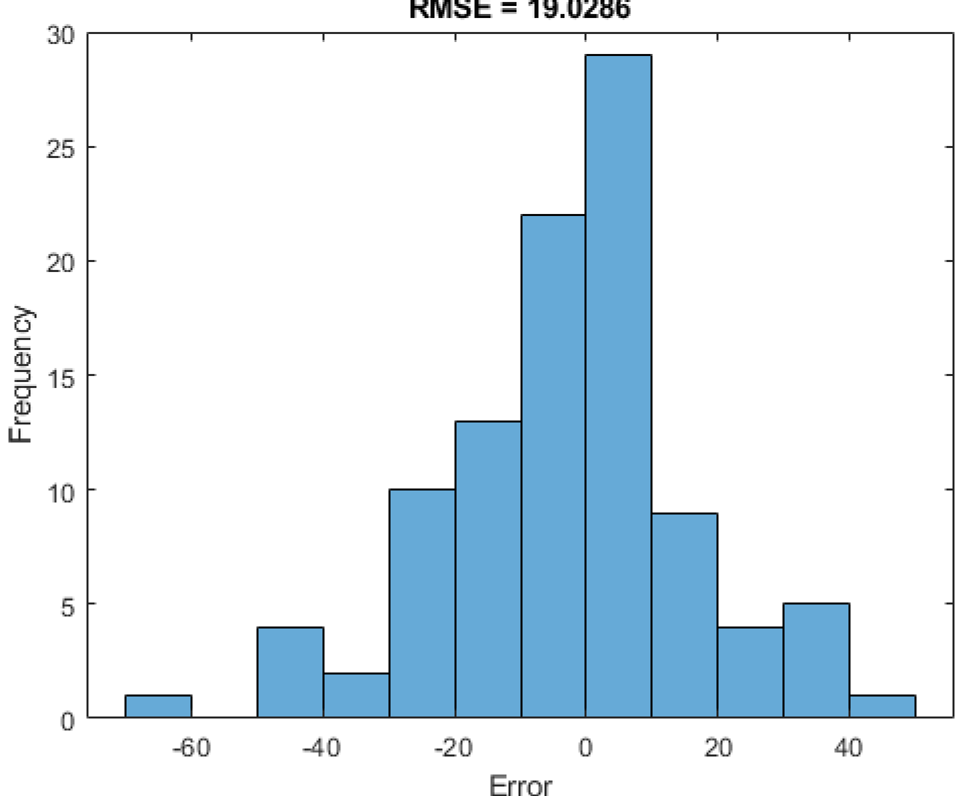
```
idx = randperm(numel(YPred),4);
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YValidate{idx(i)},'--')
    hold on
    plot(YPred{idx(i)},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted"],'Location','southeast')
```



For a given partial sequence, the predicted current RUL is the last element of the predicted sequences. Calculate the root-mean-square error (RMSE) of the predictions, and visualize the prediction error in a histogram.

```
YValidateLast = zeros(1, numel(YValidate));
YPredLast = zeros(1, numel(YValidate));
for i = 1:numel(YValidate)
    YValidateLast(i) = YValidate{i}(end);
    YPredLast(i) = YPred{i}(end);
```

```
end
figure
rmse = sqrt(mean((YPredLast - YValidateLast).^2))

rmse = 19.0286

histogram(YPredLast - YValidateLast)
title("RMSE = " + rmse)
ylabel("Frequency")
xlabel("Error")
```



Generate MEX function for `rulPredict`

To generate a MEX function for the `rulPredict` entry-point function, create a code generation configuration object `cfg` for MEX code generation. Create a deep learning configuration object that specifies that no target library is required and attach this deep learning configuration object to `cfg`.

```
cfg = coder.config('mex');
cfg.DeepLearningConfig = coder.DeepLearningConfig('TargetLibrary','none');
```

By default, the target language is set to C. If you want to generate C++ code, explicitly set the target language to C++.

Use the `coder.typeof` function to create the input type for the entry-point function `rulPredict` that you use with the `-args` option in the `codegen` command.

The data `XValidate` contains 100 observations where each observation is of double data type with a feature dimension value of 17 and a variable sequence length. In order to perform prediction on

several such observations in a single function call, you can group the observations together in a cell array and pass the cell array for prediction. The cell array must be a column cell array, and each cell must contain one observation. Each observation must have the same feature dimension, but the sequence lengths might vary as is the case for `XValidate`. Specifying the sequence length as variable-size enables us to perform prediction on an input sequence of any length.

```
matrixInput = coder.typeof(0, [17 Inf],[false true]); % input type for a single observation
cellInput = coder.typeof({matrixInput}, [100 1]); % input type for multiple observations
```

Run the codegen command. Specify the input type to be `cellInput`.

```
codegen -config cfg rulPredict -args {cellInput} -report
```

```
Code generation successful: To view the report, open('codegen\mex\rulPredict\html\report.mldatx')
```

By default for MEX code generation, the generated code calls into BLAS library for matrix operations and uses OpenMP library (if the compiler supports OpenMP) so that the any parallelizable for loops in the MEX can run on multiple threads leading to better execution performance. While OpenMP is enabled by default for standalone code generation, you will have to provide a custom BLAS callback to indicate to MATLAB Coder ™ that you want to generate BLAS calls for matrix operations following the steps mentioned in Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls.

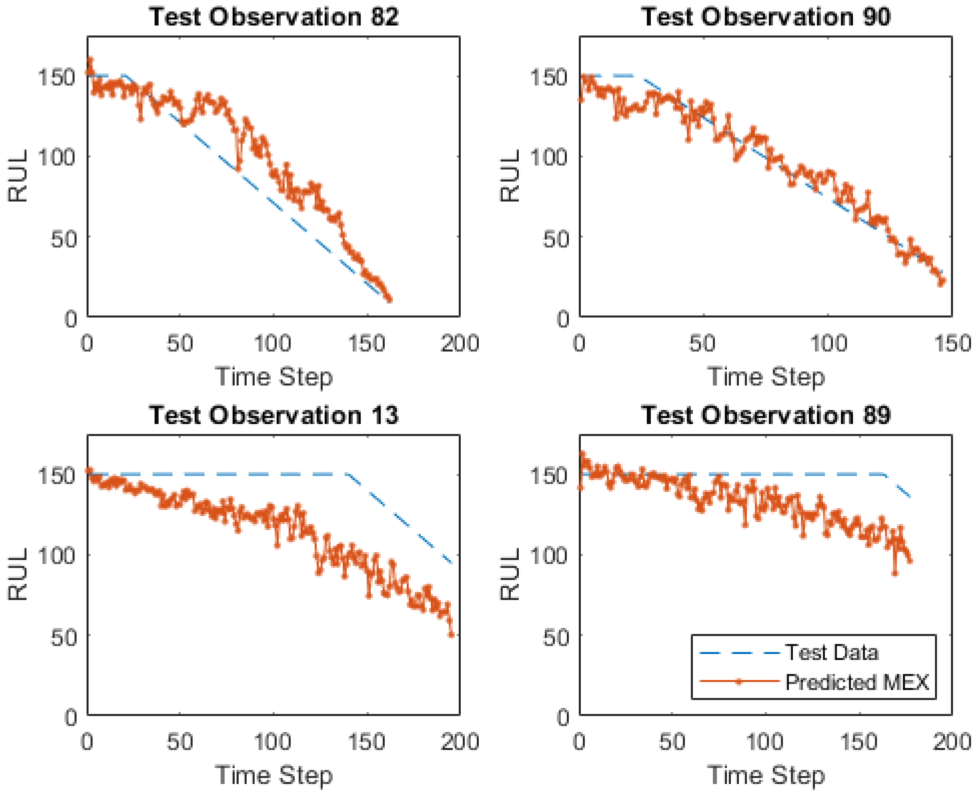
Run Generated MEX Function on Test Data

Make predictions on the test data by calling the generated MEX function `rulPredict_mex`.

```
YPredMex = rulPredict_mex(XValidate);
```

You can visualize the same predictions as before in a plot.

```
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YValidate{idx(i)},'--')
    hold on
    plot(YPredMex{idx(i)},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted MEX"],'Location','southeast')
```

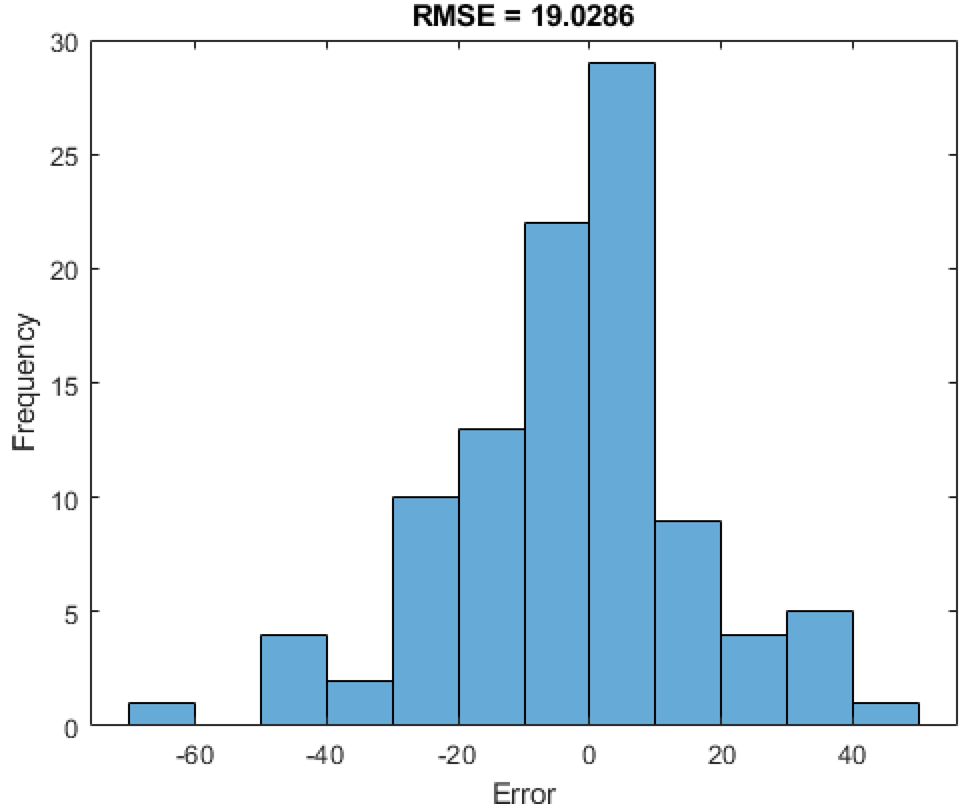


Calculate the root-mean-square error (RMSE) of the predictions, and visualize the prediction error in a histogram.

```
YPredLastMex = zeros(1, numel(YValidate));
for i = 1:numel(YValidate)
    YPredLastMex(i) = YPredMex{i}(end);
end
figure
rmse = sqrt(mean((YPredLastMex - YValidateLast).^2))
```

```
rmse = 19.0286
```

```
histogram(YPredLastMex - YValidateLast)
title("RMSE = " + rmse)
ylabel("Frequency")
xlabel("Error")
```



Generate MEX function with Stateful LSTM

Instead of passing the entire timeseries to predict in one step, you can make predictions one time step at a time by using predictAndUpdateState. This is useful when you have the values of the time steps arriving in a stream. The `predictAndUpdateState` function takes in an input, produces an output prediction, and updates the internal state of the network so that future predictions take this initial input into account. Usually, it is faster to make predictions on full sequences when compared to making predictions one time step at a time.

The entry-point function `rulPredictAndUpdate` takes in a single-timestep input and processes the input using the `predictAndUpdateState` function. `predictAndUpdateState` outputs a prediction for the input timestep and updates the network so that subsequent inputs are treated as subsequent timesteps of the same sample. After passing in all timesteps one at a time, the resulting output is the same as if all timesteps were passed in as a single input.

```
type rulPredictAndUpdate.m
```

```
function out = rulPredictAndUpdate(in)
%#codegen

% Copyright 2020 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rulNetwork.mat');
end
```

```
% pass in input to predictAndUpdateState method
[mynet, out] = predictAndUpdateState(mynet, in);
```

Run codegen on this new entry-point function. Since we are taking in a single timestep each call, we specify `matrixInput` to have a fixed sequence dimension of 1 instead of a variable sequence length.

```
matrixInput = coder.typeof(double(0),[17 1]);
codegen -config cfg rulPredictAndUpdate -args {matrixInput} -report
```

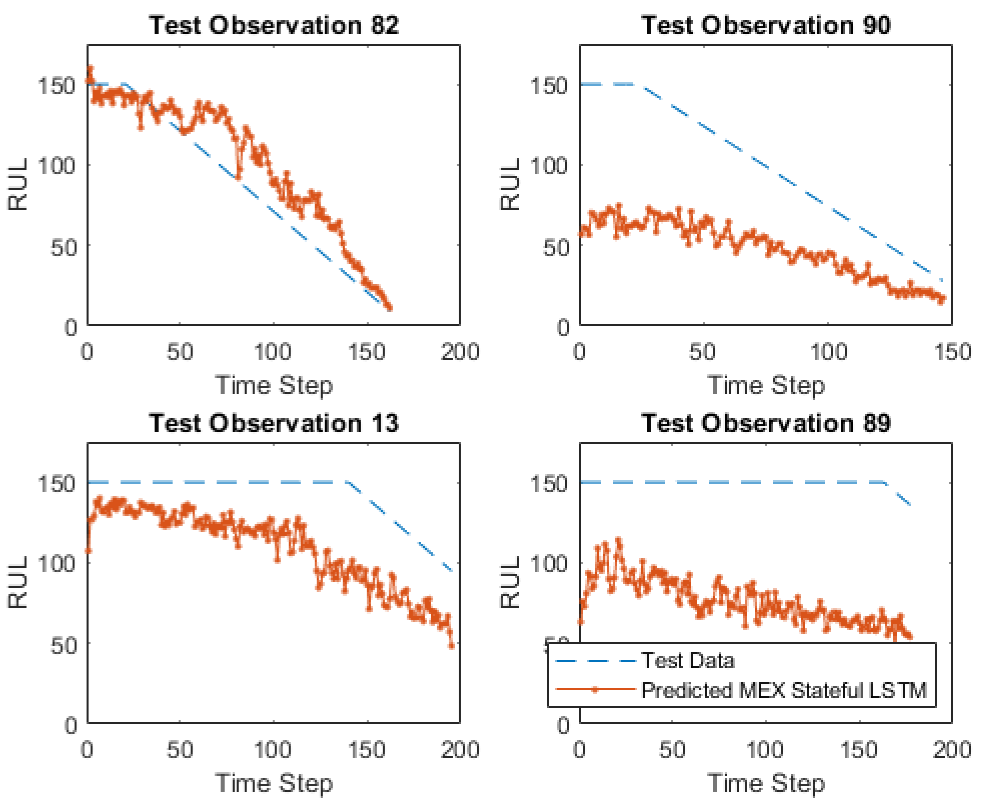
```
Code generation successful: To view the report, open('codegen\mex\rulPredictAndUpdate\html\report
```

Make predictions on the test data by calling the `rulPredictAndUpdate` function in MATLAB and `the` generated MEX function `rulPredictAndUpdate_mex`.

```
YPredStatefulMex = cell(numel(idx), 1);
for iSample = 1:numel(idx)
    sample = XValidate{idx(iSample)};
    numTimeStepsTest = size(sample, 2);
    for iStep = 1:numTimeStepsTest
        YPredStatefulMex{iSample}(1, iStep) = rulPredictAndUpdate_mex(sample(:, iStep));
    end
end
```

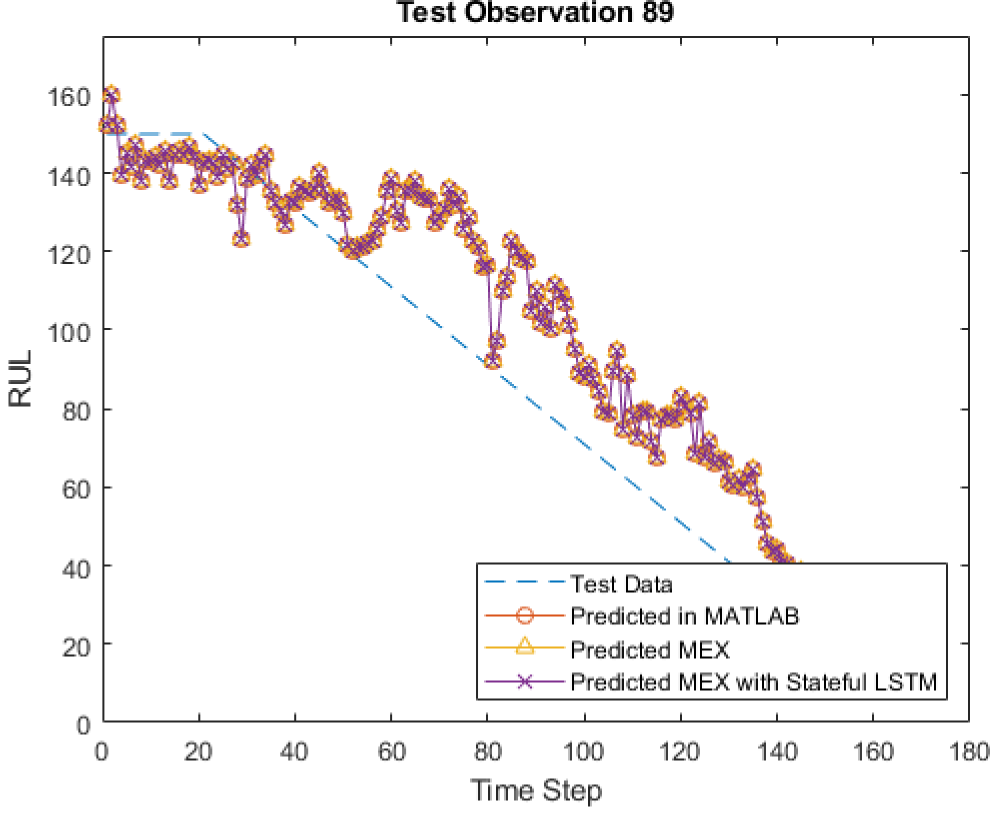
Once again you can visualize the predictions for stateful MEX as before in a plot.

```
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YValidate{idx(i)},'--')
    hold on
    plot(YPredStatefulMex{i},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted MEX Stateful LSTM"],'Location','southeast')
```



Finally you can also visualize the results for the two different MEX functions along with the MATLAB prediction in a plot for any particular sample.

```
figure()
sampleIdx = idx(1);
plot(YValidate{sampleIdx},'--')
hold on
plot(YPred{sampleIdx},'o-')
plot(YPredMex{sampleIdx},'^-')
plot(YPredStatefulMex{1},'x-')
hold off

ylim([0 175])
title("Test Observation " + idx(i))
xlabel("Time Step")
ylabel("RUL")
legend(["Test Data" "Predicted in MATLAB" "Predicted MEX" "Predicted MEX with Stateful LSTM"],'Lo
```



References

1 Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pp. 1-9. IEEE, 2008.

See Also

`codegen` | `coder.DeepLearningConfig` | `coder.config`

More About

- "Generate Generic C/C++ Code for Deep Learning Networks" on page 38-26

Generate Digit Images Using Variational Autoencoder on Intel CPUs

This example shows how to generate a MEX function for a trained variational autoencoder (VAE) network that runs on Intel® CPUs. The example illustrates:

- Generation of hand-drawn digit images in the style of the MNIST data set.
- Code generation for a `dlnetwork` (Deep Learning Toolbox) object representing a deep learning network using the Intel MKL-DNN library.
- Use of `dlarray` (Deep Learning Toolbox) objects in code generation.

This example uses a pretrained decoder network based on the *Train Variational Autoencoder (VAE) to Generate Images* example from the Deep Learning Toolbox™. For more information, see "Train Variational Autoencoder (VAE) to Generate Images" (Deep Learning Toolbox).

Third-Party Prerequisites

Required

- Intel processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- For information on the supported versions of the compilers and libraries, see "Prerequisites for Deep Learning with MATLAB Coder" on page 38-2

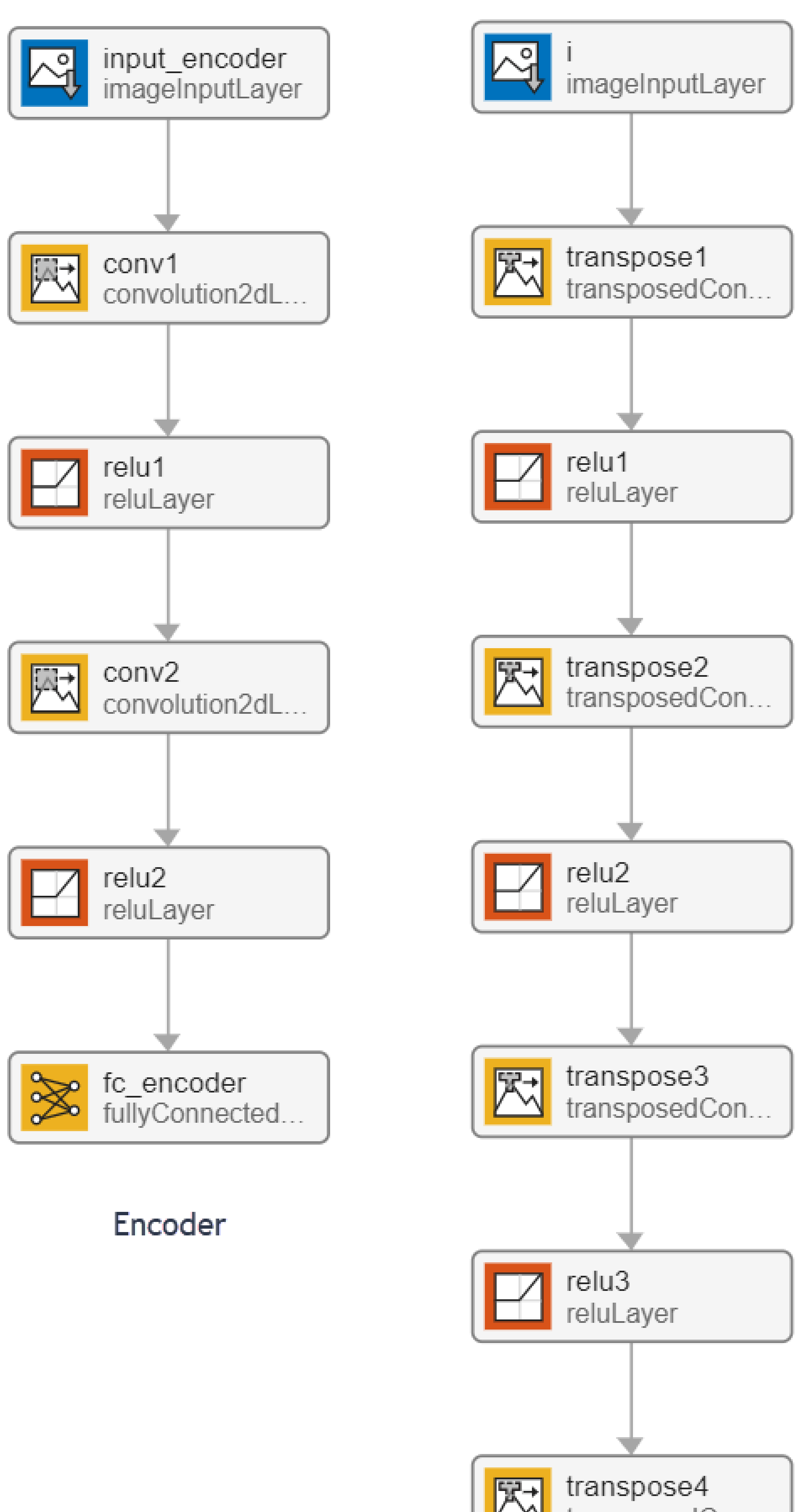
This example is not supported in MATLAB® Online.

Pretrained Variational Autoencoder Network

Autoencoders have two parts: the encoder and the decoder. The encoder takes an image input and outputs a compressed representation (the encoding), which is a vector of size `latent_dim`, equal to 20 in this example. The decoder takes the compressed representation, decodes it, and recreates the original image.

VAEs differ from regular autoencoders in that they do not use the encoding-decoding process to reconstruct an input. Instead, they impose a probability distribution on the latent space, and learn the distribution so that the distribution of outputs from the decoder matches that of the observed data. Then, they sample from this distribution to generate new data.

This example uses the decoder network trained in the *Train Variational Autoencoder (VAE) to Generate Images* example. To train the network yourself, see "Train Variational Autoencoder (VAE) to Generate Images" (Deep Learning Toolbox).


input_encoder
imageInputLayer
conv1
convolution2dL...
relu1
reluLayer
conv2
convolution2dL...
relu2
reluLayer
fc_encoder
fullyConnected...
Encoder
i
imageInputLayer
transpose1
transposedCon...
relu1
reluLayer
transpose2
transposedCon...
relu2
reluLayer
transpose3
transposedCon...
relu3
reluLayer
transpose4
transposedCon...

The generateVAE Entry-Point Function

The `generateVAE` entry-point function loads the `dlnetwork` object from the trainedDecoderVAENet MAT-file into a persistent variable and reuses the persistent object for subsequent prediction calls. It initializes a `dlarray` object containing 25 randomly generated encodings, passes them through the decoder network, and extracts the numeric data of the generated image from the deep learning array object.

```
type('generateVAE.m')
```

```
function generatedImage =  generateVAE(decoderNetFileName,latentDim,Environment) %#codegen
% Copyright 2020-2021 The MathWorks, Inc.

persistent decoderNet;
if isempty(decoderNet)
    decoderNet = coder.loadDeepLearningNetwork(decoderNetFileName);
end

% Generate random noise
randomNoise = dlarray(randn(1,1,latentDim,25,'single'),'SSCB');

if coder.target('MATLAB') && strcmp(Environment,'gpu')
    randomNoise = gpuArray(randomNoise);
end

% Generate new image from noise
generatedImage = sigmoid(predict(decoderNet,randomNoise));

% Extract numeric data from dlarray
generatedImage = extractdata(generatedImage);

end
```

Evaluate the Entry-Point Function

Evaluate the `generateVAE` entry-point function to generate digit images and plot the results.

```
latentDim = 20;
matfile = 'trainedDecoderVAENet.mat';
Env = '';

figure()
title("Generated samples of digits - MATLAB")

generatedImageML = generateVAE(matfile, latentDim, Env);
imshow(imtile(generatedImageML, "ThumbnailSize", [100,100]))
```

Generate MEX Function

To generate a MEX function for the `generateVAE` entry-point function, create a code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a MKL-DNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the code configuration object.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');

args = {coder.Constant(matfile), coder.Constant(latentDim), coder.Constant(Env)};

codegen -config cfg -args args generateVAE -report
```

```
Code generation successful: View report
```

Run the Generated MEX

Call the generated MEX and display the results.

```
figure()
title("Generated samples of digits using MKL-DNN")

generatedImage = generateVAE_mex(matfile, latentDim, Env);
imshow(imtile(generatedImage, "ThumbnailSize", [100,100]))
```

See Also

codegen | coder.DeepLearningConfig | coder.config | dlarray | dlnetwork

Related Examples

- "Train Variational Autoencoder (VAE) to Generate Images" (Deep Learning Toolbox)

More About

- "dlarray Limitations for Code Generation" on page 18-8
- "Code Generation for dlarray" on page 18-2

39

Generating Code for C++

- “C++ Code Generation” on page 39-2
- “Generate C++ Code with Class Interface” on page 39-4
- “Organize Generated C++ Code into Namespaces” on page 39-9
- “Integrate Multiple Generated C++ Code Projects” on page 39-14
- “Generate C++ Classes for MATLAB® Classes That Model Simple and Damped Oscillators” on page 39-18

C++ Code Generation

MATLAB Coder enables you to either generate C or C++ code. The code generator produces C code by default. Generated C++ code can use functionality not available in the C language that can make the C++ code more readable and easier to use.

Generate C++ Code

To generate C++ code, follow the same overall workflow steps that you use to generate C code. For example, see “Generate C Code at the Command Line”. Select the C++ language option from the command line, or with a code generation configuration setting, or from the MATLAB Coder app.

Suppose that you want to generate C++ code for a function `foo` that accepts zero inputs:

- From the command line, use the `-lang:c++` specifier. This specifier provides a quick and easy way to generate C++ code. For example, to generate a C++ static library and C++ source code for `foo`, enter:

  ```
  codegen -config:lib -lang:c++ foo
  ```
- In the configuration object, set the `TargetLang` parameter to C++. For example, to generate a C ++ dynamic library, enter:

  ```
  cfg = coder.config('dll');
  cfg.TargetLang = 'C++';
  codegen -config cfg foo
  ```
- From the app, at the **Generate Code** step, select the **C++** language button.

C++ Language Features Supported in Generated Code

To learn about code generation that utilizes key C++ language features, refer to these help topics:

Goal	More Information
Generate C++ classes for classes in your MATLAB code.	“Generate C++ Classes for MATLAB Classes” on page 16-2
Generate entry-point functions as methods in a C ++ class.	“Generate C++ Code with a Class Interface” on page 39-4
Generate C++ namespaces for MATLAB packages. Place all generated code in a namespace that you specify. Place all code generated for MathWorks code in a namespace that you specify.	“Organize Generated C++ Code into Namespaces” on page 39-9
Pass dynamically allocated arrays between your custom C++ code and the generated code. The generated C++ code implements such arrays by using the `coder::array` class template. The generated code provides a simple API that you can use to interact with this template.	“Use Dynamically Allocated C++ Arrays in Generated Function Interfaces” on page 31-15

These examples illustrate the use of these functionalities:

- "Generate C++ Classes for MATLAB® Classes That Model Simple and Damped Oscillators" on page 39-18
- "Integrate Multiple Generated C++ Code Projects" on page 39-14

Additional Differences Between Generated C Code and C++ Code

If you separately generate C and C++ code for the same MATLAB function, and inspect the generated source code, then there are implementation differences. These are some notable differences:

- The generated C++ code contains overloaded functions or methods that have the same name but support multiple signatures. The C language does not support overloading of functions.
- The generated C++ code reuses the same identifier name across different namespace hierarchies. For example, the same type name `myType` can appear in two different namespaces hierarchies with top-level namespaces `myNamespace_1` and `myNamespace_2`. The C language does not support namespaces and such reuse of identifier names.
- In generated C code, the function headers contain `#ifdef __cplusplus` include guards that specify the `extern "C"` identifier for the generated C functions. The compiler and linker use these identifiers in building C code as part of a C++ project.
- Generated C++ code uses `.cpp` file extensions for the C++ files and `.h` extensions for the header files. Generated C code uses `.c` and `.h` extensions.
- The generated C++ code uses some C++ casts, like `static_cast`, which are more explicit than the casting syntax in C.
- The generated code defines values for `Inf` and `NaN` based on different mechanisms for C++ and C.
- Generated C++ code uses the custom data types as described in "Mapping MATLAB Types to Types in Generated Code" on page 33-15.
- Generated C++ code uses different libraries than generated C code. For example, the default standard math library for C++ and C is described in "Change the Standard Math Library" on page 27-29.

See Also

`codegen`

More About

- "Configure Build Settings" on page 27-13

Generate C++ Code with Class Interface

When you generate C code, the software analyzes your MATLAB code and generates entry-point C functions corresponding to your entry-point MATLAB functions. When you generate C++ code, you can choose to generate entry-point functions as methods in a C++ class. Using this option:

- You obtain more object-oriented code.
- The code generator produces a class constructor and destructor that automatically perform memory initialization and termination.
- You allocate memory for each class instance separately. The methods for each class instance are thread-safe and reentrant.
- Multiple entry-point functions become methods in a single C++ class.

You can generate code with a class interface from the command line or from the MATLAB Coder app. From the command line, use the `CppInterfaceStyle` and `CppInterfaceClassName` configuration parameters. From the app, on the **Generate Code** step, select **Language** as **C++**, select **Interface style** as **Methods**, and then specify the **C++ interface class name**.

These examples show the command-line workflow.

Generate C++ Code with a Class Interface

This example shows how the generated C++ code differs when it uses a class interface.

MATLAB Algorithm

Consider a simple MATLAB function that performs operations on a matrix and outputs the result.

```
function out = foog %#codegen
I = eye(447);
out = ones(447)*I + 7;
```

Generate C++ Code With and Without Class Interface

To generate C++ code with a class interface, use the `CppInterfaceStyle` and `CppInterfaceClassName` parameters. Store the output in the `withClass` folder.

```
cfg = coder.config('lib');
cfg.GenCodeOnly = true;
cfg.TargetLang = 'C++';
cfg.CppInterfaceStyle = 'Methods';
cfg.CppInterfaceClassName = 'myClass';
codegen foog -config cfg -report -d withClass
```

```
Code generation successful: To view the report, open('withClass\html\report.mldatx').
```

Next, create a new configuration object and generate C++ code that does not use a class interface.

```
cfg = coder.config('lib');
cfg.GenCodeOnly = true;
cfg.TargetLang = "C++";
codegen foog -config cfg -report -d withoutClass
```

```
Code generation successful: To view the report, open('withoutClass\html\report.mldatx').
```

Inspect the generated example main function. Compare the versions with and without the class interface. With the class interface, the main function calls the entry-point function as a class method.

```
type withClass/examples/main.cpp
```

Class Definition and Implementation in the Generated Code

When the code generator produces code for the C++ interface class, it ensures that the function methods are reentrant. If the function methods use variables that can exceed the local stack memory limit, set by the configuration parameter `StackUsageMax`, then the code generator produces private data structures for the variables (identifiable by the suffix `StackData`), rather than declaring the variables as `static`. Static variables persist between function calls and are not reentrant. For information on generating reentrant C code, see “Generating and Calling Reentrant Code”.

To explore the generated class implementations, modify the function `foog` such that it contains a variable that exceeds the maximum stack usage specified by the configuration parameter `StackUsageMax`.

```
function out = foogBig %#codegen
I = eye(448);
out = ones(448)*I + 7;
```

The default value for `StackUsageMax` in bytes is:

```
cfg.StackUsageMax
```

```
ans =

  int32

   200000
```

Because `fooBig` uses a variable of 448^2 (200704) elements, and the code generator produces an 8-bit integer array to represent the variable, the default stack usage limit is exceeded by 704 bytes. Generate code for `foogBig`.

```
cfg = coder.config('lib','ecoder',false);
cfg.GenCodeOnly = true;
cfg.TargetLang = 'C++';
cfg.CppInterfaceStyle = 'Methods';
cfg.CppInterfaceClassName = 'myBigClass';
codegen foogBig -config cfg -report -d withBigClass
```

```
Code generation successful: To view the report, open('withBigClass\html\report.mldatx').
```

Inspect the Generated Interface Class Definitions

Inspect the class definitions for the `foogBig` project and for `foog`. The `foogBig` class stores variables that can exceed the maximum stack usage in a private class property, whereas the `foog` class only creates local variables on the stack.

When you work with a class definition that contains a `StackData` structure, indicating that the class requires data that exceeds the local stack usage limit, then allocate heap memory for the class instance by using `new`. See the generated example main file for your generated code for an example.

Globals and Persistents in a Generated C++ Class

When you generate C++ code with a class interface, then you access globals and persistents as members of the class. This example shows how to interact with globals and persistents in the class.

MATLAB Algorithm

Consider a MATLAB function that keeps count of the number of times you call it with a global and persistent variable.

```
function [po,go] = countCalls %#codegen
% increment persistent & global variable
persistent p
global g
if isempty(p)
    p = 0;
end
p = p+1;
g = g+1;
% set output variables
po = double(p);
go = double(g);
```

Generate C++ Code with a Class Interface

For code generation, initialize the global variable in the workspace.

```
global g;
g = 0;
```

Generate code in the class called `countClass`.

```
cfg = coder.config('lib');
cfg.GenCodeOnly = true;
cfg.TargetLang = 'C++';
cfg.CppInterfaceStyle = 'Methods';
cfg.CppInterfaceClassName = "countClass";
codegen countCalls -config cfg -report
```

```
Code generation successful: To view the report, open('codegen\lib\countCalls\html\report.mldatx')
```

Inspect the Class Definition

In the generated C++ code, an initialization function sets the global variable to the value that you specify in the workspace. You can also specify the initial global value with the `codegen -globals` syntax.

Inspect the code for the class definition in the header file `countClass.h`.

```
type codegen/lib/countCalls/countClass.h
```

The global variable is a public member of the class. Access this variable from your main function as needed. The persistent variable is stored in a private class data structure.

Put Multiple Entry-Point Functions in the Same Class

When you generate C++ code for multiple entry-point functions and use the class interface setting, then each function becomes a public method of the same class. You can use this technique to create a simpler interface to your multiple entry-point function project.

MATLAB Entry-Point Functions

Break the function `countCalls` in the previous example into two, so that one function counts the calls with a persistent variable and the other counts the calls with a global variable. Inspect the two functions.

```
function po = countPersistent %#codegen
% increment persistent variable
persistent p
if isempty(p)
    p = 0;
end
p = p+1;
% set output variable
po = double(p);
```

```
function go = countGlobal %#codegen
% increment global variable
global g
g = g+1;
% set output variable
go = double(g);
```

Generate C++ Code

Use the `codegen` command and specify the initial global variable value as an input.

```
cfg = coder.config('lib');
cfg.GenCodeOnly = true;
cfg.TargetLang = 'C++';
cfg.CppInterfaceStyle = 'Methods';
cfg.CppInterfaceClassName = 'countClassMulti';
codegen countGlobal countPersistent -config cfg -report -globals {'g',0}
```

```
Code generation successful: To view the report, open('codegen\lib\countGlobal\html\report.mldatx
```

Inspect the Generated Code

To see the generated class definition, open `countClassMulti.h`. Each entry-point function is a public method of the class.

```
type codegen/lib/countGlobal/countClassMulti.h
```

See Also

codegen

More About

- “Generate Code for Multiple Entry-Point Functions” on page 27-78
- “Generating and Calling Reentrant Code”
- “Generate Code for Global Data” on page 27-88

Organize Generated C++ Code into Namespaces

Namespaces help organize your code into logical parts, prevent name collisions, and enable you to more easily integrate your generated C++ code into a larger C++ project. Namespaces also increase compliance with the MISRA C++ standards for safety-critical code. This topic explains how to use the code generation settings to customize the organization of your generated C++ code into namespaces.

Settings That Control Namespace Structure

These are the code generation settings that enable you to control the creation of namespaces in the generated code:

Code Configuration Parameter	Description	How to specify
In a code configuration object: `CppNamespace` In the MATLAB Coder app: On the **Code Appearance** tab, **C++ Namespace**	Namespace that contains the generated C++ code. If this parameter is empty, the code generator does not create such a namespace.	In a code configuration object: `''` (default)| character vector In the MATLAB Coder app: specify in a text field
In a code configuration object: `CppNamespaceForMathworksCode` In the MATLAB Coder app: On the **Code Appearance** tab, **Namespace for MathWorks functions**	Namespace that contains code generated for all MathWorks code (for example, code for the sparse data type). If this parameter is empty, the code generator does not create such a namespace.	In a code configuration object: `'coder'` (default)| character vector In the MATLAB Coder app: specify in a text field
In a code configuration object: `CppPackagesToNamespaces` In the MATLAB Coder app: On the **Code Appearance** tab, **MATLAB package to C++ namespace**	Whether to generate C++ namespaces for the packages in the MATLAB code.	In a code configuration object: `true` (default)| `false` In the MATLAB Coder app: On the **Code Appearance** tab, select or clear the **MATLAB package to C++ namespace** check box

Additional notes about namespace generation:

- When you specify the `CppNamespace` property (or the corresponding setting in the app), the code generator packages all the generated functions and type definitions into the namespace, except for the generic type definitions contained in `tmwtypes.h` and the hardware-specific definitions in `rtwtypes.h`. The example main file and function are not packaged into the namespace.
- If your MATLAB code has nested packages (for example, `pkg1` inside `pkg2`), the generated namespaces have the same nesting.
- When creating packages for your MATLAB code that is intended for code generation, follow these guidelines:
  - Do not create a package that has the name `'coder'`.
  - If you set the `CppNamespaceForMathworksCode` property (or the equivalent parameter in the app) to a nondefault name, do not create a package that has that name.

Example: Generate C++ Code with Namespaces

This example shows how to use these code generation settings to create namespaces.

Define MATLAB® Functions

Define two MATLAB functions `foo` and `bar` in two separate files `foo.m` and `bar.m`. Place the file `bar.m` in a package `myPackage`. The function `foo` accepts a string input and then calls the function `bar`.

```
type foo.m
```

```
function out = foo(str)
temp = strlength(str);
out = mypackage.bar(temp);
end
```

```
type +mypackage/bar.m
```

```
function out = bar(in)
coder.inline('never');
out = in + 1;
end
```

Define Code Configuration Object

Create a code configuration object for a static library. Set the target language to C++. Specify the namespace that contains all generated code to be named `allcode`. Specify the namespace that contains MathWorks® code to be named `notmycode`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.CppNamespace = 'allcode';
cfg.CppNamespaceForMathworksCode = 'notmycode';
```

Generate Code

Generate a static C++ library and a code generation report. Specify the input type to be string scalar that can have any length.

```
s = "mystring";
t = coder.typeof(s);
t.Properties.Value = coder.typeof('a',[1 inf]);
```

```
codegen -config cfg foo -args {t} -report
```

```
Code generation successful: To view the report, open('codegen\lib\foo\html\report.mldatx').
```

Inspect Generated Code

Open the code generation report and inspect the generated code.

The file `foo.h` contains the declaration of the generated function `foo`. Because the MATLAB function `foo` that you created is not inside a package, the generated function is declared only inside the namespace `allcode` that contains all generated code.

```
type codegen/lib/foo/foo.h
```

```
//
// File: foo.h
//
// MATLAB Coder version            : 5.2
// C/C++ source code generated on  : 23-Feb-2021 16:47:53
//

#ifndef FOO_H
#define FOO_H

// Include Files
#include "rtwtypes.h"
#include "string1.h"
#include <cstddef>
#include <cstdlib>

// Function Declarations
namespace allcode {
extern double foo(const notmycode::rtString *str);

}

#endif
//
// File trailer for foo.h
//
// [EOF]
//
```

The file `bar.h` contains the declaration of the generated function `bar`. Because you created the MATLAB function `bar` inside the package `mypackage`, the generated function is declared inside the namespace hierarchy `allcode::myPackage`.

```
type codegen/lib/foo/bar.h
```

```
//
// File: bar.h
//
// MATLAB Coder version            : 5.2
// C/C++ source code generated on  : 23-Feb-2021 16:47:53
//

#ifndef BAR_H
#define BAR_H

// Include Files
#include "rtwtypes.h"
#include <cstddef>
#include <cstdlib>

// Function Declarations
namespace allcode {
namespace mypackage {
double bar(double in);

}
} // namespace allcode
```

```
#endif
//
// File trailer for bar.h
//
// [EOF]
//
```

The file `string1.h` contains the declaration of the generated class `rtString` that implements the MATLAB string data type. Because you instructed the code generator to place all code produced for MathWorks code inside the namespace `notmycode`, the generated class `rtString` is declared inside the namespace hierarchy `allcode::notmycode`.

```
type codegen/lib/foo/string1.h
```

```
//
// File: string1.h
//
// MATLAB Coder version            : 5.2
// C/C++ source code generated on  : 23-Feb-2021 16:47:53
//

#ifndef STRING1_H
#define STRING1_H

// Include Files
#include "rtwtypes.h"
#include "coder_array.h"
#include <cstddef>
#include <cstdlib>

// Type Definitions
namespace allcode {
namespace notmycode {
class rtString {
public:
  void init(const ::coder::array<char, 2U> &b_Value);
  rtString();
  ~rtString();
  ::coder::array<char, 2U> Value;
};

} // namespace notmycode
} // namespace allcode

#endif
//
// File trailer for string1.h
//
// [EOF]
//
```

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

- “C++ Code Generation” on page 39-2
- “Integrate Multiple Generated C++ Code Projects” on page 39-14
- “Generate C++ Classes for MATLAB® Classes That Model Simple and Damped Oscillators” on page 39-18

Integrate Multiple Generated C++ Code Projects

This example shows how to integrate two different generated C++ code projects into a single, larger project.

Your generated code projects might have similar function names, but have different settings, parameters, or functionality. Generate code with namespaces to aid in integrating different projects that share the same names. Namespaces can also improve code readability.

Generate C++ Code for a MATLAB® Algorithm

Consider a simple MATLAB function that returns a gravitational constant. The value of the gravitational constant is derived from a global variable.

```
type getGravityConst.m
```

```
function c = getGravityConst %#codegen
global g
c = g;
```

Suppose that you want to generate code for `getGravityConst` that models scenarios for the Moon and for the Earth. Generate two separate code projects with the same entry-point function. Specify a different global value, and hence, gravitational constant, for each project.

Create a code generation configuration object. Specify:

- DLL build type.
- C++ target language.
- The name of the orbital body as the namespace.
- `#pragma once` style `#include` guards.
- Packaging of the generated code files into a `.zip` file by calling the `packNGo` function.

```
cfg = coder.config('dll');
cfg.TargetLang = "C++";
cfg.CppNamespace = 'moon';
cfg.HeaderGuardStyle = "UsePragmaOnce";
cfg.PostCodeGenCommand = 'packNGo(buildInfo)';
```

Generate code for `getGravityConst` to model the Moon:

- By using the previously defined configuration object.
- With a code generation report.
- Such that the code returns the Moon's value of the gravitational constant in units of m/s^2.
- In an output folder called `projectMoon`.
- With output binaries called `getGravityConstMoon`.

```
codegen getGravityConst -config cfg -report -globals {'g', -1.62} ...
    -d projectMoon -o getGravityConstMoon
```

```
Code generation successful: To view the report, open('projectMoon\html\report.mldatx').
```

To generate code for `getGravityConst` that models the earth, first modify the:

- Namespace name
- Gravitational constant
- Output file name
- Output folder name

```
cfg = coder.config('dll');
cfg.TargetLang = "C++";
cfg.CppNamespace = 'earth';
cfg.HeaderGuardStyle = "UsePragmaOnce";
cfg.PostCodeGenCommand = 'packNGo(buildInfo)';

codegen getGravityConst -config cfg -report -globals {'g', -9.81} ...
    -d projectEarth -o getGravityConstEarth
```

```
Code generation successful: To view the report, open('projectEarth\html\report.mldatx').
```

Project Integration Scenario: Planetary Modeling

Suppose that you want to design a larger project that performs planetary modeling and computes quantities such as the flight times of falling objects. The flight time depends on the gravitational constant for each planet and the initial height of the object. You want to use the generated code functions for `getGravityConst` in this larger project.

Determine the Platform-Dependent File Extensions

The generated dynamic libraries have different extensions on different platforms. This code determines the correct extensions for your platform.

```
dllext = '';
libext = '';
if ismac
    dllext = '.dylib';
    libext = dllext;
elseif isunix
    dllext = '.so';
    libext = dllext;
elseif ispc
    dllext = '.dll';
    libext = '.lib';
else
    disp('Platform not supported')
    return
end
```

Write a Main File That Uses the Generated Code Projects

In the general case, you integrate different projects by writing or modifying a main file to call each of the projects' functions. By using namespaces, you can distinguish the generated functions for each project, even though the function names are the same.

For an example of how to write a main file that uses the generated C++ code for both projects, see the attached file `main_planetSim.cpp`. To build an executable or binary from the main file, you must specify or provide the following to the build tools (compiler, linker, and/or IDE) and their correct paths:

- Header files for any called functions.

- On Windows platforms, import libraries (`.lib` files).
- Dynamic libraries (`.dll`, `.so` and `.dylib` files).
- Include directories for other generated source and include files.

The `.zip` files that the `packNGo` command creates during code generation contain the generated code files. Unpack the zip files to folders in your build directory or build environment. You must also make your dynamic libraries accessible to the executable, for example, by moving the generated dynamic libraries to the same folder as the executable.

Write a MATLAB Function that Integrates the Two Projects

As an alternative to writing a main file by hand, you can also integrate two projects into a third generated code project by using the `coder.ceval` function. The `coder.ceval` function enables you to call external C/C++ code from generated C/C++ code.

The file `planetSim.m` shows how to use `coder.ceval` and associated build configuration functions to integrate the generated projects into the larger project.

<include>planetSim.m</include>

Generate MEX code for the `planetSim` function:

```
linkObjectMoon = ['projectMoon/getGravityConstMoon' libext];
linkObjectEarth = ['projectEarth/getGravityConstEarth' libext];

cfg = coder.config('mex');
cfg.TargetLang = "C++";
codegen('planetSim','-config',cfg,'-d','planetSim','-report',linkObjectMoon,linkObjectEarth)
```

```
Code generation successful: To view the report, open('planetSim\html\report.mldatx').
```

Test the Generated MEX Function

Use the MEX function to test the generated code in the MATLAB environment. The MEX function must have access to the generated link libraries. Move the link libraries to the current directory and call the MEX function.

```
copyfile(['projectMoon/getGravityConstMoon' dllext]);
copyfile(['projectEarth/getGravityConstEarth' dllext]);

[t_m, t_e] = planetSim_mex
```

```
t_m = 3.5136

t_e = 1.4278
```

The output shows the flight times for the falling object on the Moon and on the Earth.

See Also

`codegen` | `coder.CodeConfig` | `coder.ceval` | `coder.cinclude` | `coder.config` | `packNGo`

More About

- “C++ Code Generation” on page 39-2

- “Call C/C++ Code from MATLAB Code” on page 33-2
- “Use a Dynamic Library in a Microsoft Visual Studio Project” on page 31-20

Generate C++ Classes for MATLAB® Classes That Model Simple and Damped Oscillators

MATLAB classes provide a natural framework for modelliing physical systems:

- You can model a simple system as a MATLAB class. The private class properties are the system parameters. The class constructor creates an instance of the system with given parameters. A public method captures the dynamics of the system by returning the final state for a given initial state and a time interval. The class can also contain other helper methods that modularize the mathematical analysis.
- You often start your analysis with a simple system and then introduce additional effects (such as mechanical damping) to increase the accuracy of your analysis. In MATLAB, you can model the enhanced system as a subclass that inherits from the original class. The subclass might contain additional private properties for additional system parameters (such as damping constant). Depending on the specifics of the system, the subclass might inherit certain methods from the base class and might overload the other methods.

This example shows how to generate C++ code for a MATLAB function that compares the time evolution of a simple oscillator and a damped oscillator with identical parameters and initial conditions. The two oscillator systems are modelled by using the MATLAB classes `simpleOscillator` and `dampedOscillator` that are defined inside a MATLAB package `mySystem`. The generated code contains C++ classes for the source MATLAB classes. The example also shows how the MATLAB classes map to the generated C++ classes and how to use the generated code in a custom C++ main function.

Simple and Damped Oscillators as MATLAB Classes

Governing Equations

A simple harmonic oscillator has two parameters, the mass m and the spring constant k. The angular frequency of the oscillator is $\omega = \sqrt{\frac{k}{m}}$. The position of the oscillator x as a function of time t is given by:

$$x(t) = A\sin(\omega t + \phi)\,.$$

The amplitude A and the phase constant ϕ are determined by the initial position x_0 and the initial velocity v_0 of the simple oscillator. In this example, the MATLAB class `simpleOscillator` models this system.

A damped harmonic oscillator has one additional parameter, the damping constant b. This example considers the case where the normalized damping parameter $\gamma = \frac{b}{2m}$ is small compared to the angular frequency ω such that only first-order damping effects are significant. The position of the damped oscillator x_d as a function of time t is:

$$x_d(t) = A\,e^{-\gamma t}\sin(\omega t + \phi_d)$$

Like before, the amplitude A and the phase constant ϕ_d are determined by the initial position x_0 and the initial velocity v_0 of the damped oscillator. The main effect of damping is to cause the amplitude to decay exponentially. In this example, the MATLAB class `dampedOscillator` which is a subclass of `simpleOscillator` models the damped system.

MATLAB and C++ Files

This example uses these supporting files that are present in the current working directory:

- The package folder `+mySystem` contains the two class files `simpleOscillator.m` and `dampedOscillator.m`.
- The function `effectOfDamping` calculates and returns the trajectories of a simple oscillator and a damped oscillator with given parameters and initial conditions.
- The C++ header and source files `main_damped_oscillator.h` and `main_damped_oscillator.cpp` implement the custom C++ main function and are used to generate an executable in the last part of the example.

Run MATLAB Code

Define a structure `params` that has fields for the three oscillator parameters. Make sure the `dampingConstant` parameter is small compared to `springConstant` and `mass` (in normalized units).

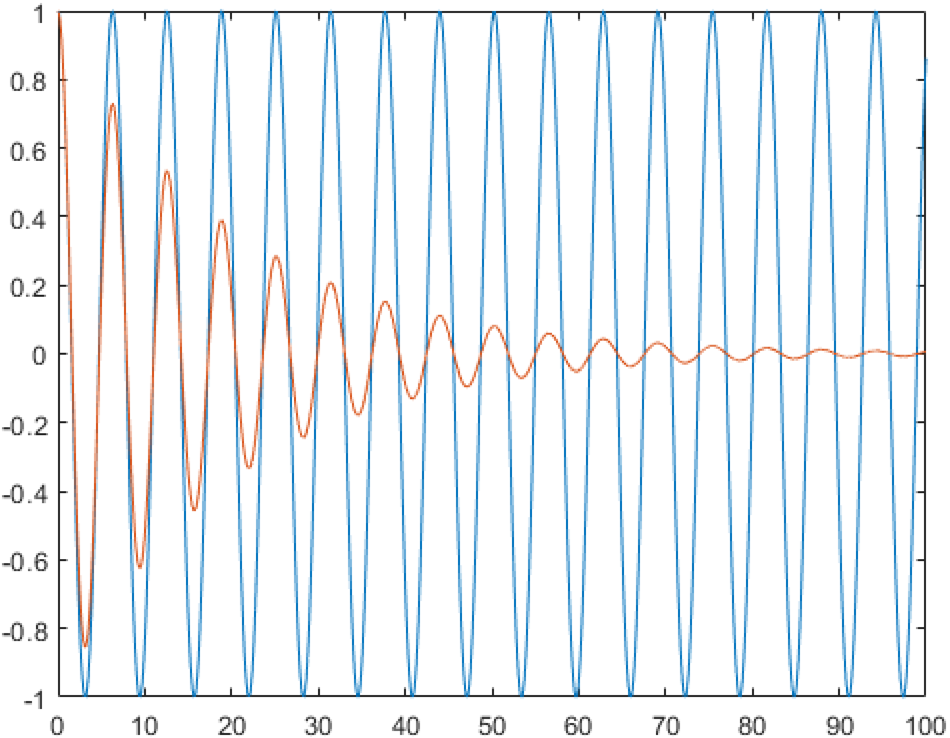
```
params.springConstant = 1;
params.dampingConstant = 0.1;
params.mass = 1;
```

Call the `effectOfDamping` function to calculate the position vs. time trajectories of the simple and damped oscillators from $t = 0$ to $t = 100$. Specify initial position $x_0 = 1$and initial velocity $v_0 = 0$.

```
[time1,position1,time2,position2] = effectOfDamping(params,1,0,100,0.01);
```

Plot position vs. time graphs of the simple and damped oscillators. Observe how the amplitude of the damped oscillator decays exponentially with time.

```
plot(time1,position1)
hold on
plot(time2,position2)
```



Display the final position of the simple oscillator.

```
disp(position1(end))
```

```
0.8623
```

Display the final position of the damped oscillator. Observe that damping causes this final position to be close to the mean position $x_{mean} = 0$.

```
disp(position2(end))
```

```
0.0056
```

Generate and Run C++ MEX

To check for run-time issues, generate a C++ MEX function for the `effectOfDamping` function. Specify the first argument to have the same type and size as `params`. Specify the other arguments to be scalar doubles.

```
codegen -lang:c++ effectOfDamping -args {params,0,0,0,0} -report
```

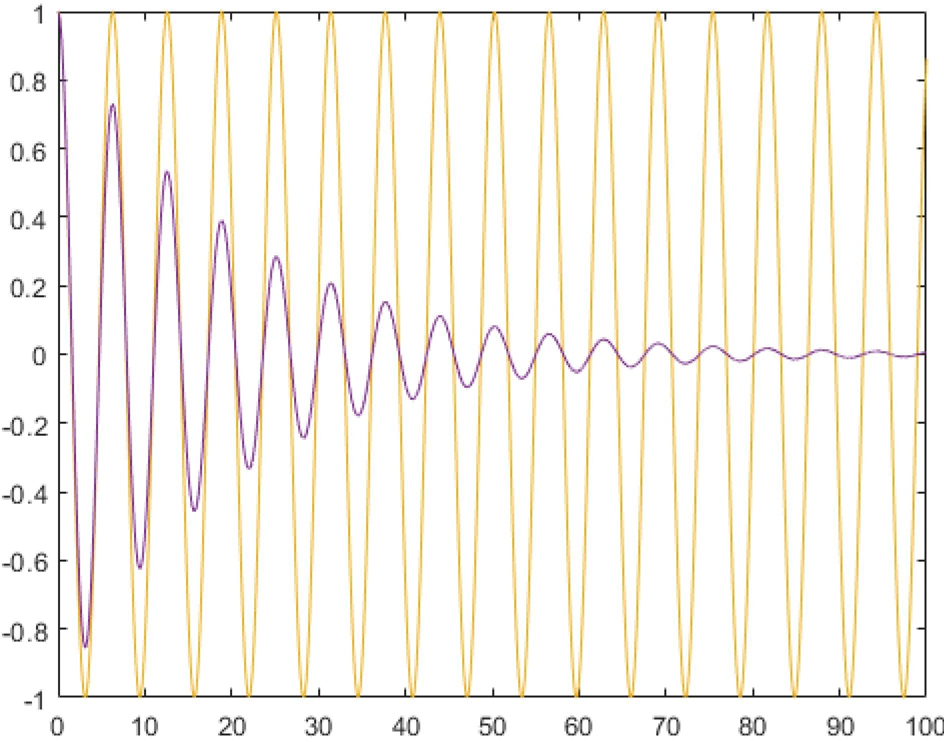
```
Code generation successful: To view the report, open('codegen\mex\effectOfDamping\html\report.mlc
```

Call the generated MEX function `effectOfDamping_mex` to calculate the position vs. time trajectories of the simple and damped oscillators from $t = 0$ to $t = 100$. Specify initial position $x_0 = 1$ and initial velocity $v_0 = 0$.

```
[time1,position1,time2,position2] = effectOfDamping_mex(params,1,0,100,0.01);
```

Plot position vs. time graphs of the simple and damped oscillators. Observe that the plot is identical to the one produced by the original MATLAB function.

```
plot(time1,position1)
hold on
plot(time2,position2)
```



Display the final positions of the two oscillators. These values are also identical to those produced by the original MATLAB code.

```
disp(position1(end))
```

```
    0.8623
```

```
disp(position2(end))
```

```
    0.0056
```

Clear the MEX file from memory.

```
clear effectOfDamping_mex
```

Generate and Inspect Static C++ Library

Create a code configuration object for generating a static C++ library with class interface. Specify the name of the interface class to be `'myOscillators'`. For these settings, the code generator

produces the entry-point function as a methods of the C++ class 'myOscillators'. The constructor and the destructor of this interface class implement the initialize and terminate functions, respectively.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.CppInterfaceStyle = 'Methods';
cfg.CppInterfaceClassName = 'myOscillators';
```

Adjust the global settings for function inlining to:

- Preserve the modularity in the code that you wrote for better readability. Set `InlineBetweenUserFunctions` to `'Readability'`.
- Generate highly optimized code for MathWorks® functions, even if that results in less readable code because you are less likely to inspect this part of your code base. Set `InlineBetweenMathWorksFunctions` to `'Speed'`.
- In the generated code, separate functions that you write and MathWorks functions so that the generated code does not look very different from your MATLAB code. Set `InlineBetweenUserAndMathWorksFunctions` to `'Readability'`.

```
cfg.InlineBetweenUserFunctions = 'Readability';
cfg.InlineBetweenUserAndMathWorksFunctions = 'Readability';
cfg.InlineBetweenMathWorksFunctions = 'Speed';
```

For more information about controlling function inlining behavior of the code generator, see "Control Inlining to Fine-Tune Performance and Readability of Generated Code" on page 34-9.

Generate a static C++ library by using the `codegen` command.

```
codegen -config cfg effectOfDamping -args {params,0,0,0,0} -report
```

```
Code generation successful: To view the report, open('codegen\lib\effectOfDamping\html\report.mld
```

Open the code generation report and inspect the generated C++ source code:

- The files `simpleOscillator.h` and `simpleOscillator.cpp` contain the implementation of the C++ class for the simple oscillator. The files `dampedOscillator.h` and `dampedOscillator.cpp` contain the implementation of the C++ class for the damped oscillator. The inheritance structure of the MATLAB classes is flattened in the generated code. So, `dampedOscillator` is not a subclass of `simpleOscillator` and reimplements all the methods that the corresponding MATLAB class inherits. For more information on the mapping between the MATLAB classes and the C++ classes, see "Generate C++ Classes for MATLAB Classes" on page 16-2.
- The MATLAB package is mapped to a C++ namespace. In the generated code, the `simpleOscillator` and `dampedOscillator` classes are defined in the `mySystem` namespace. For more information, see "Organize Generated C++ Code into Namespaces" on page 39-9.
- The files `myOscillators.h` and `myOscillators.cpp` contain the implementation of the interface class `myOscillators`. The entry-point function is implemented in the method `myOscillators::effectOfDamping`. The initialize and terminate functions are implemented in the class constructor and the class destructor, respectively. The next part of this example shows how to use this class interface in your custom C++ main function. For more information, see "Generate C++ Code with Class Interface" on page 39-4.
- The size of output arguments of the `effectOfDamping` function are determined by the run-time inputs `timeInterval` and `timeStep`. So, the generated code represents these arguments as

dynamic arrays C++ that are implemented by using the `coder::array` class template. The next part of this example shows how to use the `coder::array` class template in your custom C++ main function. For more information, see “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces” on page 31-15.

For example, here is the declaration of the generated `mySystem::simpleOscillator` class contained in the header file `simpleOscillator.h`.

```
type codegen/lib/effectOfDamping/simpleOscillator.h
```

```
//
// File: simpleOscillator.h
//
// MATLAB Coder version            : 5.2
// C/C++ source code generated on  : 23-Feb-2021 16:50:02
//

#ifndef SIMPLEOSCILLATOR_H
#define SIMPLEOSCILLATOR_H

// Include Files
#include "rtwtypes.h"
#include "coder_array.h"
#include <cstddef>
#include <cstdlib>

// Type Definitions
namespace mySystem {
class simpleOscillator {
public:
  void init(double m, double k);
  void evolution(double initialPosition, double initialVelocity,
                 double timeInterval, double timeStep,
                 coder::array<double, 1U> &b_time,
                 coder::array<double, 1U> &position) const;
  double dynamics(double initialPosition, double initialVelocity,
                  double timeInterval) const;
  double amplitude(double initialPosition, double initialVelocity) const;
  double angularFrequency() const;
  double phase(double initialPosition, double initialVelocity) const;

protected:
  double mass;
  double springConstant;
};

} // namespace mySystem

#endif
//
// File trailer for simpleOscillator.h
//
// [EOF]
//
```

If you have Embedded Coder®, you can set the `VerificationMode` property of the configuration object to `'SIL'` and generate a SIL MEX function `effectOfDamping_sil`. This SIL interface allows

you to verify the production ready source code inside the MATLAB environment. See “Software-in-the-Loop Execution From Command Line” (Embedded Coder).

Generate and Run Executable

In the previous part of this example, when you generate library code, the code generator also produces example main files `main.h` and `main.cpp` in the `examples` subfolder of the build folder. The supporting C++ files `main_damped_oscillator.h` and `main_damped_oscillator.cpp` are modified versions of these example files.

- In `main_damped_oscillator.cpp`, the `main` function uses the interface class `myOscillators` to interact with the generated code. This function uses the C++ `new` operator to allocate memory for an instance of `myOscillators`, invokes the `main_effectOfDamping` function, and finally frees the memory by using the C++ `delete` operator.
- The `main_effectOfDamping` function performs the same computation that the MATLAB script in the first part of this example does. It uses the `coder::array` API to interact with the dynamic arrays that the generated `effectOfDamping` function return. At the end of its execution, the `main_effectOfDamping` function prints the final positions of the two oscillators.

Create a code configuration object for generating a C++ executable. Use the same settings as in the previous part of this example.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
cfg.CppInterfaceStyle = 'Methods';
cfg.CppInterfaceClassName = 'myOscillators';

cfg.InlineBetweenUserFunctions = 'Readability';
cfg.InlineBetweenUserAndMathWorksFunctions = 'Readability';
cfg.InlineBetweenMathWorksFunctions = 'Speed';
```

Specify the custom C++ source file and the custom include folder.

```
cfg.CustomSource = 'main_damped_oscillator.cpp';
cfg.CustomInclude = pwd;
```

Generate an executable by using the `codegen` command.

```
codegen -config cfg main_damped_oscillator.cpp main_damped_oscillator.h effectOfDamping -args {pa
```

```
Code generation successful: To view the report, open('codegen\exe\effectOfDamping\html\report.ml
```

Run the generated executable. Observe that the final positions of the two oscillators that this execution returns match the outputs of the original MATLAB code.

```
if isunix
    system('./effectOfDamping')
elseif ispc
    system('effectOfDamping.exe')
else
    disp('Platform is not supported')
end
```

```
0.862319
0.00563263
```

```
ans = 0
```

See Also

`codegen` | `coder.config`

More About

- “C++ Code Generation” on page 39-2

Simulation Data Inspector

- "View Data in the Simulation Data Inspector" on page 40-2
- "Import Data from a CSV File into the Simulation Data Inspector" on page 40-11
- "Microsoft Excel Import, Export, and Logging Format" on page 40-16
- "Configure the Simulation Data Inspector" on page 40-24
- "How the Simulation Data Inspector Compares Data" on page 40-32
- "Save and Share Simulation Data Inspector Data and Views" on page 40-37
- "Inspect and Compare Data Programmatically" on page 40-42
- "Limit the Size of Logged Data" on page 40-47

View Data in the Simulation Data Inspector

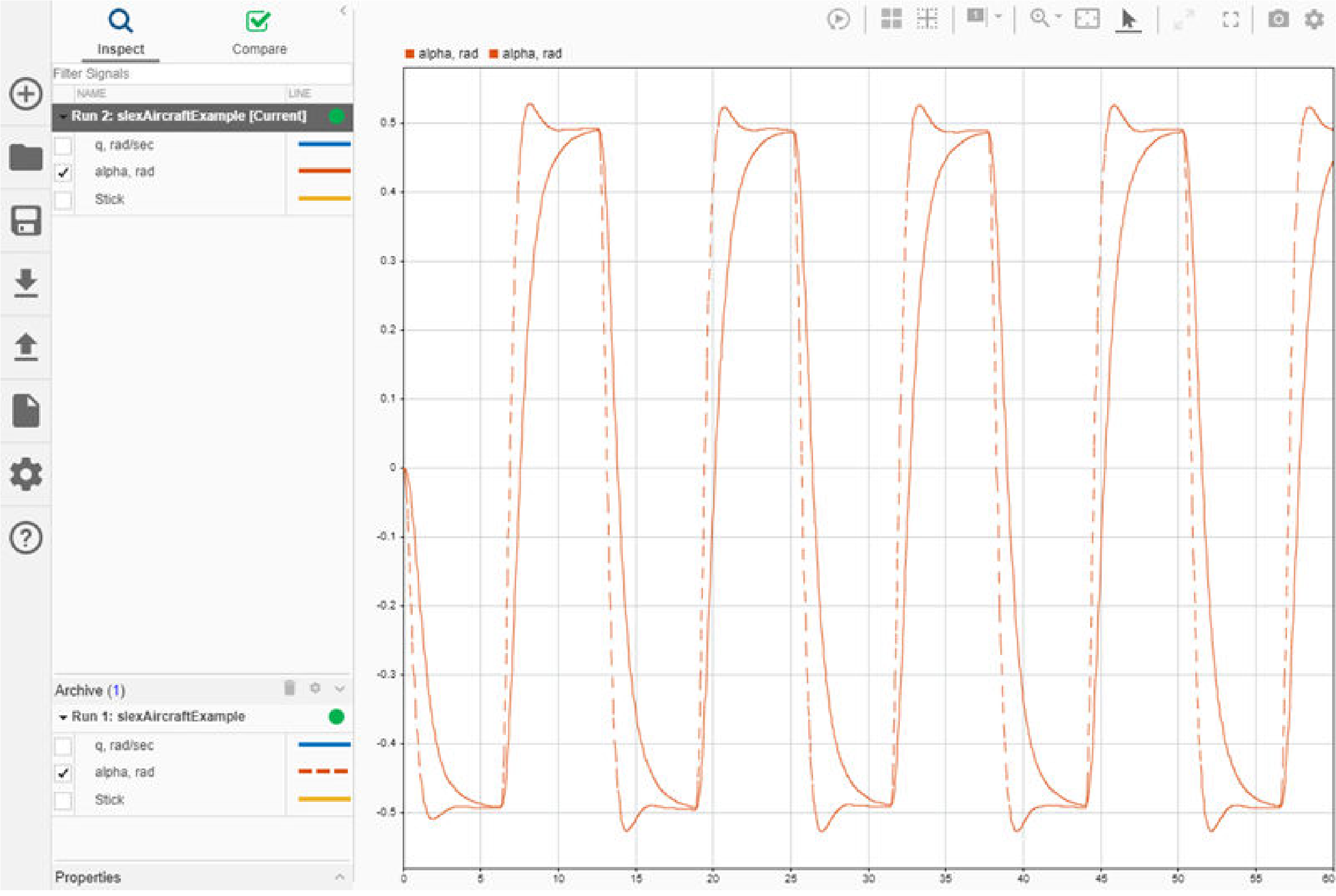
You can use the Simulation Data Inspector to visualize the data you generate throughout the design process. Simulation data that you log in a Simulink model logs to the Simulation Data inspector. You can also import test data and other recorded data into the Simulation Data Inspector to inspect and analyze it alongside the logged simulation data. The Simulation Data Inspector offers several types of plots, which allow you to easily create complex visualizations of your data.

View Logged Data

Logged signals as well as outputs and states logged using the `Dataset` format automatically log to the Simulation Data Inspector when you simulate a model. You can also record other kinds of simulation data so the data appears in the Simulation Data Inspector at the end of the simulation. To see states and output data logged using a format other than `Dataset` in the Simulation Data Inspector, in the **Model Configuration Parameters Data Import/Export** pane, select the **Record logged workspace data in Simulation Data Inspector** option.

Note When you log states and outputs using the `Structure` or `Array` format, you must also log time for the data to record to the Simulation Data Inspector.

The Simulation Data Inspector displays available data in the table in the **Inspect** pane. To plot a signal, select the check box next to the signal. You can modify the layout and add different visualizations to analyze the simulation data. For more information, see “Create Plots Using the Simulation Data Inspector” (Simulink).



The Simulation Data Inspector manages incoming simulation data using the archive. By default, the previous run moves to the archive when you start a new simulation. You can plot signals from the archive, or you can drag runs of interest back into the work area.

Import Data from the Workspace or a File

You can import data from the base workspace or from a file to view on its own or alongside simulation data. The Simulation Data Inspector supports all built-in data types and many data formats for importing data from the workspace. In general, whatever the format, sample values must be paired with sample times. The Simulation Data Inspector allows up to 8000 channels per signal in a run created from imported workspace data.

You can also import data from these types of files:

- MAT file
- CSV file — Format data as shown in “Import Data from a CSV File into the Simulation Data Inspector” (Simulink).
- Microsoft Excel® file — Format data as described in “Microsoft Excel Import, Export, and Logging Format” (Simulink).
- MDF file — MDF file import is supported for Linux and Windows operating systems. The MDF file must have a `.mdf`, `.mf4`, `.mf3`, `.data`, or `.dat` file extension and contain data with only integer and floating data types.

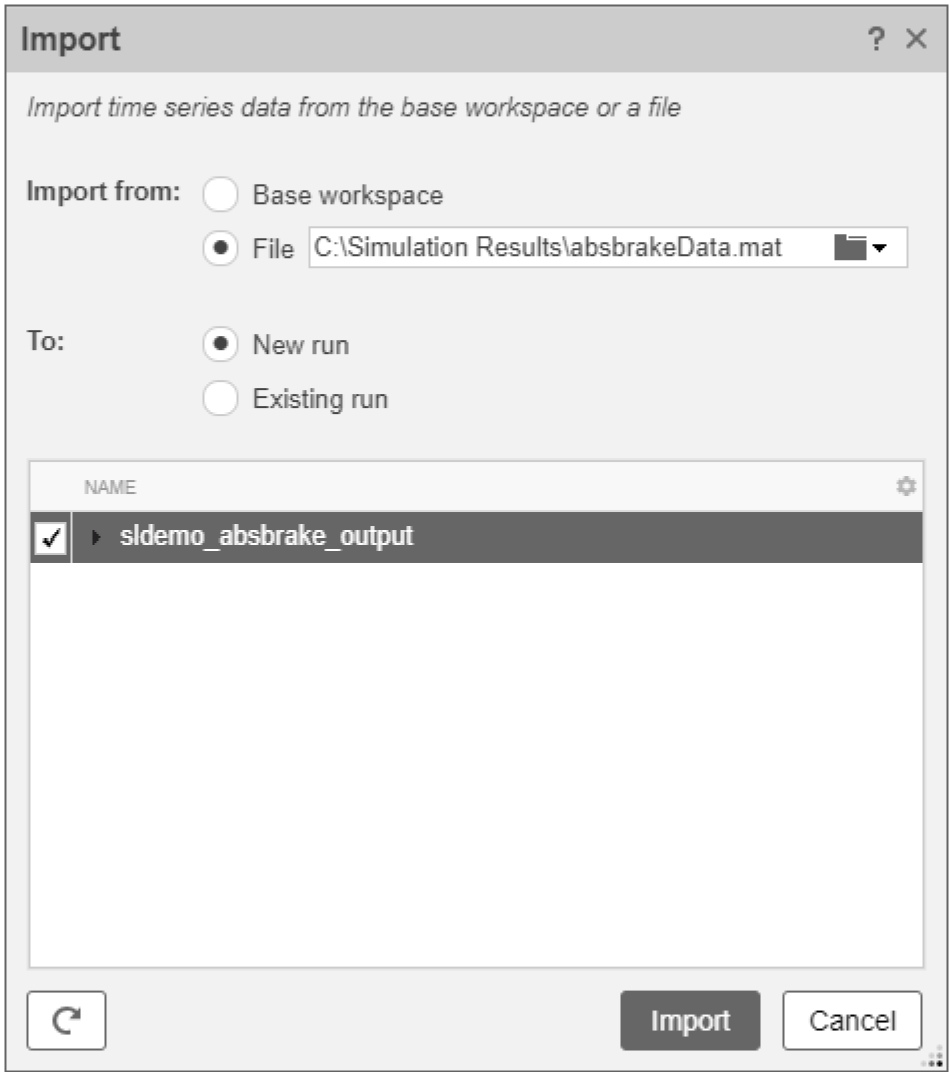
- ULG file — Flight log data import requires a UAV Toolbox license.

To import data from the workspace or from a file that is saved in a data or file format that the Simulation Data Inspector does not support, you can write your own workspace data or file reader to import the data using the `io.reader` class. You can also write a custom reader to use instead of the built-in reader for supported file types. For examples, see:

- "Import Data Using a Custom File Reader" (Simulink)
- "Import Workspace Variables Using a Custom Data Reader" (Simulink)

To import data, select the **Import** button in the Simulation Data Inspector.

In the Import dialog, you can choose to import data from the workspace or from a file. The table below the options shows data available for import. If you do not see your workspace variable or file contents in the table, that means the Simulation Data Inspector does not have a built-in or registered reader that supports that data. You can select which data to import using the check boxes, and you can choose whether to import that data into an existing run or a new run.



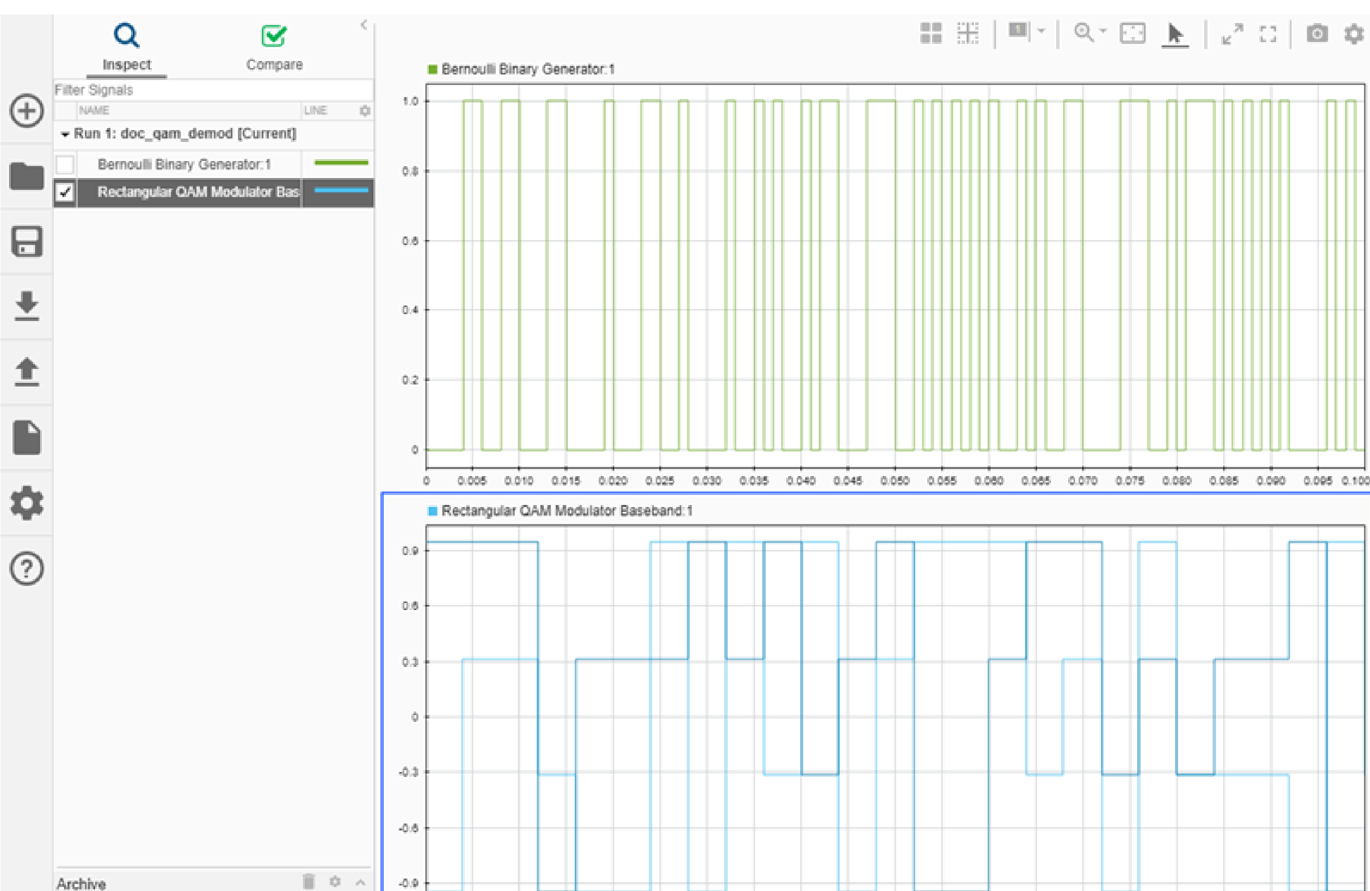
When you import data into a new run, the run always appears in the work area. You can manually move imported runs to the archive.

View Complex Data

To view complex data in the Simulation Data Inspector, import the data or log the signals to the Simulation Data Inspector. You can control how to visualize the complex signal using the **Properties** pane in the Simulation Data Inspector and in the **Instrumentation Properties** for the signal in the model. To access the **Instrumentation Properties** for a signal, right-click the logging badge for the signal and select **Properties**.

You can specify the **Complex Format** as Magnitude, Magnitude-Phase, Phase, or Real-Imaginary. If you select Magnitude-Phase or Real-Imaginary for the **Complex Format**, the Simulation Data Inspector plots both components of the signal when you select the check box for the signal. For signals in Real-Imaginary format, the **Line Color** specifies the color of the real component of the signal, and the imaginary component is a different shade of the **Line Color**. For example, the `Rectangular QAM Modular Baseband` signal on the lower graph displays the real component of

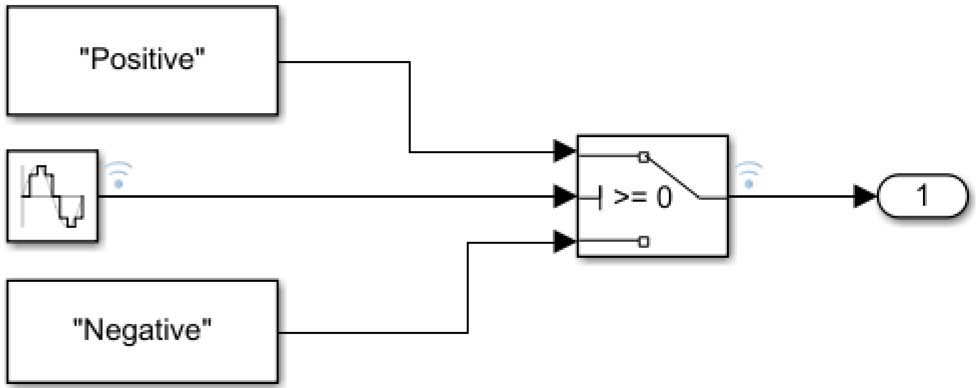
the signal in light blue, matching the **Line Color** parameter, and the imaginary component is shown in a darker shade of blue.



For signals in Magnitude-Phase format, the **Line Color** specifies the color of the magnitude component, and the phase is displayed in a different shade of the **Line Color**.

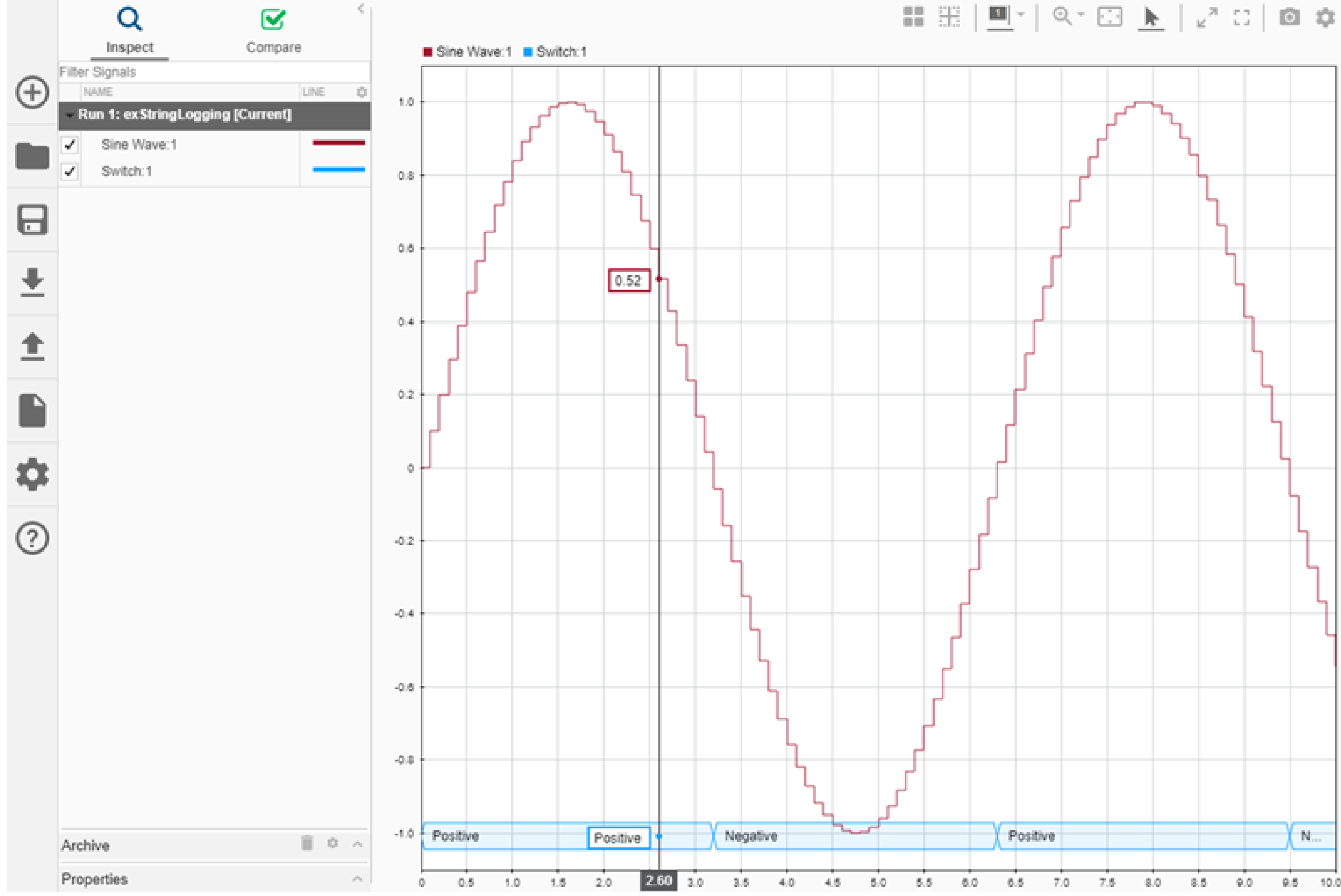
View String Data

You can log and view string data with your signal data in the Simulation Data Inspector. For example, consider this simple model. The value of the sine wave block controls whether the switch sends a string reading `Positive` or `Negative` to the output.

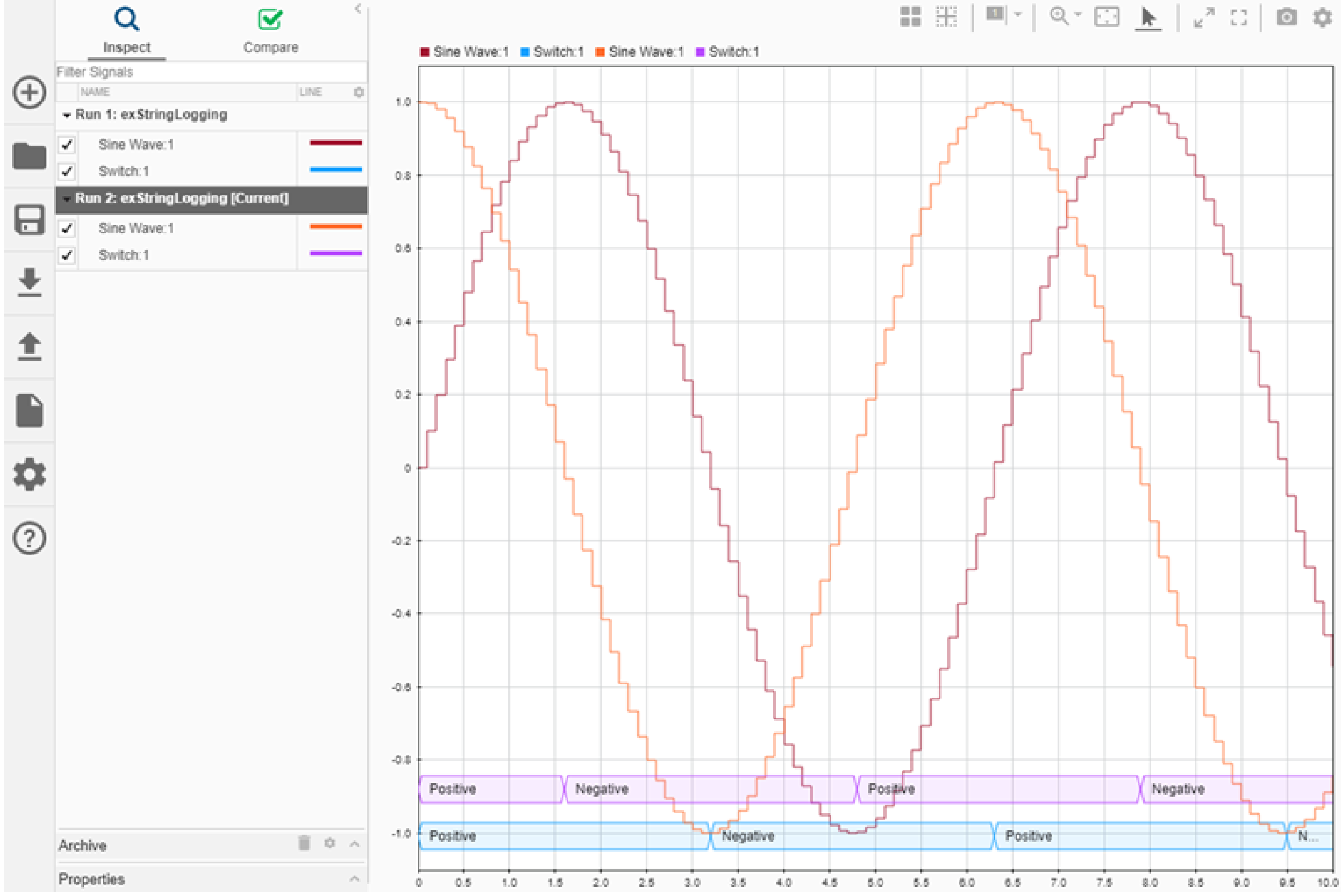


The plot shows the results of simulating the model. The string signal is shown at the bottom of the graphical viewing area. The value of the signal is displayed inside a band, and transitions in the string signal's value are marked with criss-crossed lines.

You can use cursors to inspect how the string signal values correspond with the sine signal's values.



When you plot multiple string signals on a plot, the signals stack in the order they were simulated or imported, with the most recent signal positioned at the top. For example, you might consider the effect of changing the phase of the sine wave controlling the switch.

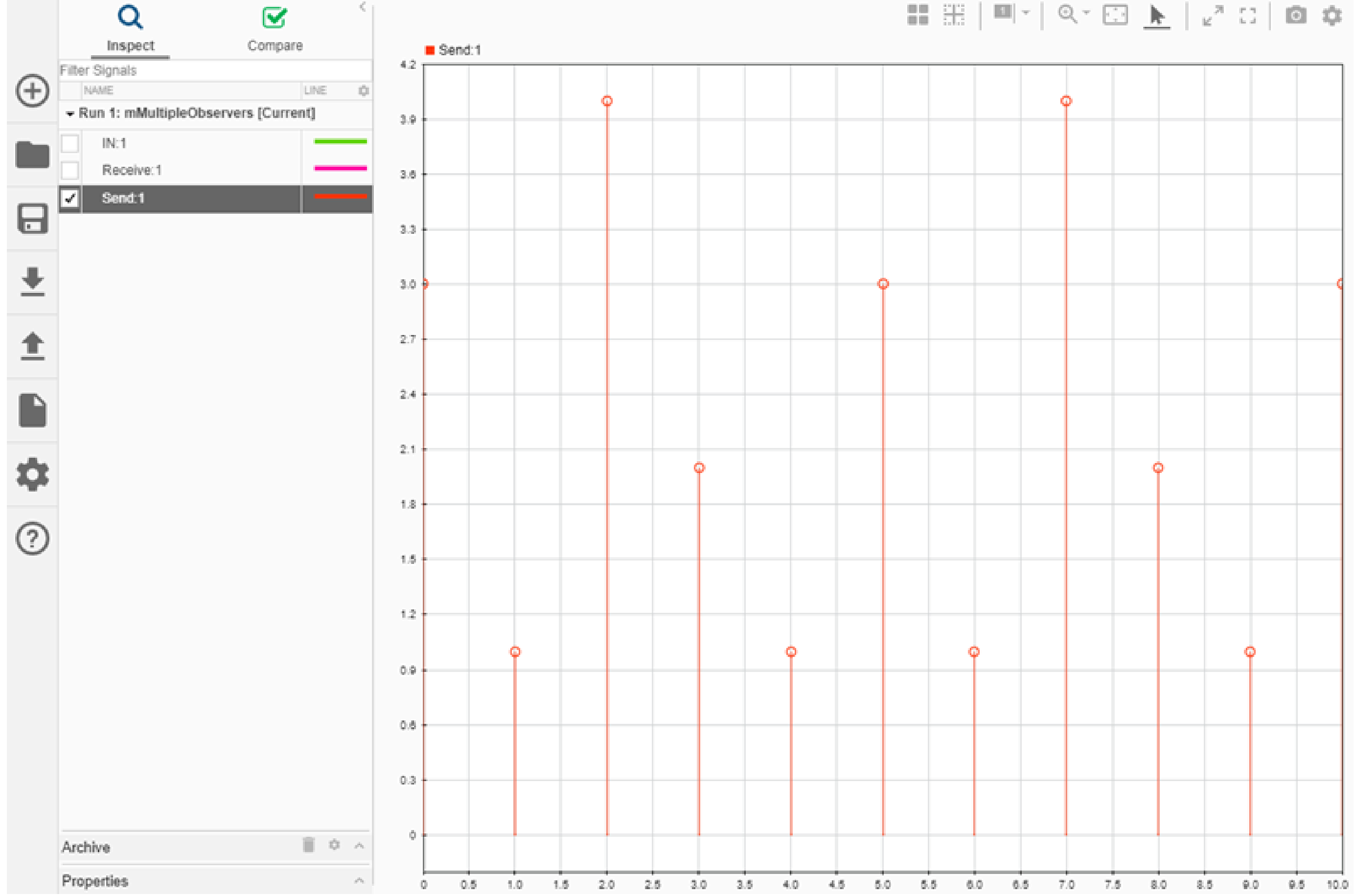


View Frame-Based Data

Processing data in frames rather than point by point provides a performance boost needed in some applications. To view frame-based data in the Simulation Data Inspector, you have to specify that the signal is frame-based in the **Instrumentation Properties** for the signal. To access the **Instrumentation Properties** dialog for a signal, right-click the signal's logging badge and select **Properties**. To specify a signal as frame-based, select **Columns as channels (frame based)** for **Input processing**.

View Event-Based Data

You can log or import event data to the Simulation Data Inspector. To view the logged event-based data, select the check box next to `Send: 1`. The Simulation Data Inspector displays the data as a stem plot, with each stem representing the number of events that occurred for a given sample time.



See Also

More About

- Inspect Simulation Data (Simulink)
- Compare Simulation Data (Simulink)
- Share Simulation Data Inspector Data and Views on page 40-37
- Decide How to Visualize Data (Simulink)
- Dataset Conversion for Logged Data (Simulink)

Import Data from a CSV File into the Simulation Data Inspector

To import data into the Simulation Data Inspector from a CSV file, format the data in the CSV file. Then, you can import the data using the Simulation Data Inspector UI or the `Simulink.sdi.createRun` function.

Tip When you want to import data from a CSV file where the data is formatted differently from the specification in this topic, you can write your own file reader for the Simulation Data Inspector using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the CSV file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.

myData - Notepad

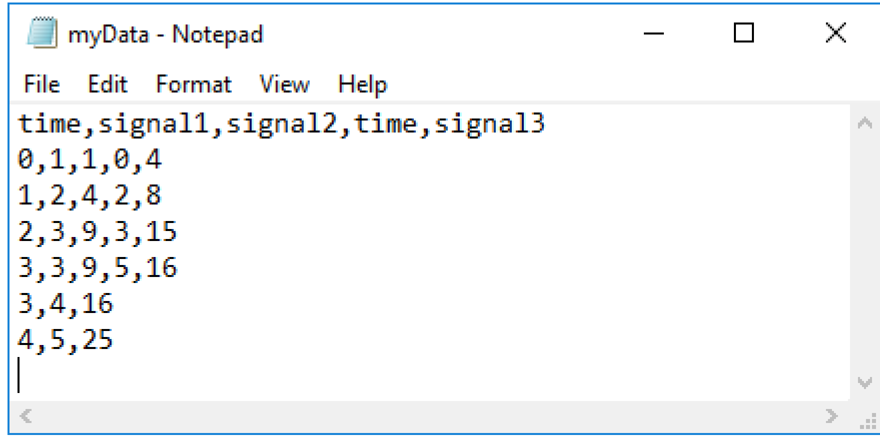
File Edit Format View Help

```
time,signal1,signal2,signal3
0,1,1,4
1,2,4,8
2,3,9,15
3,3,9,16
3,4,16,23
4,5,25,42
```

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values render as missing data. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.



Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

You can specify signal metadata in the CSV file to indicate the signal data type, units, interpolation method, block path, and port index. List metadata for each signal in rows between the signal name and the signal data. Label metadata according to this table.

Signal Property	Label	Value
Data type	`Type:`	Built-in data type.
Units	`Unit:`	Supported unit. For example, `Unit: m/s` specifies units of meters per second. For a list of supported units, enter `showunitslist` in the MATLAB Command Window.
Interpolation method	`Interp:`	`linear`, `zoh` for zero order hold, or `none`.
Block Path	`BlockPath:`	Path to the block that generated the signal.
Port Index	`PortIndex:`	Integer.

You can also import a signal with a data type defined by an enumeration class. Instead of using the `Type:` label, use the `Enum:` label and specify the value as the name of the enumeration class. The definition for the enumeration class must be saved on the MATLAB path.

When an imported file does not specify signal metadata, the Simulation Data Inspector assumes double data type and linear interpolation. You can specify the interpolation method as linear, zoh (zero-order hold), or none. If you do not specify units for the signals in your file, you can assign units to the signals in the Simulation Data Inspector after you import the file.

You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

myData - Notepad

File Edit Format View Help

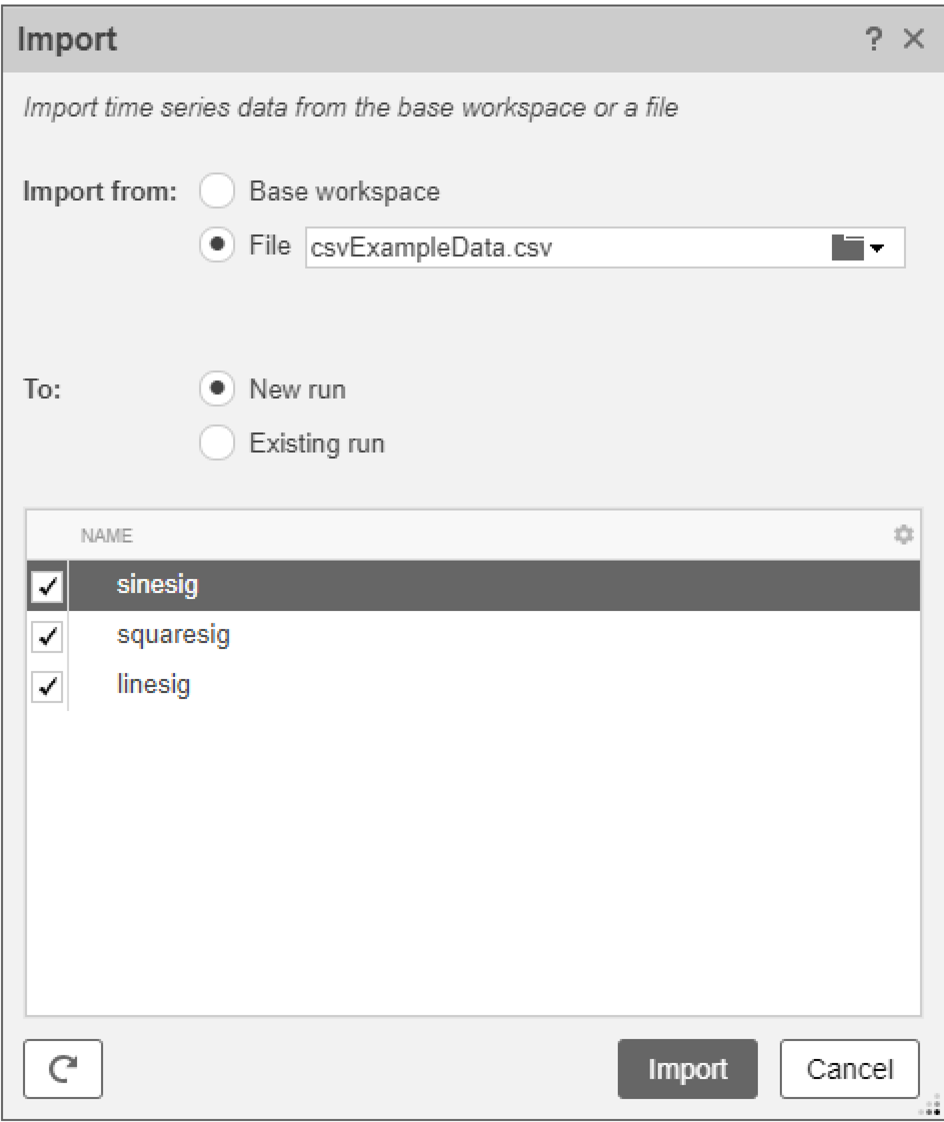
```
time,signal1,signal2,time,signal3
 ,Interp: zoh, , ,Interp: zoh
 ,Type: int8,Type: int32
 ,Unit: m, , ,Unit: m/s
0,1,1,0,4
1,2,4,2,8
2,3,9,3,15
3,3,9,5,16
3,4,16
4,5,25
```

Import Data from a CSV File

You can import data from a CSV file using the Simulation Data Inspector UI or using the `Simulink.sdi.createRun` function.

To import data using the UI, open the Simulation Data Inspector using the `Simulink.sdi.view` function or the **Data Inspector** button in the Simulink™ toolstrip. Then, click the **Import** button.

In the Import dialog, select the option to import data from a file and navigate in the file system to select the file. After you select the file, data available for import shows in the table. You can choose which signals to import and whether to import them to a new or existing run. This example imports all available signals to a new run. After selecting the options, click the **Import** button.



When you import data into a new run using the UI, the new run name includes the run number followed by `Imported_Data`.

When you import data programmatically, you can specify the name of the imported run.

```
csvRunID = Simulink.sdi.createRun('CSV File Run','file','csvExampleData.csv');
```

See Also

Functions
`Simulink.sdi.createRun`

More About

- "View Data in the Simulation Data Inspector" (Simulink)

- “Microsoft Excel Import, Export, and Logging Format” (Simulink)
- “Import Data Using a Custom File Reader” (Simulink)

Microsoft Excel Import, Export, and Logging Format

Using the Simulation Data Inspector or Simulink Test, you can import data from a Microsoft Excel file or export data to a Microsoft Excel file. You can also log data to an Excel file using the Record block. The Simulation Data Inspector, Simulink Test, and the Record block all use the same file format, so you can use the same Microsoft Excel file with multiple applications.

Tip When the format of the data in your Excel file does not match the specification in this topic, you can write your own file reader to import the data using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the Excel file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.

	A	B	C	D
1	time	signal1	signal2	signal3
2	0	1	1	4
3	1	2	4	8
4	2	3	9	15
5	3	3	9	16
6	3	4	16	23
7	4	5	25	42

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values imported from the Excel file render as missing data in the Simulation Data Inspector. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2	0	1	1	0	4
3	1	2	4	2	8
4	2	3	9	3	15
5	3	3	9	5	16
6	3	4	16		
7	4	5	25		

Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

The file can include metadata for signals such as data type, units, and interpolation method. Metadata for each signal is listed in rows between the signal names and the signal data. You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Label each piece of metadata according to this table. The table also indicates which tools and operations support each piece of metadata.

Property Descriptions

Signal Property	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Data type	`Type:`	Built-in data type.	**Supported**	**Supported**	**Supported**
Units	`Unit:`	Supported unit. For example, `Unit: m/s` specifies units of meters per second. For a list of supported units, enter `showunitslist` in the MATLAB Command Window.	**Supported**	**Supported**	**Supported**
Interpolation method	`Interp:`	`linear`, `zoh` for zero order hold, or `none`.	**Supported**	**Supported**	**Supported**
Synchronization method	`Sync:`	`union` or `intersection`.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Relative tolerance	`RelTol:`	Percentage, represented as a decimal. For example, `RelTol: 0.1` specifies a 10% relative tolerance.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Absolute tolerance	`AbsTol:`	Numeric value.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Time tolerance	`TimeTol:`	Numeric value, in seconds.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**

Signal Property	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Leading tolerance	`LeadingTol:`	Numeric value, in seconds.	**Supported** *Only visible in Simulink Test.*	**Not Supported** *Metadata not included in exported file.*	**Supported**
Lagging tolerance	`LaggingTol:`	Numeric Value, in seconds.	**Supported** *Only visible in Simulink Test.*	**Not Supported** *Metadata not included in exported file.*	**Supported**
Block Path	`BlockPath:`	Path to the block that generated the signal.	**Supported**	**Supported**	**Supported**
Port Index	`PortIndex:`	Integer.	**Supported**	**Supported**	**Supported**
Name	`Name:`	Signal name	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**

When an imported file does not specify signal metadata, `double` data type, `linear` interpolation, and `union` synchronization are used.

User-Defined Data Types

In addition to built-in data types, you can use other labels in place of the `DataType:` label to specify fixed-point, enumerated, alias, and bus data types.

Property Descriptions

Data Type	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Enumeration	`Enum:`	Name of the enumeration class.	**Supported** *Enumeration class definition must be saved on the MATLAB path.*	**Supported** *Enumeration class definition must be saved on the MATLAB path.*	**Supported** *Enumeration class definition must be saved on the MATLAB path.*
Alias	`Alias:`	Name of a `Simulink.AliasType` object in the MATLAB workspace.	**Supported** *For matrix and complex signals, specify the alias data type on the first channel.*	**Not Supported**	**Supported** *For matrix and complex signals, specify the alias data type on the first channel.*
Fixed-point	`Fixdt:`	• `fixdt` constructor. • Name of a `Simulink.NumericType` object in the MATLAB workspace. • Name of a fixed-point data type as described in “Fixed-Point Numbers in Simulink” (Fixed-Point Designer).	**Supported**	**Not Supported**	**Supported**
Bus	`Bus:`	Name of a `Simulink.Bus` object in the MATLAB workspace.	**Supported**	**Not Supported**	**Supported**

When you specify the type using the name of a `Simulink.Bus` object and the object is not in the MATLAB workspace, the data still imports from the file. However, individual signals in the bus use data types described in the file rather than data types defined in the `Simulink.Bus` object.

Complex, Multidimensional, and Bus Signals

You can import and export complex, multidimensional, and bus signals using an Excel file. The signal name for a column of data indicates whether that data is part of a complex, multidimensional, or bus signal. Excel file import and export do not support array of bus signals.

Multidimensional signal names include index information in parentheses. For example, the signal name for a column might be `signal1(2,3)`. When you import data from a file that includes multidimensional signal data, elements in the data not included in the file take zero sample values with the same data type and complexity as the other elements.

Complex signal data is always in real-imaginary format. Signal names for columns containing complex signal data include `(real)` and `(imag)` to indicate which data each column contains. When you import data from a file that includes imaginary signal data without specifying values for the real component of that signal, the signal values for the real component default to zero.

Multidimensional signals can contain complex data. The signal name includes the indication for the index within the multidimensional signal and the real or imaginary tag. For example, `signal1(1,3)(real)`.

Dots in signal names specify the hierarchy for bus signals. For example:

- `bus.y.a`
- `bus.y.b`
- `bus.x`

	A	B	C	D	E
1	time	bus.y.a	bus.y.b	time	bus.x
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Tip When the name of your signal includes characters that could make it appear as though it were part of a matrix, complex signal, or bus, use the `Name` metadata option to specify the name you want the imported signal to use in the Simulation Data Inspector and Simulink Test.

Function-Call Signals

Signal data specified in columns before the first time column is imported as one or more function-call signals. The data in the column specifies the times at which the function-call signal was enabled. The imported signals have a value of `1` for the times specified in the column. The time values for function-call signals must be double, scalar, and real, and must increase monotonically.

When you export data from the Simulation Data Inspector, function-call signals are formatted the same as other signals, with a time column and a column for signal values.

Simulation Parameters

You can import data for parameter values used in simulation. In the Simulation Data Inspector, the parameter values are shown as signals. Simulink Test uses imported parameter values to specify values for those parameters in the tests it runs based on imported data.

Parameter data is specified using two or three columns. The first column specifies the parameter names, with the cell in the header row for that column labeled `Parameter:`. The second column specifies the value used for each parameter, with the cell in the header row labeled `Value:`. Parameter data may also include a third column that contains the block path associated with each parameter, with the cell in the header row labeled `BlockPath:`. Specify names, values, and block paths for parameters starting in the first row that contains signal data, below rows used to specify signal metadata. For example, this file specifies values for two parameters, X and Y.

	A	B	C	D	E	F	G
1	time	signal1	signal2	time	signal3	Parameter:	Value:
2		Interp: zoh			Interp: zoh		
3		Type: int8	Type: int32				
4		Unit: m			Unit: m/s		
5	0	1	1	0	4	X	2
6	1	2	4	2	8	Y	1.2
7	2	3	9	3	15		
8	3	3	9	5	16		
9	3	4	16				
10	4	5	25				

Multiple Runs

You can include data for multiple runs in a single file. Within a sheet, you can divide data into runs by labeling data with a simulation number and a source type, such as `Input` or `Output`. Specify the simulation number and source type as additional signal metadata, using the label `Simulation:` for the simulation number and the label `Source:` for the source type. The Simulation Data Inspector uses the simulation number and source type only to determine which signals belong in each run. Simulink Test uses the information to define inputs, parameters, and acceptance criteria for tests to run based on imported data.

You do not need to specify the simulation number and output type for every signal. Signals to the right of a signal with a simulation number and source use the same simulation number and source until the next signal with a different source or simulation number. For example, this file defines data for two simulations and imports into four runs in the Simulation Data Inspector:

- **Run 1** contains `signal1` and `signal2`.
- **Run 2** contains `signal3`, X, and Y.
- **Run 3** contains `signal4`.

- **Run 4** contains `signal5`.

	A	B	C	D	E	F	G	H	I	J
1	time	signal1	signal2	time	signal3	Parameter:	Values:	time	signal4	signal5
2		Interp: zoh			Interp: zoh					
3		Type: int8	Type: int32							
4		Unit: m			Unit: m/s					
5		Simulation: 1							Simulation: 2	
6		Source: Input			Source: Output				Source: Input	Source: Output
7	0	1	1	0	4	X	2	1	2	1
8	1	2	4	2	8	Y	1.2	2	6	3
9	2	3	9	3	15			3	4	5
10	3	3	9	5	16			4	8	7
11	3	4	16					5	10	2
12	4	5	25							

You can also use sheets within the Microsoft Excel file to divide the data into runs and tests. When you do not specify simulation number and source information, the data on each sheet is imported into a separate run in the Simulation Data Inspector. When you export multiple runs from the Simulation Data Inspector, the data for each run is saved on a separate sheet. When you import a Microsoft Excel file that contains data on multiple sheets into Simulink Test, you are prompted to specify how to import the data.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.exportRun`

More About

- “View Data in the Simulation Data Inspector” (Simulink)
- “Import Data from a CSV File into the Simulation Data Inspector” (Simulink)
- “Import Data Using a Custom File Reader” (Simulink)

Configure the Simulation Data Inspector

The Simulation Data Inspector supports a wide range of use cases for analyzing and visualizing data. You can modify preferences in the Simulation Data Inspector to match your visualization and analysis requirements. The preferences that you specify persist between MATLAB sessions.

By specifying preferences in the Simulation Data Inspector, you can configure options such as:

- How signals and metadata are displayed.
- Which data automatically imports from parallel simulations.
- Where prior run data is retained and how much prior data to store.
- How much memory is used during save operations.
- The system of units used to display signals.

Open the Simulation Data Inspector preferences by selecting the **Preferences** button.

Note You can restore all preferences in the Simulation Data Inspector to default values by clicking **Restore Defaults** in the dialog or by using the `Simulink.sdi.clearPreferences` function.

Logged Data Size and Location

By default, simulation data logs to disk with data loaded into memory on demand, and the maximum size of logged data is constrained only by available disk space. You can use the **Disk Management** settings in the Simulation Data Inspector to directly control the size and location of logged data.

The **Record mode** setting specifies whether logged data is retained after simulation. When you change the **Record mode** setting to **View during simulation only**, no logged data is available in the Simulation Data Inspector or the workspace after the simulation completes. Only use this mode when you do not want to save logged data. The **Record mode** setting reverts to **View and record data** each time you start MATLAB. Changing the **Record mode** setting can affect other applications, such as visualization tools. For details, see “View Data Only During Simulation” (Simulink).

To directly limit the size of logged data, you can specify a minimum amount of free disk space or a maximum size for the logged data. By default, logged data must leave at least 100 MB of free disk space with no maximum size limit. Specify the required disk space and maximum size in GB, and specify `0` to apply no disk space requirement or no maximum size limit.

When you specify a minimum disk space requirement or a maximum size for logged data, you can also specify whether to prioritize retaining data from the current simulation or data from prior simulations when approaching the limit. By default, the Simulation Data Inspector prioritizes retaining data for the current run by deleting data for prior runs. To prioritize retaining prior data, change the **When low on disk space** setting to **Keep prior runs and stop recording**. You see a warning message when prior runs are deleted and when recording is disabled. If recording is disabled due to the size of logged data, you need to change the **Record Mode** back to **View and record data** to continue logging data, after you have freed up disk space. For more information, see “Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data” (Simulink).

The **Storage Mode** setting specifies whether to log data to disk or to memory. By default, data logs to disk. When you configure a parallel worker to log data to memory, data transfer back to the host is not supported.

You can also specify the location of the temporary file that stores logged data. By default, data logs to the temporary files directory on your computer. You may change the file location when you need to log large amounts of data and a secondary drive provides more storage capacity. Logging data to a network location can degrade performance.

Programmatic Use

You can programmatically configure and check each preference value.

Preference	Functions
Record mode	`Simulink.sdi.setRecordData` `Simulink.sdi.getRecordData`
Required Free Space	`Simulink.sdi.setRequiredFreeSpace` `Simulink.sdi.getRequiredFreeSpace`
Max Disk Usage	`Simulink.sdi.setMaxDiskUsage` `Simulink.sdi.getMaxDiskUsage`
When low on disk space	`Simulink.sdi.setDeleteRunsOnLowSpace` `Simulink.sdi.getDeleteRunsOnLowSpace`
Storage Mode	`Simulink.sdi.setStorageMode` `Simulink.sdi.getStorageMode`
Storage Location	`Simulink.sdi.setStorageLocation` `Simulink.sdi.getStorageLocation`

Archive Behavior and Run Limit

The Simulation Data Inspector archive stores runs in a collapsible pane, allowing you to manage the contents of the work area without deleting run data. You can configure whether the Simulation Data Inspector automatically moves prior simulation runs to the archive. You can also limit the number of runs stored in the archive.

Manage Runs in the Archive

By default, the Simulation Data Inspector automatically archives simulation runs. When you simulate a model, the prior simulation run moves to the archive, and the Simulation Data Inspector updates the view to show the data for aligned signals in the current run.

The archive does not impose functional limitations on the runs and signals it contains. You can plot signals from the archive, and you can use runs and signals in the archive in comparisons. You can drag runs of interest from the archive to the work area and vice versa whether the **Automatically Archive** setting is enabled or disabled. To prevent the Simulation Data Inspector from automatically moving prior simulations runs to the archive, clear the **Automatically archive** setting.

When you import runs into the Simulation Data Inspector, the imported runs appear in the work area, and the **Current** tag remains on the most recent simulation run. You can import signals to existing runs in the work area and in the archive.

Tip You can delete the contents of the archive manually using the trash icon.

Limit Data Retention

To reduce the amount of data the Simulation Data Inspector retains, you can configure a limit for the number of runs stored in the archive. When the number of runs in the archive reaches the size limit, the Simulation Data Inspector starts to delete runs on a first-in, first-out basis.

The size limit applies only to runs in the archive. For the Simulation Data Inspector to automatically limit the data it retains by deleting old runs, select **Automatically archive** and specify a size limit.

By default, the Simulation Data Inspector retains the last `20` runs moved to the archive. To remove the limit, select **No limit**. To specify the maximum number of runs to store in the archive, select **Last n runs** and enter the desired limit. The Simulation Data Inspector warns you when you specify a limit that would delete runs already in the archive.

Programmatic Use

Configure the **Automatically archive** setting programmatically using the `Simulink.sdi.setAutoArchiveMode` function.

Specify the number of runs to retain in the archive using the `Simulink.sdi.setArchiveRunLimit` function.

Incoming Run Names and Location

You can configure how the Simulation Data Inspector handles incoming runs from import or simulation. You can choose whether new runs are added at the top of the work area or the bottom and specify a naming rule to use for runs created from simulation.

By default, the Simulation Data Inspector adds new runs below prior runs in the work area. The **Archive** settings also affect the location of runs. By default, prior runs are moved to the archive when a new simulation run is created.

The run naming rule is used to name runs created from simulation. You can create the run naming rule using a mix of literal text that is used in the run name as-is and one or more tokens that represent metadata about the run. By default, the Simulation Data Inspector names runs using the run index and model name: `Run <run_index>: <model_name>`.

Tip To rename an existing run, double-click the name in the work area and enter the new name, or modify the run name in the **Properties** pane.

Programmatic Use

You can programmatically check and modify the naming rule using the `Simulink.sdi.getRunNamingRule` and `Simulink.sdi.setRunNamingRule` functions. Restore

the naming rule to its default programmatically using the `Simulink.sdi.resetRunNamingRule` function.

Signal Metadata to Display

You can control which signal metadata is displayed in the work area of the **Inspect** pane and in the results section on the **Compare** pane in the Simulation Data Inspector. You specify the metadata to display separately for each pane using the **Table Columns** preferences in the **Inspect** and **Compare** sections of the Preferences dialog, respectively.

Inspect Pane

By default, the signal name and the line style and color used to plot the signal are displayed on the **Inspect** pane. To display different or additional metadata in the work area on the **Inspect** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Inspect** section. You can always view complete metadata for the selected signal in the **Inspect** pane using the **Properties** pane.

Note Metadata displayed in the work area on **Inspect** pane is included when you generate a report of plotted signals. You can also specify metadata to include in the report regardless of what is displayed in the work area when you create the report programmatically using the `Simulink.sdi.report` function.

Compare Pane

By default, the **Compare** pane shows the signal name, the absolute and relative tolerances used in the signal comparison, and the maximum difference from the comparison result. To display different or additional metadata in the results on the **Compare** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Compare** section. You can always view complete metadata for the signals compared for a selected signal result using the **Properties** pane, where metadata that differs between the compared signals is highlighted. Signal metadata displayed on the **Compare** pane does not affect the contents of comparison reports.

Signal Selection on the Inspect Pane

You can configure how you select signals to plot on the selected subplot in the Simulation Data Inspector. By default, you use check boxes next to each signal to plot. You can also choose to plot signals based on selection in the work area. Use **Check Mode** when creating views and visualizations that represent findings and analysis of a data set. Use **Browse Mode** to quickly view and analyze data sets with a large number of signals.

For more information about creating visualizations using **Check Mode**, see “Create Plots Using the Simulation Data Inspector” (Simulink).

For more information about using **Browse Mode**, see “Visualize Many Logged Signals” (Simulink).

Note To use **Browse Mode**, your layout must include only **Time Plot** visualizations.

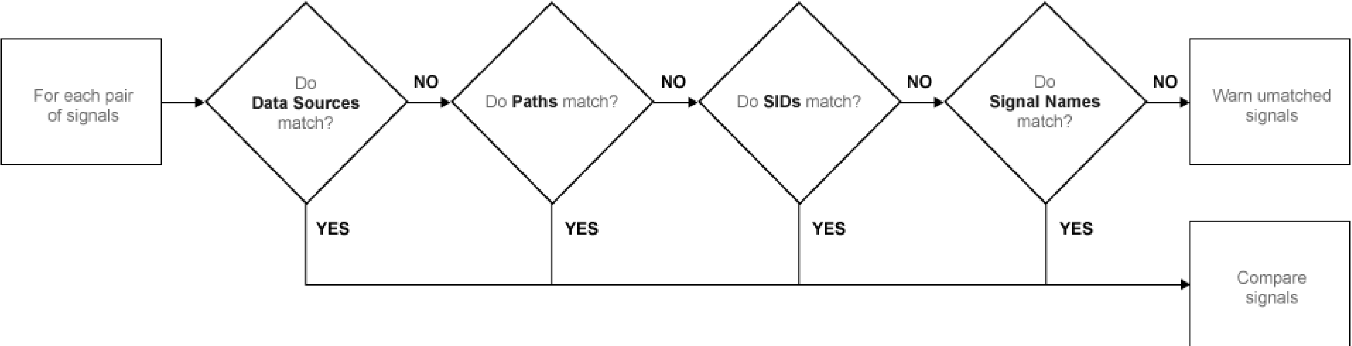
How Signals Are Aligned for Comparison

When you compare runs using the Simulation Data Inspector, the comparison algorithm pairs signals for signal comparison through a process called alignment. You can align signals between the compared runs using one or more of the signal properties shown in the table.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Simulink identifier For more information about SIDs, see “Simulink Identifiers” (Simulink)
Signal Name	Name of the signal

You can specify the priority for each piece of metadata used for alignment. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of **Then By** fields blank.

By default, the Simulation Data Inspector aligns signals between runs according to this flow chart.



For more information about configuring comparisons in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data” (Simulink).

Colors Used to Display Comparison Results

You can configure the colors used to display comparison results using the Simulation Data Inspector preferences. You can specify whether to use the signal color from the **Inspect** pane or a fixed color for the baseline and compared signals. You can also choose colors for the tolerance and the difference signal.

By default, the Simulation Data Inspector displays comparison results using fixed colors for the baseline and compared signals. Using a fixed color allows you to avoid the baseline signal color and compared signal color being either the same or too similar to distinguish.

Signal Grouping

You can specify how to group signals within a run in the Simulation Data Inspector. The preferences apply to both the **Inspect** and **Compare** panes. You can group signals by:

- Domain — Signal type. For example, signals created by signal logging have a domain of `Signal`, while signals created from logging model outputs have a domain of `Outports`.
- Physical System Hierarchy — Signal Simscape™ physical system hierarchy. The option to group by physical system hierarchy is available when you have a Simscape license.
- Data Hierarchy — Signal location within structured data. For example, data hierarchy grouping reflects the hierarchy of a bus.
- Model Hierarchy — Signal location within model hierarchy. Grouping by model hierarchy can be helpful when you log data from a model that includes model or subsystem references.

Grouping signals adds rows for the hierarchical nodes, which you can expand to show the signals within that node. By default, the Simulation Data Inspector groups signals by domain, then by physical system hierarchy (if you have a Simscape license), and then by data hierarchy.

To remove grouping and display a flat list of signals in each run, select `None` for all grouping options.

Programmatic Use

To specify how to group signals programmatically, use the `Simulink.sdi.setTableGrouping` function.

Data to Stream from Parallel Simulations

When you run parallel simulations using the `parsim` function, you can stream logged simulation data to the Simulation Data Inspector. A dot next to the run name in the **Inspect** pane indicates the status of the simulation that corresponds to the run, so you can monitor simulation progress while visualizing the streamed data. You can control whether data streams from a parallel simulation based on the type of worker the data comes from.

By default, the Simulation Data Inspector is configured for manual import of data from parallel workers. You can use the Simulation Data Inspector programmatic interface to inspect the data on the worker and decide whether to send it to the client Simulation Data Inspector for further analysis. To manually move data from a parallel worker to the Simulation Data Inspector, use the `Simulink.sdi.sendWorkerRunToClient` function.

You may want to automatically stream data from parallel simulations that run on local workers or on local and remote workers. Streaming data from both local and remote workers may affect simulation performance, depending on how many simulations you run and how much data you log. When you choose to stream data from local workers or all parallel workers, all logged simulation data automatically shows in the Simulation Data Inspector.

Programmatic Use

You can configure Simulation Data Inspector support for parallel worker data programmatically using the `Simulink.sdi.enablePCTSupport` function.

Options for Saving and Loading Session Files

You can specify a maximum amount of memory to use while loading or saving a session file. By default, the Simulation Data Inspector uses a maximum of 100 MB of memory when you load or save a session file. You can specify a memory use limit as low as 50 MB.

To reduce the size of the saved session file, you can specify a compression option.

- `None` — Do not compress saved data.
- `Normal` — Compress the saved file as much as possible.
- `Fastest` — Compress the saved file less than `Normal` compression for faster save time.

Signal Display Units

Signals in the Simulation Data Inspector have two units properties: stored units and display units. The stored units represent the units of the data saved to disk. The display units specify how the Simulation Data Inspector displays the data. You can configure the Simulation Data Inspector to use a system of units to define the display units for all signals. You can choose either the **SI** or **US Customary** system of units, or you can display data using its stored units.

When you use a system of units to define display units for signals in the Simulation Data Inspector, the display units update for any signal with display units that are not valid for that unit system. For example, if you select **SI** units, the display units for a signal may update from `ft` to `m`.

Note The system of units you choose to use in the Simulation Data Inspector does not affect the stored units for any signal. You can convert the stored units for a signal using the `convertUnits` function. Conversion may result in loss of precision.

In addition to selecting a system of units, you can specify override units so that all signals of a given measurement type are displayed using consistent units. For example, if you want to visualize all signals that represent weight using units of `kg`, specify `kg` as an override unit.

Tip For a list of units supported by Simulink, enter `showunitslist` in the MATLAB Command Window.

You can also modify the display units for a specific signal using the **Properties** pane. For more information, see "Modify Signal Properties in the Simulation Data Inspector" (Simulink).

Programmatic Use

Configure the unit system and override units using the `Simulink.sdi.setUnitSystem` function. You can check the current units preferences using the `Simulink.sdi.getUnitSystem` function.

See Also

Functions

`Simulink.sdi.clearPreferences` | `Simulink.sdi.enablePCTSupport` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode` | `Simulink.sdi.setRunNamingRule` | `Simulink.sdi.setTableGrouping`

More About

- “Iterate Model Design Using the Simulation Data Inspector” (Simulink)
- “How the Simulation Data Inspector Compares Data” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Create Plots Using the Simulation Data Inspector” (Simulink)
- “Modify Signal Properties in the Simulation Data Inspector” (Simulink)

How the Simulation Data Inspector Compares Data

You can tailor the Simulation Data Inspector comparison process to fit your requirements in multiple ways. When comparing runs, the Simulation Data Inspector:

1. Aligns signal pairs in the **Baseline** and **Compare To** runs based on the **Alignment** settings.

   The Simulation Data Inspector does not compare signals that it cannot align.
2. Synchronizes aligned signal pairs according to the specified **Sync Method**.

   Values for time points added in synchronization are interpolated according to the specified **Interpolation Method**.
3. Computes the difference of the signal pairs.
4. Compares the difference result against specified tolerances.

When the comparison run completes, the results of the comparison are displayed in the navigation pane.

Status	Comparison Result
	Difference falls within the specified tolerance.
	Difference violates specified tolerance.
	The signal does not align with a signal from the **Compare To** run.

When you compare signals with differing time intervals, the Simulation Data Inspector compares the signals on their overlapping interval.

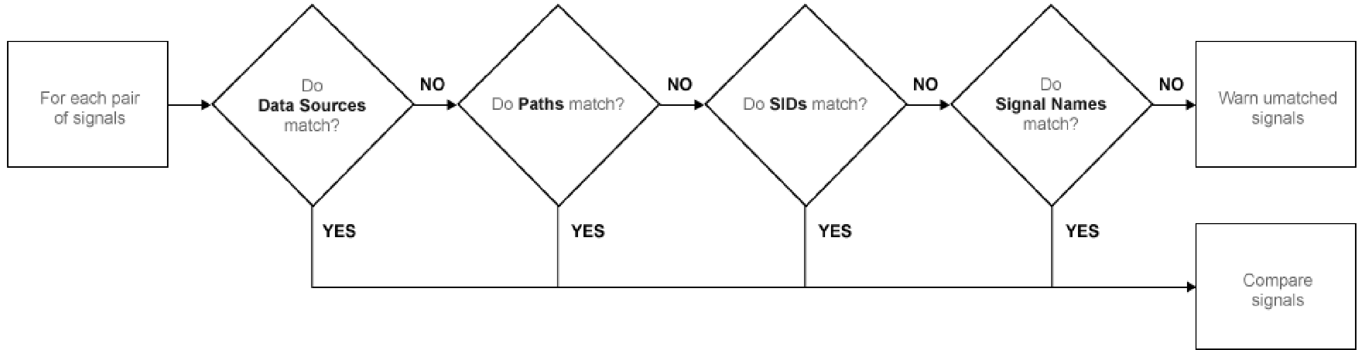
Signal Alignment

In the alignment step, the Simulation Data Inspector decides which signal from the **Compare To** run pairs with a given signal in the **Baseline** run. When you compare signals with the Simulation Data Inspector, you complete the alignment step by selecting the **Baseline** and **Compare To** signals.

The Simulation Data Inspector aligns signals using a combination of their Data Source, Path, SID, and Signal Name properties.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Simulink identifier For more information about SIDs, see “Simulink Identifiers” (Simulink)
Signal Name	Name of the signal in the model

With the default alignment settings, the Simulation Data Inspector aligns signals between runs according to this flow chart.

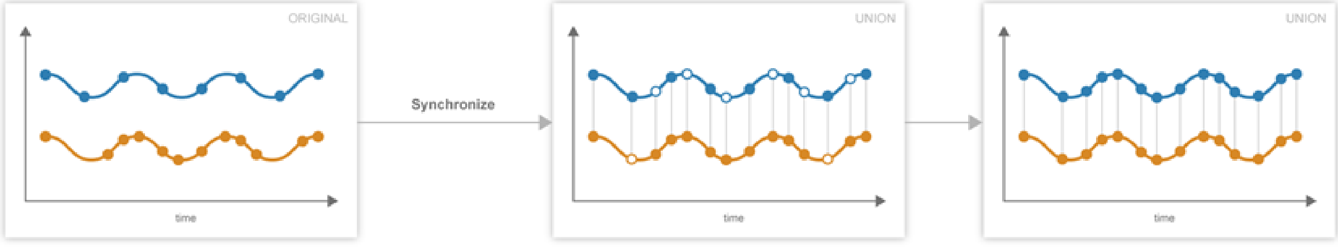


You can specify the priority for each of the signal properties used for alignment in the Simulation Data Inspector **Preferences**. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of the **Then By** fields blank.

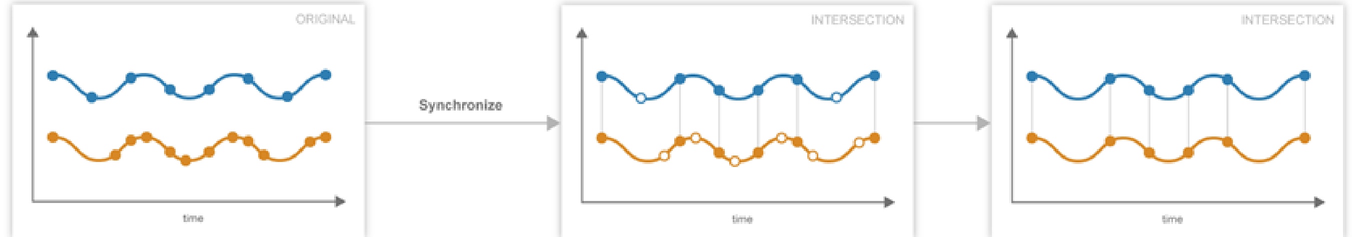
Synchronization

Often, signals that you want to compare don't contain the exact same set of time points. The synchronization step in Simulation Data Inspector comparisons resolves discrepancies in signals' time vectors. You can choose `union` or `intersection` as the synchronization method.

When you specify `union` synchronization, the Simulation Data Inspector builds a time vector that includes every sample time between the two signals. For each sample time not originally present in either signal, the Simulation Data Inspector interpolates the value. The second graph in the illustration shows the union synchronization process, where the Simulation Data Inspector identifies samples to add in each signal, represented by the unfilled circles. The final plot shows the signals after the Simulation Data Inspector has interpolated values for the added time points. The Simulation Data Inspector computes the difference using the signals in the final graph, so that the computed difference signal contains all the data points between the signals.



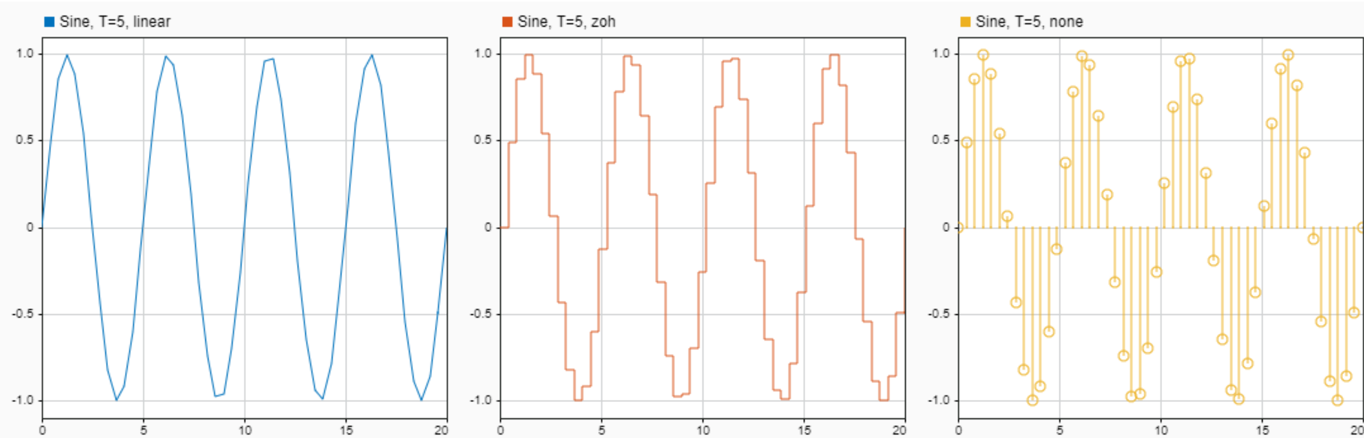
When you specify `intersection` synchronization, the Simulation Data Inspector uses only the sample times present in both signals in the comparison. In the second graph, the Simulation Data Inspector identifies samples that do not have a corresponding sample for comparison, shown as unfilled circles. The final graph shows the signals used for the comparison, without the samples identified in the second graph.



The choice between the synchronization options involves a trade off between speed and accuracy. The interpolation required by `union` synchronization takes time, but provides a more precise result. When you use `intersection` synchronization, the comparison finishes quickly because the Simulation Data Inspector computes the difference for fewer data points and does not interpolate. However, some data is discarded and precision lost with `intersection` synchronization.

Interpolation

The interpolation property of a signal determines how the Simulation Data Inspector displays the signal and how additional data values are computed in synchronization. You can choose to interpolate your data with a zero-order hold (zoh) or a linear approximation. You can also specify no interpolation.



When you specify `zoh` or `none` for the **Interpolation Method**, the Simulation Data Inspector replicates the data of the previous sample for interpolated sample times. When you specify `linear` interpolation, the Simulation Data Inspector uses samples on either side of the interpolated point to linearly approximate the interpolated value. Typically, discrete signals use `zoh` interpolation and continuous signals use `linear` interpolation. You can specify the **Interpolation Method** for your signals in the signal properties.

Tolerance Specification

The Simulation Data Inspector allows you to specify the scope and value of the tolerance for your signal. You can define a tolerance band using any combination of absolute, relative, and time tolerance values, and you can specify whether the specified tolerance applies to an individual signal or to all the signals in a run.

Tolerance Scope

In the Simulation Data Inspector, you can specify the tolerance for your data globally or for an individual signal. Global tolerance values apply to all signals in a run that do not have **Override Global Tol** set to `yes`. You can specify global tolerance values for your data at the top of the graphical viewing area in the **Compare** view. To specify signal specific tolerance values, edit the signal properties and ensure the **Override Global Tol** property is set to `yes`.

Tolerance Computation

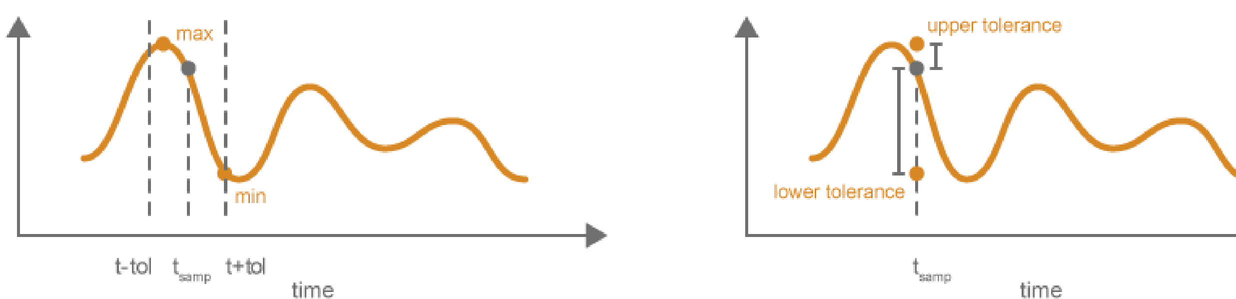
In the Simulation Data Inspector, you can specify a tolerance band for your run or signal using a combination of absolute, relative, and time tolerance values. When you specify the tolerance for your run or signal using multiple types of tolerances, each tolerance can yield a different answer for the tolerance at each point. The Simulation Data Inspector computes the overall tolerance band by selecting the most lenient tolerance result for each data point.

When you define your tolerance using only the absolute and relative tolerance properties, the Simulation Data Inspector computes the tolerance for each point as a simple maximum.

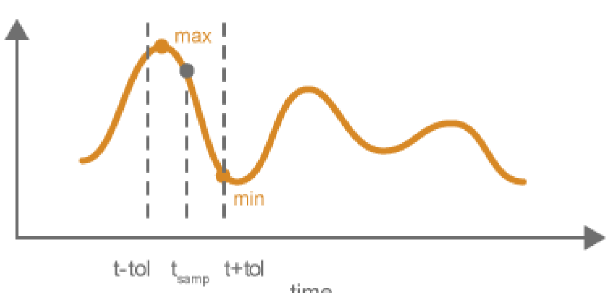
```
tolerance = max(absoluteTolerance,relativeTolerance*abs(baselineData));
```

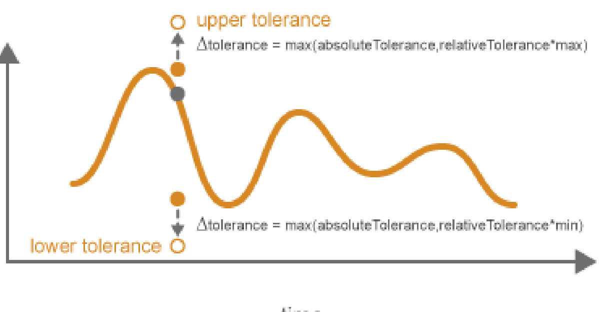
The upper boundary of the tolerance band is formed by adding `tolerance` to the **Baseline** signal. Similarly, the Simulation Data Inspector computes the lower boundary of the tolerance band by subtracting `tolerance` from the **Baseline** signal.

When you specify a time tolerance, the Simulation Data Inspector evaluates the time tolerance first, over a time interval defined as $[(t_{samp}-tol), (t_{samp}+tol)]$ for each sample. The Simulation Data Inspector builds the lower tolerance band by selecting the minimum point on the interval for each sample. Similarly, the maximum point on the interval defines the upper tolerance for each sample.



If you specify a tolerance band using an absolute or relative tolerance in addition to a time tolerance, the Simulation Data Inspector applies the time tolerance first, and then applies the absolute and relative tolerances to the maximum and minimum points selected with the time tolerance.






```
upperTolerance = max + max(absoluteTolerance,relativeTolerance*max)

lowerTolerance = min - max(absoluteTolerance,relativeTolerance*min)
```

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types `int64` or `uint64`.
- Variable-size signals.

See Also

Related Examples

- "Compare Simulation Data" (Simulink)

Save and Share Simulation Data Inspector Data and Views

After you inspect, analyze, or compare your data in the Simulation Data Inspector, you can share your results with others. The Simulation Data Inspector provides several options for sharing and saving your data and results, depending on your needs. With the Simulation Data Inspector, you can:

- Save your data and layout modifications in a Simulation Data Inspector session.
- Share your layout modifications in a Simulation Data Inspector view.
- Share images and figures of plots you create in the Simulation Data Inspector.
- Create a Simulation Data Inspector report.
- Export your data from the Simulation Data Inspector.

Save and Load Simulation Data Inspector Sessions

If you want to save or share data along with a configured view in the Simulation Data Inspector, save your data and settings in a Simulation Data Inspector session. You can save sessions as MAT- or MLDATX-files. The default format is MLDATX. When you save a Simulation Data Inspector session, the session file contains:

- All runs, data, and properties from the **Inspect** pane, including which run is the current run and which runs are in the archive.
- Plot display selection for signals in the **Inspect** pane.
- Subplot layout and line style and color selections.

Note Comparison results and global tolerances are not saved in Simulation Data Inspector sessions.

To save a Simulation Data Inspector session:

1. Hover over the save icon on the left side bar. Then, click **Save As**.

2. Name the file.
3. Browse to the location where you want to save the session, and click **Save**.

   For large datasets, a status overlay in the bottom right of the graphical viewing area displays information about the progress of the save operation and allows you to cancel the save operation.

The **Save** tab of the Simulation Data Inspector preferences menu on the left side bar allows you to configure options related to save operations for MLDATX-files. You can set a limit as low as 50MB on the amount of memory used for the save operation. You can also select one of three **Compression** options:

- `None`, the default, applies no compression during the save operation.
- `Normal` creates the smallest file size.
- `Fastest` creates a smaller file size than you would get by selecting `None`, but provides a faster save time than `Normal`.

To load a Simulation Data Inspector session, click the open icon on the left side bar. Then, browse to select the MLDATX-file you want to open, and click **Open**.

Alternatively, you can double-click the MLDATX-file. MATLAB and the Simulation Data Inspector open if they are not already open.

When the Simulation Data Inspector already contains runs and you open a session, all of the runs in the session move to the archive. The view updates to reflect show plotted signals from the session file. You can drag runs between the work area and archive as desired.

When the Simulation Data Inspector does not contain runs and you open a session, the Simulation Data Inspector puts runs in the work area and archive as specified in the file.

Share Simulation Data Inspector Views

When you have different sets of data that you want to visualize the same way, you can save a view. A view saves the layout and appearance characteristics of the Simulation Data Inspector without saving the data. Specifically, a view saves:

- Plot layout, axis ranges, linking characteristics, and normalized axes.
- Location of signals in the plots, including plotted signals in the archive.
- Signal grouping and columns on display in the **Inspect** pane.
- Signal color and line styling.

To save a view:

1. Click the layout button .
2. Click **Save current view**.
3. In the dialog box, specify a name for the view and browse to the location where you want to save the MLDATX-file.
4. Click **Save**.

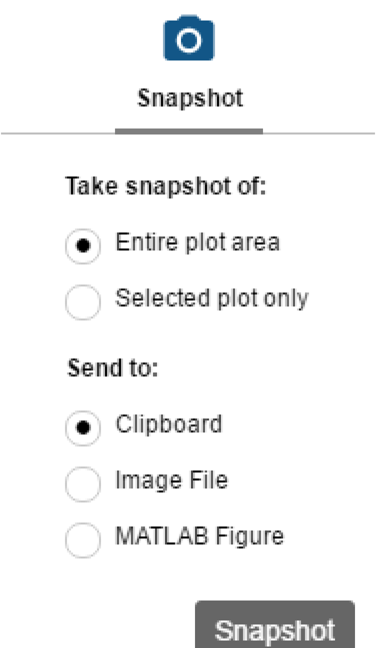
To load a view:

1. Click the layout button .
2. Click **Open saved view**.
3. Browse to the view you would like to load, and click **Open**.

Share Simulation Data Inspector Plots

Use the snapshot feature to share the plots you generate in the Simulation Data Inspector. You can export your plots to the clipboard to paste into a document, as an image file, or to a MATLAB figure. You can choose to capture the entire plot area, including all subplots in the plot area, or to capture only the selected subplot.

Click the camera icon on the toolbar to access the snapshot menu. Use the radio buttons to select the area you want to share and how you want to share the plot. After you make your selections, click **Snapshot** to export the plot.



If you create an image, select where you would like to save the image in the file browser.

You can create snapshots of your plots in the Simulation Data Inspector programmatically using `Simulink.sdi.snapshot`.

Create a Simulation Data Inspector Report

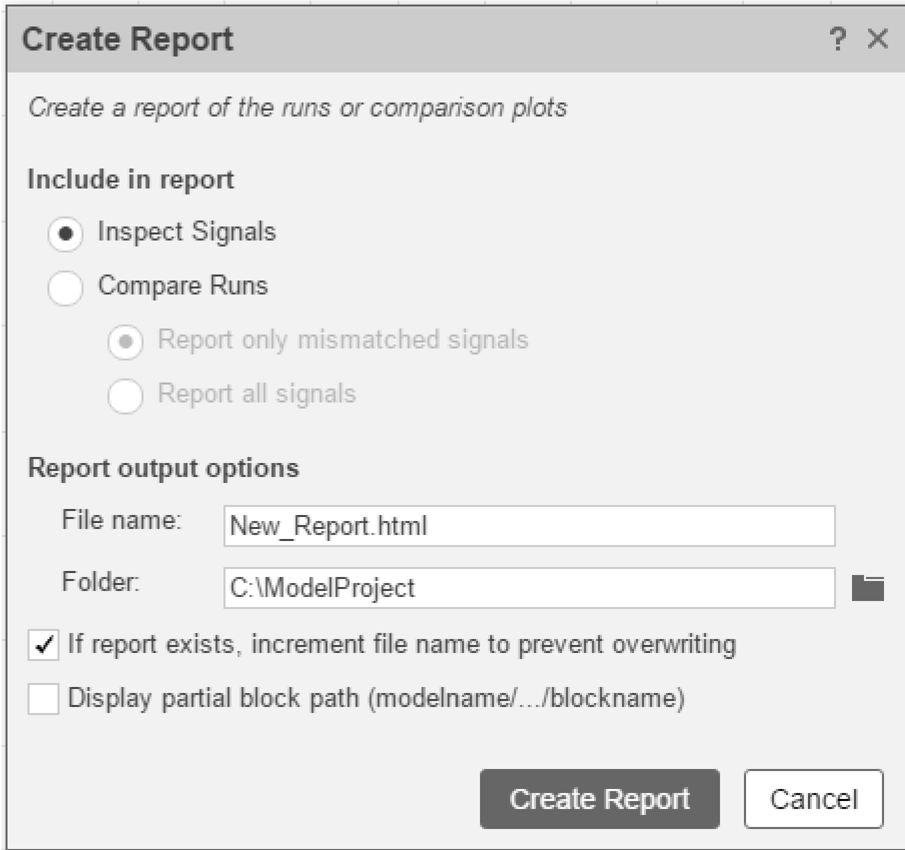
To generate documentation of your results quickly, create a Simulation Data Inspector report. You can create a report of your data in either the **Inspect** or the **Compare** pane. The report is an HTML file that includes information about all the signals and plots in the active pane. The report includes all signal information displayed in the signal table in the navigation pane. For more information about configuring the table, see “Inspect Metadata” (Simulink).

To generate a Simulation Data Inspector Report:

1. Click the create report icon on the left side bar.
2. Under **Include in report**, specify the type of report you want to create.

   - Select **Inspect Signals** to include the plots and signals from the **Inspect** pane.
   - Select **Compare Runs** to include the data and plots from the **Compare** pane. When you generate a **Compare Runs** report, you can choose to **Report only mismatched signals** or

to **Report all signals**. If you select **Report only mismatched signals**, the report shows only signal comparisons that are not within the specified tolerances.



3. Specify a **File name** for the report, and navigate to the **Folder** where you want to save the report.
4. Click **Create Report**.

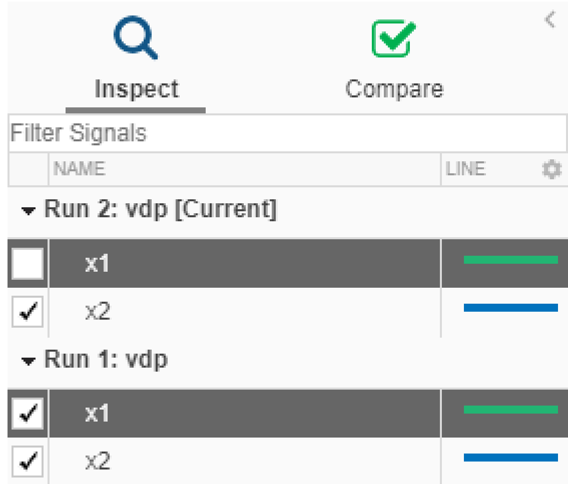
   The generated report automatically opens in your default browser.

Export Data from the Simulation Data Inspector

You can use the Simulation Data Inspector to export data to the base workspace, a MAT-file, or a Microsoft Excel file. You can export a selection of runs and signals, runs in the work area, or all runs in the **Inspect** pane, including the **Archive**.

When you export a selection of runs and signals, make the selection of data to export before clicking the export button.

Only the selected runs and signals are exported. In this example, only the `x1` signals from Run 1 and Run 2 are exported. The check box selections for the plotting area do not affect whether a signal is exported.



When you export a single signal, the signal is stored in `timeseries` format in the workspace variable or MAT-file. Exported data for a run or multiple signals is stored in `Simulink.SimulationData.Dataset` format.

Note When you export a run that contains logged parameter data, the exported `Simulink.SimulationData.Dataset` contains a `Simulink.SimulationData.Parameter` element for each logged parameter.

To export data to a file, select the **File** option in the **Export** dialog. You can specify a file name and browse to the location where you want to save the exported file. When you export data to a MAT-file, a single exported signal is stored in `timeseries` format, and runs or multiple signals are stored in `Simulink.SimulationData.Dataset` format. When you export data to a Microsoft Excel file, the data is stored in the format described in “Microsoft Excel Import, Export, and Logging Format” (Simulink).

To export to a Microsoft Excel file, select the XLSX extension from the drop-down. When you export data to a Microsoft Excel file, you can specify additional options for the format of the data in the exported file. If the file name you provided already exists, you can choose to overwrite the entire file or to only overwrite sheets containing data that corresponds to the exported data. You can also choose which metadata to include and whether signals with identical time data share a time column in the exported file.

See Also

Related Examples

- “View Data in the Simulation Data Inspector” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)

Inspect and Compare Data Programmatically

You can harness the capabilities of the Simulation Data Inspector from the MATLAB command line using the Simulation Data Inspector API.

The Simulation Data Inspector organizes data in runs and signals, assigning a unique numeric identification to each run and signal. Some Simulation Data Inspector API functions use the run and signal IDs to reference data, rather than accepting the run or signal itself as an input. To access the run IDs in the workspace, you can use `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`. You can access signal IDs through a `Simulink.sdi.Run` object using the `getSignalIDByIndex` method.

The `Simulink.sdi.Run` and `Simulink.sdi.Signal` classes provide access to your data and allow you to view and modify run and signal metadata. You can modify the Simulation Data Inspector preferences using functions like `Simulink.sdi.setSubPlotLayout`, `Simulink.sdi.setRunNamingRule`, and `Simulink.sdi.setMarkersOn`. To restore the Simulation Data Inspector's default settings, use `Simulink.sdi.clearPreferences`.

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

Create `timeseries` objects to contain data for a sine signal and a cosine signal. Give each `timeseries` object a descriptive name.

```
time = linspace(0,20,100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Run and Add Data

Use the `Simulink.sdi.view` function to open the Simulation Data Inspector.

```
Simulink.sdi.view
```

To import data into the Simulation Data Inspector from the workspace, create a `Simulink.sdi.Run` object using the `Simulink.sdi.Run.create` function. Add information about the run to its metadata using the `Name` and `Description` properties of the `Run` object.

```
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';
```

Use the `add` function to add the data you created in the workspace to the empty run.

```
add(sinusoidsRun,'vars',sine_ts,cos_ts);
```

Plot the Data in the Simulation Data Inspector

Use the getSignalByIndex function to access Simulink.sdi.Signal objects that contain the signal data. You can use the Simulink.sdi.Signal object properties to specify the line style and color for the signal and plot it in the Simulation Data Inspector. Specify the LineColor and LineDashed properties for each signal.

```
sine_sig = getSignalByIndex(sinusoidsRun,1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';

cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.LineDashed = '--';
```

Use the Simulink.sdi.setSubPlotLayout function to configure a 2-by-1 subplot layout in the Simulation Data Inspector plotting area. Then use the plotOnSubplot function to plot the sine signal on the top subplot and the cosine signal on the lower subplot.

```
Simulink.sdi.setSubPlotLayout(2,1);

plotOnSubPlot(sine_sig,1,1,true);
plotOnSubPlot(cos_sig,2,1,true);
```

Close the Simulation Data Inspector and Save Your Data

When you have finished inspecting the plotted signal data, you can close the Simulation Data Inspector and save the session to an MLDATX file.

```
Simulink.sdi.close('sinusoids.mldatx')
```

Compare Two Signals in the Same Run

You can use the Simulation Data Inspector programmatic interface to compare signals within a single run. This example compares the input and output signals of an aircraft longitudinal controller.

First, load the session that contains the data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the Simulink.sdi.Run.getLatest function to access the latest run in the data.

```
aircraftRun = Simulink.sdi.Run.getLatest;
```

Then, you can use the Simulink.sdi.getSignalsByName function to access the Stick signal, which represents the input to the controller, and the alpha, rad signal that represents the output.

```
stick = getSignalsByName(aircraftRun,'Stick');
alpha = getSignalsByName(aircraftRun,'alpha, rad');
```

Before you compare the signals, you can specify a tolerance value to use for the comparison. Comparisons use tolerance values specified for the baseline signal in the comparison, so set an absolute tolerance value of 0.1 on the Stick signal.

```
stick.AbsTol = 0.1;
```

Now, compare the signals using the `Simulink.sdi.compareSignals` function. The `Stick` signal is the baseline, and the `alpha, rad` signal is the signal to compare against the baseline.

```
comparisonResults = Simulink.sdi.compareSignals(stick.ID,alpha.ID);
match = comparisonResults.Status
```

```
match = 
OutOfTolerance
```

The comparison result is out of tolerance. You can use the `Simulink.sdi.view` function to open the Simulation Data Inspector to view and analyze the comparison results.

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` (Simulink) function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` (Simulink) function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 3
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` (Simulink) function.

```
saveResult(runResult,'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status
```

```
ans =
OutOfTolerance
```

```
alphaResult.Status
```

```
ans =
OutOfTolerance
```

The comparison used a value of `0` for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```

Specify an absolute tolerance of `0.1` and a time tolerance of `0.6` for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);

qResult2.Status
```

```
ans =
WithinTolerance
```

```
alphaResult2.Status
```

```
ans =
WithinTolerance
```

See Also

Simulation Data Inspector

Related Examples

- “Compare Simulation Data” (Simulink)
- “How the Simulation Data Inspector Compares Data” (Simulink)
- “Create Plots Using the Simulation Data Inspector” (Simulink)

Limit the Size of Logged Data

In this section...

"Limit the Number of Runs Retained in the Simulation Data Inspector Archive" on page 40-47
"Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data" on page 40-47
"View Data Only During Simulation" on page 40-48
"Reduce the Number of Data Points Logged from Simulation" on page 40-48

Logging simulation data can produce large amounts of data that fill up disk space. Such situations include logging many signals, logging data for long simulations, and running many simulations without deleting run data from the Simulation Data Inspector. You can choose among several options to limit the size of logged simulation data. You can:

- Limit the number of runs retained in the Simulation Data Inspector archive.
- Reduce the number of data points logged in each simulation.
- Specify a minimum disk space requirement or maximum size for logged data.
- Configure logging for only viewing data during simulation.

Depending on your requirements, you can use more than one strategy to limit the size of logged data.

Limit the Number of Runs Retained in the Simulation Data Inspector Archive

When you run multiple simulations in a single MATLAB session, logged simulation data accumulates in the Simulation Data Inspector even if you overwrite the logging data in the MATLAB workspace. To reduce the amount of data the Simulation Data Inspector retains, you can configure a limit for the number of runs stored in the archive. When the number of runs in the archive reaches the size limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis.

Configure the archive **Size** setting in the Simulation Data Inspector preferences. The size limit only applies to runs in the archive. For the Simulation Data Inspector to automatically limit data retention, select **Automatically archive** and specify the maximum number of runs to retain in the archive. By default, **Automatically archive** is enabled with an archive size limit of twenty runs. If you experience issues with logged data consuming too much disk space, consider adjusting the size limit for the archive in the Simulation Data Inspector preferences.

Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data

You can use preferences in the Simulation Data Inspector to directly limit the size of logged data by specifying a minimum amount of disk space to leave free or by specifying a maximum size for logged data on disk. Each setting accounts for all kinds of logged data. By default, logged data must leave at least 100 MB of free disk space with no maximum size limit. Specify the required disk space and maximum size in GB, and specify `0` to apply no disk space requirement or no maximum size limit.

When you specify a minimum disk space requirement or a maximum size for logged data, you can also specify whether to prioritize retaining data from the current simulation or data from prior simulations when approaching the limit. By default, the Simulation Data Inspector prioritizes

retaining data for the current run. As the free disk space or logged data size approaches the limit, prior runs are deleted first to free up space for data being logged in the current run. If deleting runs does not free up enough space, recording is disabled. To prioritize retaining prior data, change the **When low on disk space** setting to **Keep prior runs and stop recording**. You see a warning message when prior runs are deleted and when recording is disabled. If recording is disabled due to the size of logged data, you need to change the **Record Mode** back to **View and record data** to continue logging data, after you have freed up disk space.

View Data Only During Simulation

In some situations, you may want to only view the data for logged signals and not save the values. For example, when using the Simulation Data Inspector to visualize data streaming from hardware, you may only want to view the data live and not record it. You can change the **Record mode** in the Simulation Data Inspector preferences to **View during simulation only** so that logged data is not saved and you can still view the data during simulation. The **Record mode** is reset to **View and record data** at the start of each MATLAB session.

When you change the **Record mode** to **View during simulation only**:

- Logged data is not available in the Simulation Data Inspector or workspace after simulation.
- You can view data using dashboard blocks, scopes, and the Simulation Data Inspector, but plots clear when you pan or zoom.
- You cannot access logged data during simulation using the Simulation Data Inspector programmatic interface.

Reduce the Number of Data Points Logged from Simulation

Model configuration parameters and signal properties allow you to limit the number of data points logged in a simulation. Be sure to carefully consider data requirements when limiting logged data points. Limiting data can skip critical time points, and can lead to aliasing, if your effective sample rate is too low.

You can reduce the number of data points using:

- Decimation — Log every *n*th signal value.
- Limit data points to last — Only log the last *n* signal values.
- Logging intervals — Specify specific time intervals in which to log data.

For details, see “Specify Signal Values to Log” (Simulink).

See Also

Tools
Simulation Data Inspector

Related Examples

- “Specify Signal Values to Log” (Simulink)
- “Configure the Simulation Data Inspector” (Simulink)